Semantics of Type Systems
Lecture Notes

Derek Dreyer
MPI-SWS

Ralf Jung
MPI-SWS

Jan-Oliver Kaiser
MPI-SWS

Hoang-Hai Dang
MPI-SWS

David Swasey
MPI-SWS

Jan Menz
MPI-SWS

Lennard Gäher
MPI-SWS

Simon Spies
MPI-SWS

February 14, 2022

Contents

1 Simply Typed Lambda Calculus

$$\begin{array}{rrl}
\text{Variables} & x, y & \quad \dots \\
\text{Runtime Terms} & e & ::= \; x \mid \lambda x.\, e \mid e_1\, e_2 \mid e_1 + e_2 \mid \overline{n} \\
\text{(Runtime) Values} & v & ::= \; \lambda x.\, e \mid \overline{n}
\end{array}$$

Variables and Substitution In the simply typed λ-calculus, variables and substitution are instrumental to define how terms are evaluated (called the *operational semantics*). During evaluation, if we apply a λ-abstraction $\lambda x.\, e$ to a value v, then the variable x is *substituted* with the value v in e. For example, the term $(\lambda x.\, x + \overline{1})\, \overline{41}$ proceeds with $\overline{41} + \overline{1}$ in the next step of the evaluation.

We write $e[e'/x]$ for the substitution operation that replaces x with e' in e. We define it recursively by:

$$\begin{aligned}
y[e'/x] &:= e' && \text{if}\, x = y \\
y[e'/x] &:= y && \text{if}\, x \neq y \\
(\lambda y.\, e)[e'/x] &:= \lambda y.\, e && \text{if}\, x = y \\
(\lambda y.\, e)[e'/x] &:= \lambda y.\, (e[e'/x]) && \text{if}\, x \neq y \\
(e_1\, e_2)[e'/x] &:= (e_1[e'/x])\, (e_2[e'/x]) \\
(e_1 + e_2)[e'/x] &:= (e_1[e'/x]) + (e_2[e'/x]) \\
\overline{n}[e'/x] &:= \overline{n}
\end{aligned}$$

Getting substitution right can be tricky. This definition is typically considered incorrect. To explain what goes wrong, we have to distinguish between *free* and *bound* variables: A variable x is bound in a term if it appears inside of a binder λx (*e.g.*, x is bound in $\lambda x.\, x + y$). All other variables are called *free* (*e.g.*, y is free in $\lambda x.\, x + y$).

The problem with the naive definition of substitution above is called *variable capturing.* Variable capturing occurs if we insert an expression with a free variable such as $\overline{2} + x$ into an expression which binds the free variable. For example, if we naively substitute $\overline{2} + x$ for y in $\lambda x.\, y + x$, then we obtain $\lambda x.\, (\overline{2} + x) + x$. Since x was free in $\overline{2} + x$ but is bound in $\lambda x.\, (\overline{2}+x)+x$, one speaks of variable capturing. Variable capturing is problematic, because the programmer would have to anticipate which variables are free in function arguments.

To avoid variable capturing, a correct substitution renames bound variables where conflicts arise. For example, $(\lambda x.\, y + x)[\overline{2} + x/y]$ would result in $\lambda z.\, (\overline{2} + x) + z$, such that x remains free in the resulting term. Unfortunately, defining (and reasoning about) a substitution operation that properly renames bound variables is oftentimes tedious, especially in proof assistants. Thus, on paper, people typically rely on *Barendregt's variable convention* [3]: all bound and free variables are distinct and this invariant is maintained implicitly.

Since we *mechanize* our proofs in Coq, we cannot assume Barendregt's variable convention. Instead, we use the (slightly broken) substitution operation above. In our use cases, the substitution will only insert *closed* terms (*i.e.*, terms without any free variables), which avoids the problem of variable capture entirely. (In later sections, we will discuss the more complicated DeBruijn representation, which makes it easy to define a substitution which avoids variable capturing.)

1.1 Operational Semantics

In order to reason about programs, we have to assign a semantics to them. In the following, we assign an *operational semantics* to programs—we describe how runtime terms are evaluated. We distinguish three different operational semantics: *structural semantics*, *contextual semantics*, and *big-step semantics.*

Structural Semantics $\boxed{e \succ e'}$

We define a structural *call-by-value, right-to-left* operational semantics on our runtime terms. This means we do not allow reduction below lambda abstraction, and we always evaluate the right term to a value, before we start evaluating the left term.

$$\textsc{app-struct-r}\ \frac{e_2 \succ e_2'}{e_1\, e_2 \succ e_1\, e_2'} \qquad \textsc{app-struct-l}\ \frac{e_1 \succ e_1'}{e_1\, v \succ e_1'\, v} \qquad \textsc{beta}\ (\lambda x.\, e)\, v \succ e[v/x]$$

$$\textsc{plus-struct-l}\ \frac{e_1 \succ e_1'}{e_1 + v \succ e_1' + v} \qquad \textsc{plus-struct-r}\ \frac{e_2 \succ e_2'}{e_1 + e_2 \succ e_1 + e_2'} \qquad \textsc{plus}\ \overline{n} + \overline{m} \succ \overline{n + m}$$

Exercise 1 Prove that the structural semantics $e \succ e'$ is *deterministic.* That is, show that if $e \succ e'$ and $e \succ e''$, then $e' = e''$. ●

Exercise 2 The semantics $e \succ e'$ is a *call-by-value* semantics, meaning arguments are evaluated to values before they are inserted into lambda abstractions. The call-by-value approach is used by many programming languages (e.g., Java, C, Standard ML, and OCaml). Alternatively, one can defer the evaluation of function arguments and, instead, insert their unevaluated form directly into lambda abstractions. This style of operational semantics is called *call-by-name* semantics and is followed by some functional languages (e.g., Haskell). The core rule of this semantics is:

$$\textsc{cbn-beta}\ (\lambda x.\, e)\, e' \succ_{\mathsf{cbn}} e[e'/x]$$

a) Complete the definition of $e \succ_{\mathsf{cbn}} e'$ and give one expression, which evaluates to different values under *call-by-name* and *call-by-value* semantics.
Hint: For call-by-name, the left side of an application has to be evaluated first.

b) Prove that $e \succ_{\mathsf{cbn}} e'$ is deterministic.

●

Big-Step Semantics $\boxed{e \downarrow v}$

Sometimes, it will be convenient to use so-called big-step semantics. Whereas the (small-step) structural semantics describes step by step how an expression executes, the big-step semantics directly relates expressions to their final value.

$$\textsc{literal}\ \overline{n} \downarrow \overline{n} \qquad \textsc{lambda}\ \lambda x.\, e \downarrow \lambda x.\, e \qquad \textsc{app}\ \frac{e_1 \downarrow \lambda x.\, e \quad e_2 \downarrow v_2 \quad e[v_2/x] \downarrow v}{e_1\, e_2 \downarrow v} \qquad \textsc{plus}\ \frac{e_1 \downarrow \overline{n_1} \quad e_2 \downarrow \overline{n_2}}{e_1 + e_2 \downarrow \overline{n_1 + n_2}}$$

Exercise 3 Prove that the big-step and the small-step semantics $e \succ e'$ are equivalent:

$$e \downarrow v \quad \textit{iff} \quad e \succ^* v$$

●

Exercise 4 The evaluation order for $e \succ e'$ is right-to-left. We are now going to consider left-to-right evaluation order.

a) Define a semantics $e \succ_{\mathsf{ltr}} e'$, which evaluates expressions in left-to-right order.

b) Pick terms e and e' such that $e \succ_{\mathsf{ltr}} e'$ but not $e \succ e'$.

c) Show that both are equivalent if we evaluate to values.
Hint: It suffices to show $e \succ_{\mathsf{ltr}}^* v \quad \textit{iff} \quad e \downarrow v$.

d) Think of the programming languages you have encountered in your past. Is it in one of them possible to obtain different results, depending on left-to-right or right-to-left evaluation order?

●

Contextual Semantics

The structural semantics $e \succ e'$ has two kinds of rules: (1) rules such as APP-STRUCT-L, which descend into the term to find the next subterm to reduce (which is called a *redex*) and (2) rules such as BETA and PLUS, which reduce redexes. Next, we define a third operational semantics, the contextual operational semantics $e_1 \rightsquigarrow e_2$, which separates the search for the redex (*i.e.*, structurally descending in the term) from the reduction. While separating redex search and reduction does not have any immediate benefits for us at the moment, it will lead to more elegant reasoning principles later on.

For the contextual semantics, we first define evaluation contexts:

$$\text{Evaluation Contexts} \quad K ::= \bullet \mid K\,v \mid e\,K \mid K + v \mid e + K$$

Evaluation contexts are expressions with a hole (*e.g.*, $\bullet + \overline{41}$), which describe where in the expression the next redex can be found. We can fill the hole with an expression using the following function:

Context Filling $\boxed{K[e]}$

$$\begin{aligned} \bullet[e] &:= e & (K + v)[e] &:= K[e] + v \\ (K\,v)[e] &:= (K[e])\,v & (e' + K)[e] &:= e' + K[e] \\ (e'K)[e] &:= e'(K[e]) \end{aligned}$$

Base reduction and contextual reduction $\boxed{e_1 \rightsquigarrow_{\mathrm{b}} e_2 \text{ and } e_1 \rightsquigarrow e_2}$

$$\frac{}{(\lambda x.\, e)\, v \rightsquigarrow_{\mathrm{b}} e[v/x]}\,\textsc{beta} \qquad \frac{}{\overline{n} + \overline{m} \rightsquigarrow_{\mathrm{b}} \overline{n + m}}\,\textsc{plus} \qquad \frac{e_1 \rightsquigarrow_{\mathrm{b}} e_2}{K[e_1] \rightsquigarrow K[e_2]}\,\textsc{ctx}$$

Lemma 1 (Context Lifting)**.** *If* $e_1 \leadsto e_2$*, then* $K[e_1] \leadsto K[e_2]$.

Proof. Assume that $e_1 \leadsto e_2$. By inversion, there exist K', e'_1, e'_2 such that $e'_1 \leadsto_{\mathrm{b}} e'_2$ and $e_1 = K'[e'_1]$ and $e_2 = K'[e'_2]$. To construct a proof of $K[K'[e'_1]] \leadsto K[K'[e'_2]]$, we essentially need to compose the two contexts in order to apply CTX.

We define a context composition operation in the following. We can then close this proof by Lemma 2. □

Context Composition $\boxed{K_1 \circ K_2}$

$$\begin{aligned} \bullet \circ K_2 &:= K_2 & (K + v) \circ K_2 &:= (K \circ K_2) + v \\ (K\, v) \circ K_2 &:= (K \circ K_2)\, v & (e' + K) \circ K_2 &:= e' + (K \circ K_2) \\ (e'K) \circ K_2 &:= e'(K \circ K_2) \end{aligned}$$

Lemma 2. $K_1[K_2[e]] = (K_1 \circ K_2)[e]$

Proof. By induction on K_1. □

Exercise 5 Prove that the structural and the contextual semantics are equivalent:

$$e \succ e' \quad \textit{iff} \quad e \leadsto e'$$

●

1.2 The Untyped λ-calculus

Before we start to integrate *types* into our calculus (in Section 1.3), we first examine the *untyped* version of the lambda calculus. The untyped lambda calculus, even without addition and natural numbers, is quite expressive computationally—it is Turing complete! We will now explore its computational power in the fragment $e ::= x \mid \lambda x.\, e \mid e_1\, e_2$. The slogan is: *All you need are lambdas, variables, and beta reduction.*

Let us try to define some simple terms:

$$I := \lambda x.\, x \qquad F := \lambda x, y.\, x \qquad S := \lambda x, y.\, y \qquad \omega := \lambda x.\, xx \qquad \Omega := \omega\omega$$

I computes the identity function, while F and S evaluate to their first or second argument, respectively. ω and Ω are more interesting. ω applies its argument to itself, and Ω applies ω to itself. So let us try out what happens when we evaluate Ω.

$$\Omega = \omega\omega = (\lambda x.\, xx)\omega \succ (xx)[\omega/x] = \omega\omega = \Omega$$

As it turns out, Ω reduces to itself! Thus, there are terms in the untyped lambda calculus such as Ω which *diverge* (*i.e.*, their reduction chains do not terminate).

Scott encodings We can not only write diverging terms, but we can also encode inductive data types in the untyped lambda calculus. For example, we can encode natural numbers in their Peano representation (*i.e.*, with the constructors 0 and S) as lambda terms. The basic idea is to interpret natural numbers as "case distinctions". That is, each number will be an abstraction with two arguments, s and f, the cases. If the number is 0, then it will return the argument s. If the number is $\mathsf{S}n$, then it will return the result of f applied to the predecessor n.

$$\mathsf{zero} := \lambda s, f.\, s \qquad\qquad \mathsf{succ}(n) := \lambda s, f.\, f\ n$$

Note that here, the n in the definition of succ is *a lambda term.*

Following this principle, we can define an encoding function that defines the encoding of a number as a lambda term:

$$\begin{aligned}\mathsf{enc}(0) &= \mathsf{zero}\\ \mathsf{enc}(\mathsf{S}n) &= \mathsf{succ}(\mathsf{enc}(n))\end{aligned}$$

Let us write down a few numbers:

$$\begin{aligned}&\mathsf{enc}(0) = \mathsf{zero} = \lambda s, f.\, s\\ &\mathsf{enc}(1) = \mathsf{succ}(\mathsf{zero}) = \lambda s, f.\, f\ (\lambda s, f.\, s)\\ &\mathsf{enc}(2) = \mathsf{succ}(\mathsf{succ}(\mathsf{zero})) = \lambda s, f.\, f\ (\lambda s, f.\, f\ (\lambda s, f.\, s))\end{aligned}$$

We can use this representation, known as the Scott encoding, to compute with natural numbers. That is, since we can do a case analysis on the numbers *by definition*, we can distinguish them and compute different results depending on the case. For example, the function $\mathsf{pred} = \lambda n.\, n\ \mathsf{zero}\ I$ computes the predecessor of a Scott encoded natural number.

Exercise 6 Define a function on Scott encoded natural numbers which returns the identity for 0 and diverges for every other number. ●

Exercise 7 (Scott encoding of Booleans and Pairs) Similar to numbers, there are Scott encodings for all first-order inductive data types (*e.g.*, pairs and lists). These encodings allow us to work with structured data in the untyped lambda calculus. Define a Scott encoding for Booleans and pairs. ●

If we want to define slightly more interesting functions on our numbers such as addition $\mathsf{add}\ n\ m$, we run into a problem. The naive definition of addition would be:

$$\mathsf{add} := \lambda n, m.\, n\ m\ (\lambda n'.\, \mathsf{Succ}(\mathsf{add}\ n'\ m)) \qquad \text{where } \mathsf{Succ} := \lambda n, s, f.\, f\ n$$

Unfortunately, this definition is broken! The term that we want to define, add, occurs in its own definition on the right hand side. To fix this problem, we are now going to develop a mechanism to add recursion to the untyped lambda calculus.

Recursion To enable recursion, we define a recursion operator fix (sometimes called fix-point combinator or Y-combinator). The idea of fix is that given a template $\lambda f, x.\, e$ of the recursive function that we want to define (where f can be used for recursive calls in e), the expression $\mathsf{fix}\ (\lambda f, x.\, e)\ v$ reduces to $e[\mathsf{fix}\ (\lambda f, x.\, e)/f][v/x]$, so it replaces x by the argument and f by the recursive function. More precisely, the desired reduction behavior of fix is

$$\mathsf{fix}(\lambda f, x.\, e) v \succ^* e[\mathsf{fix}\ (\lambda f, x.\, e)/f][v/x]$$

We will define fix below. Beforehand, let us explore how we can define recursive functions such as add with fix. We define add as $\mathsf{add} := \mathsf{fix}(\lambda a, n.\, \lambda m.\, n\ m\ (\lambda n'.\, \mathsf{Succ}(a\ n'\ m)))$. With this definition, we have:

$$\begin{aligned} \mathsf{add}\ n\ m &\succ^* (\lambda m.\, n\ m\ (\lambda n'.\, \mathsf{Succ}(\mathsf{add}\ n'\ m)))\ m \\ &\succ n\ m\ (\lambda n'.\, \mathsf{Succ}(\mathsf{add}\ n'\ m)) \end{aligned}$$

as desired. In fact, we can prove that add has the desired computational behavior:

Lemma 3. $\mathsf{add}\ (\mathsf{enc}(n))\ (\mathsf{enc}(m)) \succ^* \mathsf{enc}(n+m)$

To define fix, we will abstract over the template and assume it is some abstract value F. That is, we will define the operator fix such that $\mathsf{fix}\ F\ v \succ^* F\ (\mathsf{fix}\ F)\ v$. If F is a template of the form $\lambda f, x.\, e$, then $F\ (\mathsf{fix}\ F)\ v$ reduces in two more steps to $e[\mathsf{fix}\ (\lambda f, x.\, e)/f][v/x]$.

The definition of $\mathsf{fix}\ F$ (where fix is parametric in F) consists of two parts:

$$\begin{aligned} \mathsf{fix}\ F &:= \lambda x.\, \mathsf{fix}'\ \mathsf{fix}'\ F\ x \\ \mathsf{fix}' &:= \lambda f, F.\, F\ (\lambda x.\, f\ f\ F\ x) \end{aligned}$$

The idea is that fix is defined using a term fix', which relies on self-application (like Ω) to implement recursion. So we provide fix' with itself, the template F, and the function argument x. In the definition of fix', we are given fix' as f and the template F. The argument for x is omitted to get the right reduction behavior (as we will see below). Given these arguments, we want to apply the template F to $\mathsf{fix}\ F$. What this means in the scope of fix' is that we apply F to the term $\lambda x.\, f\ f\ F\ x$.

With this definition, $\mathsf{fix}\ F$ has the right reduction behavior:

$$\begin{aligned} \mathsf{fix}\ F\ v &= (\lambda x.\, \mathsf{fix}'\ \mathsf{fix}'\ F\ x)\ v \\ &\succ \mathsf{fix}'\ \mathsf{fix}'\ F\ v \\ &\succ (\lambda F.\, F\ (\lambda x.\, \mathsf{fix}'\ \mathsf{fix}'\ F\ x))\ F\ v \\ &\succ F\ (\lambda x.\, \mathsf{fix}'\ \mathsf{fix}'\ F\ x)\ v \\ &= F\ (\mathsf{fix}\ F)\ v \end{aligned}$$

With first-order inductive data types and recursion, one can define basic arithmetic operations (*e.g.*, addition, multiplication) and build up larger programs. In fact, we have now seen all the basic building blocks that are needed to prove that the untyped lambda calculus is Turing complete (which we will not do in this course).

1.3 Typing

We now extend our language with a *type system.* We distinguish between *source terms*, containing type information, and *runtime terms*, which we have used in the previous sections.

$$\begin{array}{rrcl} \text{Types} & A, B & ::= & \mathsf{int} \mid A \to B \\ \text{Variable Contexts} & \Gamma & ::= & \emptyset \mid \Gamma, x : A \\ \text{Source Terms} & E & ::= & x \mid \lambda x : A.\, E \mid E_1\, E_2 \mid E_1 + E_2 \mid \overline{n} \end{array}$$

Church-style typing $\boxed{\Gamma \vdash E : A}$

Typing on source terms amounts to *checking* whether a source term is properly annotated.

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\,\text{VAR} \qquad \frac{\Gamma, x : A \vdash E : B}{\Gamma \vdash \lambda x : A.\, E : A \to B}\,\text{LAM} \qquad \frac{\Gamma \vdash E_1 : A \to B \qquad \Gamma \vdash E_2 : A}{\Gamma \vdash E_1\, E_2 : B}\,\text{APP}$$

$$\frac{\Gamma \vdash E_1 : \mathsf{int} \qquad \Gamma \vdash E_2 : \mathsf{int}}{\Gamma \vdash E_1 + E_2 : \mathsf{int}}\,\text{PLUS} \qquad \frac{}{\Gamma \vdash \overline{n} : \mathsf{int}}\,\text{INT}$$

Curry-style typing $\boxed{\Gamma \vdash e : A}$

Typing on runtime terms amounts to *assigning* a type to a term (if possible).

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\,\text{VAR} \qquad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.\, e : A \to B}\,\text{LAM} \qquad \frac{\Gamma \vdash e_1 : A \to B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\, e_2 : B}\,\text{APP}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{int}}\,\text{PLUS} \qquad \frac{}{\Gamma \vdash \overline{n} : \mathsf{int}}\,\text{INT}$$

Exercise 8 (Typing Uniqueness) Prove that Church typing is unique:

$$\textit{if}\ \Gamma \vdash E : A\ \textit{and}\ \Gamma \vdash E : B,\ \textit{then}\ A = B$$

Does the same hold for Curry typing? Prove it or give a counter example. •

For our language, the connection between Church-style typing and Curry-style typing can be stated easily with the help of a type erasure function. Erase takes source terms and turns them into runtime terms by erasing the type annotations in lambda abstractions.

Type Erasure $\boxed{\mathrm{Erase}(\cdot)}$

$$\begin{aligned}
\mathrm{Erase}(x) &:= x \\
\mathrm{Erase}(\lambda x : A.\, E) &:= \lambda x.\, \mathrm{Erase}(E) \\
\mathrm{Erase}(E_1\, E_2) &:= \mathrm{Erase}(E_1)\, \mathrm{Erase}(E_2) \\
\mathrm{Erase}(E_1 + E_2) &:= \mathrm{Erase}(E_1) + \mathrm{Erase}(E_2) \\
\mathrm{Erase}(\overline{n}) &:= \overline{n}
\end{aligned}$$

Lemma 4 (Erasure)**.** *If* $\vdash E : A$*, then* $\vdash \mathit{Erase}(E) : A$.

Exercise 9 Prove the Erasure lemma. •

Type Inference $\boxed{\mathsf{infer}\ \Gamma\ E}$

Source terms in the simply-typed λ-calculus contain enough typing information for us to infer their types automatically.

$$
\begin{array}{rll}
\mathsf{infer}\ \Gamma\ x = \Gamma[x] & & \\
\mathsf{infer}\ \Gamma\ \overline{n} = \mathsf{Some\ int} & & \\
\mathsf{infer}\ \Gamma\ (\lambda x : A.\, E) = \mathsf{Some}\ (A \to B) & & \textit{if}\ \mathsf{infer}\ (\Gamma, x : A)\ E = \mathsf{Some}\ B \\
\mathsf{infer}\ \Gamma\ (\lambda x : A.\, E) = \mathsf{None} & & \textit{if}\ \mathsf{infer}\ (\Gamma, x : A)\ E = \mathsf{None} \\
\mathsf{infer}\ \Gamma\ (E_1\ E_2) = \mathsf{Some}\ B & & \textit{if}\ \mathsf{infer}\ \Gamma\ E_1 = \mathsf{Some}\ (A \to B) \\
& & \textit{and}\ \mathsf{infer}\ \Gamma\ E_2 = \mathsf{Some}\ A \\
\mathsf{infer}\ \Gamma\ (E_1\ E_2) = \mathsf{None} & & \textit{otherwise} \\
\mathsf{infer}\ \Gamma\ (E_1 + E_2) = \mathsf{Some\ int} & & \textit{if}\ \mathsf{infer}\ \Gamma\ E_1 = \mathsf{Some\ int} \\
& & \textit{and}\ \mathsf{infer}\ \Gamma\ E_2 = \mathsf{Some\ int} \\
\mathsf{infer}\ \Gamma\ (E_1 + E_2) = \mathsf{None} & & \textit{otherwise}
\end{array}
$$

Exercise 10 Prove that the function infer is correct. That is, show the correspondence:

$$\mathsf{infer}\ \Gamma\ E = \mathsf{Some}\, A \quad \textit{iff} \quad \Gamma \vdash E : A$$

●

1.4 Type Safety

We now turn to the traditional property to prove usefulness of a type system: *type safety.*

Statement 5 (Type Safety)**.** *If* $\vdash e : A$ *and* $e \succ^\star e'$*, then* e' *is progressive.*

Here, we call the following terms progressive:

Definition 6 (Progressive Terms)**.** *A (runtime) term* e *is* progressive *if either it is a value or there exists* e' *s.t.* $e \rightsquigarrow e'$*.*

To prove type safety, we first have to prove a sequence of intermediate results about our type system.

Lemma 7 (Weakening for Curry-style typing)**.** *If* $\Gamma_1 \vdash e : A$ *and* $\Gamma_1 \subseteq \Gamma_2$*, then* $\Gamma_2 \vdash e : A$*.*

Proof. By induction on $\Gamma_1 \vdash e : A$ for arbitrary Γ_2. □

Lemma 8 (Preservation of Typing under Substitution)**.**
If $\Gamma, x : A \vdash e : B$ *and* $\vdash e' : A$*, then* $\Gamma \vdash e[e'/x] : B$*.*

Proof. By induction on e for arbitrary B and Γ. □

Lemma 9 (Canonical forms)**.** *If* $\vdash v : A$*, then:*

- *if* $A = \mathsf{int}$*, then* $v = \overline{n}$ *for some* n
- *if* $A = A_1 \to A_2$ *for some* A_1*,* A_2*, then* $v = \lambda x.\, e$ *for some* x*,* e

Proof. By inversion. □

Theorem 10 (Progress)**.** *If* $\vdash e : A$*, then* e *is progressive.*

Proof Sketch. By induction on $\vdash e : A$. We discuss the case of application. Let $\vdash e_1 : B \to A$ and $\vdash e_2 : B$. By induction e_1 and e_2 are progressive. We distinguish three cases:

a) Let e_1 and e_2 be values. Then by Lemma 9 $e_1 = \lambda x.\, e$ for some x and e. Thus, by BETA we have $e_1 e_2 \rightsquigarrow e[e_2/x]$.

b) Let $e_1 \rightsquigarrow e_1'$ and e_2 be a value. Then $e_1\, e_2 \rightsquigarrow e_1'\, e_2$ by Lemma 1 with $K := (\bullet\, e_2)$.

c) Let $e_2 \rightsquigarrow e_2'$. Then $e_1\, e_2 \rightsquigarrow e_1\, e_2'$ by Lemma 1 with $K := (e_1\, \bullet)$.

□

To complete the proof of type safety, we still need one important property of our type system: preservation (see Theorem 14). For the proof of preservation, we define the notion of *contextual typing* $\vdash K : A \Rightarrow B$, which expresses that a context is of type B if filled with an expression of type A. As for many other mathematical concepts, there are two ways to define contextual typing: For years, Derek used an *intensional definition* of contextual typing, meaning a definition with inductive rules similar to typing rules from which valid context typings can be derived. In the following, we are going to use the more concise *extensional definition* of Jules Jacobs, where we take the property that we want from contextual typing as the definition.

Contextual typing $\boxed{\vdash K : A \Rightarrow B}$

$$\vdash K : A \Rightarrow B \quad := \quad \forall e.\, \vdash e : A \Rightarrow \vdash K[e] : B$$

Lemma 11 (Decomposition)**.** *If $\vdash K[e] : A$, then there exists B s.t.*

$$\vdash K : B \Rightarrow A \text{ and } \vdash e : B.$$

Proof. By induction on K. □

Lemma 12 (Composition)**.** *If $\vdash K : B \Rightarrow A$ and $\vdash e : B$, then $\vdash K[e] : A$.*

Proof. By definition. This would be an induction for the intensional definition of contextual typing. □

Lemma 13 (Base preservation)**.** *If $\vdash e : A$ and $e \rightsquigarrow_b e'$, then $\vdash e' : A$.*

Proof. By cases on $e \rightsquigarrow_b e'$:

Case 1: BETA, $e = (\lambda x.\, e_1)\, v$ and $e' = e_1[v/x]$. It remains to show that $\vdash e_1[v/x] : A$. By inversion on $\vdash e : A$ we have $\vdash \lambda x.\, e_1 : A_0 \to A$ and $\vdash v : A_0$ for some A_0. By inversion on $\vdash \lambda x.\, e_1 : A_0 \to A$ we have $x : A_0 \vdash e_1 : A$. By Lemma 8, we have $\vdash e_1[v/x] : A$.

Case 2: PLUS, $e = \overline{n} + \overline{m}$ and $e' = \overline{n + m}$. By inversion on $\vdash e : A$, we have $A = \mathsf{int}$. We establish $\vdash \overline{n + m} : \mathsf{int}$ by INT. □

Theorem 14 (Preservation)**.** *If $\vdash e : A$ and $e \rightsquigarrow e'$, then $\vdash e' : A$.*

Proof. Invert $e \rightsquigarrow e'$ to obtain K, e_1, e_1' s.t. $e = K[e_1]$ and $e' = K[e_1']$ and $e_1 \rightsquigarrow_b e_1'$.
By Lemma 11, there exists B s.t. $\vdash K : B \Rightarrow A$ and $\vdash e_1 : B$.
By Lemma 13, from $\vdash e_1 : B$ and $e_1 \rightsquigarrow_b e_1'$, we have $\vdash e_1' : B$.
By Lemma 12, from $\vdash e_1' : B$ and $\vdash K : B \Rightarrow A$, we have $\vdash K[e_1'] : A$, so we are done. □

Corollary 15 (Type Safety)**.** *If $\vdash e : A$ and $e \rightsquigarrow^\star e'$, then e' is progressive.*

Exercise 11 We call an expression e *safe* if for any expression e' s.t. $e \rightsquigarrow^\star e'$, e' is progressive. If e is closed and well-typed, by Type Safety we know that e must be safe. Is there a closed expression that is safe, but *not* well-typed? In other words, is there a closed expression e that is safe but there is no type A s.t. $\vdash e : A$? Give one example if there is such an expression, otherwise prove their non-existence. ●

Exercise 12 (Products and Sums) From logic, you know the connectives conjunction ($\wedge$) and disjunction ($\vee$). The corresponding constructs in programming are *products* and *sums*. In the following, we will extend our lambda calculus with support for products and sums. Let us start by extending the syntax of the language and the type system:

$$\begin{array}{rrcl}
\text{Types} & A, B & ::= & \ldots \mid A \times B \mid A + B \\
\text{Source Terms} & E & ::= & \ldots \mid \langle E_1, E_2 \rangle \mid \pi_1\, E \mid \pi_2\, E \\
 & & & \mid \mathsf{inj}_1^{A+B}\, E \mid \mathsf{inj}_2^{A+B}\, E \\
 & & & \mid (\mathsf{case}\ E_0\ \mathsf{of}\ E_1 \mid E_2\ \mathsf{end}) \\
\text{Runtime Terms} & e & ::= & \ldots \mid \langle e_1, e_2 \rangle \mid \pi_1\, e \mid \pi_2\, e \\
 & & & \mid \mathsf{inj}_1\, e \mid \mathsf{inj}_2\, e \mid (\mathsf{case}\ e_0\ \mathsf{of}\ e_1 \mid e_2\ \mathsf{end}) \\
\text{(Runtime) Values} & v & ::= & \ldots \mid \langle v_1, v_2 \rangle \mid \mathsf{inj}_1\, v \mid \mathsf{inj}_2\, v
\end{array}$$

The structural operational semantics of products and sums is given by:

$$\ldots \quad \overset{\text{PROD-STRUCT-L}}{\dfrac{e_1 \succ e_1'}{\langle e_1, v_2 \rangle \succ \langle e_1', v_2 \rangle}} \quad \overset{\text{PROD-STRUCT-R}}{\dfrac{e_2 \succ e_2'}{\langle e_1, e_2 \rangle \succ \langle e_1, e_2' \rangle}} \quad \overset{\text{PROJ-STRUCT}}{\dfrac{e \succ e'}{\pi_i\, e \succ \pi_i\, e'}} \quad \overset{\text{PROJ}}{\pi_i\, \langle v_1, v_2 \rangle \succ v_i}$$

$$\overset{\text{INJ-STRUCT}}{\dfrac{e \succ e'}{\mathsf{inj}_i\, e \succ \mathsf{inj}_i\, e'}} \quad \overset{\text{CASE-STRUCT}}{\dfrac{e_0 \succ e_0'}{\mathsf{case}\ e_0\ \mathsf{of}\ e_1 \mid e_2\ \mathsf{end} \succ \mathsf{case}\ e_0'\ \mathsf{of}\ e_1 \mid e_2\ \mathsf{end}}}$$

$$\overset{\text{CASE-INJ}}{\mathsf{case}\ \mathsf{inj}_i\, v\ \mathsf{of}\ e_1 \mid e_2\ \mathsf{end} \succ e_i\ v}$$

and their typing rules are given by:

$$\ldots \quad \overset{\text{PROD}}{\dfrac{\Gamma \vdash E_1 : A \qquad \Gamma \vdash E_2 : B}{\Gamma \vdash \langle E_1, E_2 \rangle : A \times B}} \quad \overset{\text{PROJ}}{\dfrac{\Gamma \vdash E : A_1 \times A_2}{\Gamma \vdash \pi_i\, E : A_i}} \quad \overset{\text{INJ}}{\dfrac{\Gamma \vdash E : A_i}{\Gamma \vdash \mathsf{inj}_i^{A_1+A_2}\, E : A_1 + A_2}}$$

$$\overset{\text{CASE}}{\dfrac{\Gamma \vdash E_0 : B + C \qquad \Gamma \vdash E_1 : B \to A \qquad \Gamma \vdash E_2 : C \to A}{\Gamma \vdash \mathsf{case}\ E_0\ \mathsf{of}\ E_1 \mid E_2\ \mathsf{end} : A}}$$

We can define a bit of syntactic sugar for our convenience:

$$\mathsf{match}\ e_0\ \mathsf{of}\ \mathsf{inj}_1\, x_1.\, e_1 \mid \mathsf{inj}_2\, x_2.\, e_2\ \mathsf{end} := \mathsf{case}\ e_0\ \mathsf{of}\ (\lambda x_1.e_1) \mid (\lambda x_2.e_2)\ \mathsf{end}$$

In this exercise, you will extend the *contextual* operational semantics and the type safety proof for products and sums.

a) Extend the typing rules for runtime terms (Curry-style).

b) Extend the statement of the canonical forms lemma.

c) Extend the definition of evaluation contexts K.

d) Extend the definition of context filling $K[e]$.

e) Extend the definition of the base reduction $\leadsto_{\mathrm{b}}$.

f) Extend the definition of the big-step semantics.

g) Extend the proof of progress for the new cases.

h) Extend the proof of composition and decomposition for the new cases.

i) Extend the proof of base preservation. Note how we only need to extend the proof of base preservation: the proof of preservation itself does not change.

Note that the proofs for the old cases do not change. ●

Exercise 13 (Binary operators) Our simple calculus has only a single operator: binary addition. In this exercise, we are going to add additional binary operators: multiplication $\times$ as well as subtraction $-$.

It turns out that the operational semantics and typing rules for binary operators follow a very similar pattern. Thus, it will be useful to setup a bit of shared machinery. We define the binary operators as follows and adapt the expressions of our language with a common expression for all binary operators:

$$o : O ::= + \mid \times \mid -$$
$$\text{Runtime Terms } e ::= \dots \mid \overline{n} \mid e_1\ o\ e_2$$

We can define a common rule for the contextual semantics, where we have a function for evaluating the binary operators (returning an option, in case the operator is not applicable to the two operands), where we leave it to you to define $\mathsf{bin\_eval}$.

$$\textsc{binop}\ \frac{\mathsf{bin\_eval}\ v_1\ v_2 = \mathsf{Some}(v)}{v_1\ o\ v_2 \leadsto_{\mathrm{b}} v}$$

For extending the typing rules and the proofs of progress and preservation, it will similarly pay off to set up some shared structure. We define a separate typing judgment for binary operators, making it general enough to extend it with non-integer binary operators in the future (*e.g.*, for binary operators on Booleans):

$$\textsc{plus}\ \ \mathsf{int}\,;\mathsf{int} \vdash_{\mathsf{binop}} + : \mathsf{int} \qquad \textsc{mul}\ \ \mathsf{int}\,;\mathsf{int} \vdash_{\mathsf{binop}} \times : \mathsf{int} \qquad \dots$$

Then, we need just a single typing rule for binary operators:

$$\textsc{binop}\ \frac{\Gamma \vdash E_1 : A_1 \qquad \Gamma \vdash E_2 : A_2 \qquad A_1\,; A_2 \vdash_{\mathsf{binop}} o : A}{\Gamma \vdash E_1 o E_2 : A}$$

It is your task to fill out the remaining details by following the steps outlined above. Concretely, perform the following steps for each of the exercises:

a) Extend the definition of the base reduction $\leadsto_{\mathrm{b}}$. For that, complete the definition of $\mathsf{bin\_eval}$.

b) Extend the definition of evaluation contexts K.

c) Extend the definition of context filling $K[e]$ and context composition $K_1 \circ K_2$.

d) Extend the typing rules for runtime terms (Curry-style) and define the new typing judgment $\vdash_{\mathsf{binop}}$.

e) Extend the statement of the canonical forms lemma.

f) Extend the statement of the substitutivity lemma (preservation of typing under substitution).

g) Extend the proof of progress for the new cases.

h) Extend the proof of base preservation. Note how we only need to extend the proof of base preservation: the proof of preservation itself does not change.

i) Extend the definition of the big-step semantics.

You will note that the proofs for the existing constructs of the language do not change.

●

1.5 Termination

In this section, we want to prove that every well-typed term eventually reduces to a value.

Statement 16 (Termination). *If $\vdash e : A$, then there exists v s.t. $e \leadsto^* v$.*

We define the notion of "semantically good" expressions and values.

Value Relation $\boxed{\mathcal{V}[\![A]\!]}$

$$\begin{aligned} \mathcal{V}[\![\mathsf{int}]\!] &:= \{\overline{n} \mid n \in \mathbb{Z}\} \\ \mathcal{V}[\![A \to B]\!] &:= \{\lambda x.\, e \mid \forall v.\, v \in \mathcal{V}[\![A]\!] \Rightarrow e[v/x] \in \mathcal{E}[\![B]\!]\} \end{aligned}$$

Expression Relation $\boxed{\mathcal{E}[\![A]\!]}$

$$\mathcal{E}[\![A]\!] := \{e \mid \exists v.\, e \downarrow v \wedge v \in \mathcal{V}[\![A]\!]\}$$

Context Relation $\boxed{\mathcal{G}[\![\Gamma]\!]}$

$$\emptyset \in \mathcal{G}[\![\emptyset]\!] \qquad\qquad \frac{\gamma \in \mathcal{G}[\![\Gamma]\!] \qquad v \in \mathcal{V}[\![A]\!]}{\gamma[x \mapsto v] \in \mathcal{G}[\![\Gamma, x : A]\!]}$$

Above, we have discussed substitution of a single variable. For the definition of semantic typing, we need *parallel substitution*: the action of substituting multiple free variables by values from a map γ.

Substitution $\boxed{\gamma(e)}$

$$
\begin{aligned}
\gamma(x) &:= \begin{cases} v & \text{if } \gamma(x) = v \\ x & \text{ow.} \end{cases} \\
\gamma(\overline{n}) &:= \overline{n} \\
\gamma(\lambda x.\, e) &:= \lambda x.\, (\gamma[x \mapsto \bot])\, e \\
\gamma(e_1\, e_2) &:= \gamma(e_1)\, \gamma(e_2) \\
\gamma(e_1 + e_2) &:= \gamma(e_1) + \gamma(e_2)
\end{aligned}
$$

We can now define a *semantic typing judgment*.

Semantic Typing $\boxed{\Gamma \vDash e : A}$

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

Lemma 17 (Value Inclusion)**.** *If $e \in \mathcal{V}[\![A]\!]$, then $e \in \mathcal{E}[\![A]\!]$.*

Theorem 18 (Semantic Soundness)**.** *If $\Gamma \vdash e : A$, then $\Gamma \vDash e : A$.*

Proof. By induction on $\Gamma \vdash e : A$, and then using the *compatibility* lemmas of the semantic typing (see Lemma 19, Lemma 20, and Lemma 21). The compatibility lemmas state that we can derive the rules of syntactic typing for semantic typing. As such, the semantic typing is *compatible* with the syntactic typing. In these semantic soundness proofs, for each syntactic rule, the inductive hypotheses give us the semantic premises of the rule. We then only need to apply the corresponding compatibility lemma to close the case.

Compatibility lemmas for VAR, LAM, and APP are Lemma 19, Lemma 20, and Lemma 21, respectively. We omit the compatibility lemmas for INT and PLUS. □

Lemma 19 (Compatibility with VAR)**.** *If $x : A \in \Gamma$ then $\Gamma \vDash x : A$.*

Proof.

We have:	To show:
$x : A \in \Gamma$	$\Gamma \vDash x : A$
Suppose $\gamma \in \mathcal{G}[\![\Gamma]\!]$	$\gamma(x) \in \mathcal{E}[\![A]\!]$
$\gamma(x) \in \mathcal{V}[\![A]\!]$	
We conclude by value inclusion (Lemma 17).	

□

Lemma 20 (Compatibility with LAM)**.** *If $\Gamma, x : A \vDash e : B$ then $\Gamma \vDash \lambda x.\, e : A \to B$.*

Proof.

We have:	To show:
$\Gamma, x : A \vDash e : B$	$\Gamma \vDash \lambda x.\, e : A \to B$
Suppose $\gamma \in \mathcal{G}[\![\Gamma]\!]$	$\gamma(\lambda x.\, e) \in \mathcal{E}[\![A \to B]\!]$
	$\lambda x.\, (\gamma[x \mapsto \bot])(e) \in \mathcal{E}[\![A \to B]\!]$
	By Lemma 17, to show $\lambda x.\, (\gamma[x \mapsto \bot])(e) \in \mathcal{V}[\![A \to B]\!]$
Suppose $v \in \mathcal{V}[\![A]\!]$	$((\gamma[x \mapsto \bot])e)[v/x] \in \mathcal{E}[\![B]\!]$
Let $\gamma' := \gamma[x \mapsto v]$, so $\gamma'(e) = \gamma[x \mapsto v](e)$	
$= \gamma[x \mapsto \bot][x \mapsto v](e)$	
$= (\gamma[x \mapsto \bot](e))[v/x]$	
	$\gamma'(e) \in \mathcal{E}[\![B]\!]$
From $\gamma \in \mathcal{G}[\![\Gamma]\!]$ and $v \in \mathcal{V}[\![A]\!]$, have $\gamma' \in \mathcal{G}[\![\Gamma, x : A]\!]$	
We are done by using the assumption $\Gamma, x : A \vDash e : B$.	

□

Lemma 21 (Compatibility with APP)**.** *If* $\Gamma \vDash e_1 : A \to B$ *and* $\Gamma \vDash e_2 : A$ *then* $\Gamma \vDash e_1\, e_2 : B$.

Proof.

We have:	To show:
$\Gamma \vDash e_1 : A \to B$	
$\Gamma \vDash e_2 : A$	$\Gamma \vDash e_1\, e_2 : B$
Suppose $\gamma \in \mathcal{G}[\![\Gamma]\!]$	$\gamma(e_1\, e_2) \in \mathcal{E}[\![B]\!]$
	$\gamma(e_1)\, \gamma(e_2) \in \mathcal{E}[\![B]\!]$
From assumptions,	
there exist x, e', v_2 s.t. $\gamma(e_1) \downarrow \lambda x.\, e \in \mathcal{V}[\![A \to B]\!]$ and $\gamma(e_2) \downarrow v_2 \in \mathcal{V}[\![A]\!]$.	
So $e[v_2/x] \in \mathcal{E}[\![B]\!]$	
$\exists v.\, e[v_2/x] \downarrow v \in \mathcal{V}[\![B]\!]$	
By APP, $\gamma(e_1)\, \gamma(e_2) \downarrow v \in \mathcal{V}[\![B]\!]$, so we are done	

□

Exercise 14 In the value relation, we define the set of "good" function values as:

$$\mathcal{V}[\![A \to B]\!] := \{\lambda x.\, e \mid \forall v.\, v \in \mathcal{V}[\![A]\!] \Rightarrow e[v/x] \in \mathcal{E}[\![B]\!]\}$$

If we instead define the set as:

$$\mathcal{V}[\![A \to B]\!] := \{v \mid \forall v'.\, v' \in \mathcal{V}[\![A]\!] \Rightarrow v\, v' \in \mathcal{E}[\![B]\!]\}$$

does the proof of semantic soundness still go through? ●

Corollary 22 (Termination)**.** *If* $\emptyset \vdash e : A$*, then there exists* v *s.t.* $e \downarrow v$.

Proof. By Theorem 18, we have $\emptyset \vDash e : A$. Pick γ to be the identity, which clearly is in $\mathcal{G}[\![\emptyset]\!]$. Hence $e \in \mathcal{E}[\![A]\!]$. By definition then, $\exists v.\, e \downarrow v$. □

Exercise 15 Extend the termination proof, which requires extending the semantic soundness proof, to cover products and sums.

●

Exercise 16 (Call-by-name Termination) Recall the call-by-name small-step semantics $e \succ_{\mathsf{cbn}} e'$. The corresponding big-step semantics is defined as:

$$\overline{n} \downarrow_{\mathsf{cbn}} \overline{n} \qquad \lambda x.\, e \downarrow_{\mathsf{cbn}} \lambda x.\, e \qquad \frac{e_1 \downarrow_{\mathsf{cbn}} \overline{n} \qquad e_2 \downarrow_{\mathsf{cbn}} \overline{m}}{e_1 + e_2 \downarrow_{\mathsf{cbn}} \overline{n+m}} \qquad \frac{e_1 \downarrow_{\mathsf{cbn}} \lambda x.\, e \qquad e[e_2/x] \downarrow_{\mathsf{cbn}} v}{e_1\ e_2 \downarrow_{\mathsf{cbn}} v}$$

To prove that expressions terminate under call-by-name evaluation, we can setup a similar logical relation to the one for call-by-value evaluation:

$$\begin{aligned}
\mathcal{V}[\![\mathsf{int}]\!] &:= \{\overline{n} \mid n \in \mathbb{Z}\} \\
\mathcal{V}[\![A \to B]\!] &:= \left\{\lambda x.\, e \,\middle|\, \forall e'.\, e' \in \mathcal{E}[\![A]\!] \Rightarrow e[e'/x] \in \mathcal{E}[\![B]\!]\right\} \\
\mathcal{E}[\![A]\!] &:= \{e \mid \exists v.\, e \downarrow_{\mathsf{cbn}} v \wedge v \in \mathcal{V}[\![A]\!]\} \\
\mathcal{G}[\![\emptyset]\!] &:= \{\gamma \mid \gamma = \emptyset\} \\
\mathcal{G}[\![\Gamma, x : A]\!] &:= \{\gamma[x \mapsto e] \mid \gamma \in \mathcal{G}[\![\Gamma]\!] \wedge e \in \mathcal{E}[\![A]\!]\} \\
\Gamma \vDash e : A &:= \forall \gamma \in \mathcal{G}[\![\Gamma]\!].\, \gamma(e) \in \mathcal{E}[\![A]\!]
\end{aligned}$$

Prove that all well-typed, closed terms terminate under call-by-name evaluation. That is, show that if $\vdash e : A$, then $e \downarrow_{\mathsf{cbn}} v$ for some v.

●

2 System F: Polymorphism and Existential Types

We extend the STLC with polymorphism and existential types. Polymorphism is widespread in modern (functional) programming languages, while existential types serve as a way of creating data abstraction, with a rough correspondence to modules in ML-like languages.

2.1 System F

$$\begin{array}{rrcl}
\text{Types} & A, B & ::= & \dots \mid \forall \alpha.\, A \mid \exists \alpha.\, A \mid \alpha \\
\text{Type Variable Contexts} & \Delta & ::= & \emptyset \mid \Delta, \alpha \\
\text{Source Terms} & E & ::= & \dots \mid \Lambda \alpha.\, E \mid E\,\langle A \rangle \mid \mathsf{pack}\,[A, E]\,\mathsf{as}\,\exists \alpha.\, B \\
 & & & \mid \mathsf{unpack}\,E\,\mathsf{as}\,[\alpha, x]\,\mathsf{in}\,E' \\
\text{Runtime Terms} & e & ::= & \dots \mid \Lambda.\, e \mid e\,\langle\rangle \mid \mathsf{pack}\,e \mid \mathsf{unpack}\,e\,\mathsf{as}\,x\,\mathsf{in}\,e' \\
\text{(Runtime) Values} & v & ::= & \dots \mid \Lambda.\, e \mid \mathsf{pack}\,v \\
\text{Evaluation Contexts} & K & ::= & \dots \mid K\,\langle\rangle \mid \mathsf{pack}\,K \mid \mathsf{unpack}\,K\,\mathsf{as}\,x\,\mathsf{in}\,e
\end{array}$$

Contextual operational semantics $\boxed{e_1 \leadsto_{\mathrm{b}} e_2}$

$$\textsc{bigBeta}\quad (\Lambda.\, e)\,\langle\rangle \leadsto_{\mathrm{b}} e \qquad\qquad \textsc{unpack}\quad \mathsf{unpack}\,(\mathsf{pack}\,v)\,\mathsf{as}\,x\,\mathsf{in}\,e \leadsto_{\mathrm{b}} e[v/x]$$

Because we now deal with type variables, we have to deal with a new kind of contexts: *type variable contexts.* The typing judgments will now carry both a type variable context and "normal" variable context. All the existing typing rules remain valid, with the type variable context being the same in all premises and the conclusion of the typing rules. However, for the typing rule for lambdas, we have to make sure that the argument type is actually well-formed in the current typing context.

Type Well-Formedness $\boxed{\Delta \vdash A}$

$$\frac{\mathrm{FV}(A) \subseteq \Delta}{\Delta \vdash A}$$

Church-style typing $\boxed{\Delta\,;\Gamma \vdash E : A}$

$$\dots \qquad \textsc{lam}\ \frac{\Delta \vdash A \qquad \Delta\,;\Gamma, x : A \vdash E : B}{\Delta\,;\Gamma \vdash \lambda x : A.\, E : A \to B} \qquad \textsc{bigLam}\ \frac{\Delta, \alpha\,;\Gamma \vdash E : A}{\Delta\,;\Gamma \vdash \Lambda \alpha.\, E : \forall \alpha.\, A}$$

$$\textsc{bigApp}\ \frac{\Delta, \Gamma \vdash E : \forall \alpha.\, B \qquad \Delta \vdash A}{\Delta\,;\Gamma \vdash E\,\langle A \rangle : B[A/\alpha]} \qquad \textsc{pack}\ \frac{\Delta \vdash A \qquad \Delta\,;\Gamma \vdash E : B[A/\alpha] \qquad (\Delta \vdash \exists \alpha.\, B)}{\Delta\,;\Gamma \vdash \mathsf{pack}\,[A, E]\,\mathsf{as}\,\exists \alpha.\, B : \exists \alpha.\, B}$$

$$\textsc{unpack}\ \frac{\Delta\,;\Gamma \vdash E : \exists \alpha.\, B \qquad \Delta, \alpha\,;\Gamma, x : B \vdash E' : C \qquad \Delta \vdash C}{\Delta\,;\Gamma \vdash \mathsf{unpack}\,E\,\mathsf{as}\,[\alpha, x]\,\mathsf{in}\,E' : C}$$

Curry-style typing $\boxed{\Delta\,;\Gamma \vdash e : A}$

$$\ldots \qquad \frac{\Delta \vdash A \qquad \Delta\,;\Gamma, x : A \vdash e : B}{\Delta\,;\Gamma \vdash \lambda x.\, e : A \to B}\;\textsc{Lam} \qquad \frac{\Delta\,;\Gamma \vdash e_1 : A \to B \qquad \Delta\,;\Gamma \vdash e_2 : A}{\Delta\,;\Gamma \vdash e_1\, e_2 : B}\;\textsc{App}$$

$$\frac{\Delta, \alpha\,;\Gamma \vdash e : A}{\Delta\,;\Gamma \vdash \Lambda.\, e : \forall \alpha.\, A}\;\textsc{bigLam} \qquad \frac{\Delta, \Gamma \vdash e : \forall \alpha.\, B \qquad \Delta \vdash A}{\Delta\,;\Gamma \vdash e\,\langle\rangle : B[A/\alpha]}\;\textsc{bigApp}$$

$$\frac{\Delta \vdash A \qquad \Delta\,;\Gamma \vdash e : B[A/\alpha]}{\Delta\,;\Gamma \vdash \mathsf{pack}\, e : \exists \alpha.\, B}\;\textsc{Pack}$$

$$\frac{\Delta\,;\Gamma \vdash e : \exists \alpha.\, B \qquad \Delta, \alpha\,;\Gamma, x : B \vdash e' : C \qquad \Delta \vdash C}{\Delta\,;\Gamma \vdash \mathsf{unpack}\, e\, \mathsf{as}\, x\, \mathsf{in}\, e' : C}\;\textsc{Unpack}$$

The problem with named binders In the above definition of System F, we have used strings (*e.g.*, $\alpha, \beta, \gamma, \ldots$) for the type variables. While this approach is very intuitive for humans, it causes trouble when we attempt to mechanize System F in Coq. The issue is once again *variable capturing* (see Section 1). The substitution on types $A[B/\alpha]$ has the same problems as the substitution on terms $e[e'/x]$: if we define $A[B/\alpha]$ naively, then it incurs variable capturing if the type we insert contains free variables (*e.g.*, α in $\alpha \to \alpha$).

For the term substitution $e[e'/x]$ variable capturing was not a problem, because we usually insert *closed* expressions e'. As it turns out, for our type substitution variable capturing will be a problem. To see why, consider the following example:

$$E_{\mathsf{capt}} := \Lambda\beta.\, (\Lambda\alpha.\, \Lambda\beta.\, \lambda x : \alpha \to \beta.\, x)\langle\beta\rangle$$

Intuitively, E_{capt} should have type $\forall \beta, \beta'.\, (\beta \to \beta') \to (\beta \to \beta')$, since the inner $(\Lambda\alpha.\, \cdots)\,\langle\beta\rangle$ "cancel each other out". Unfortunately, if we attempt to type check E_{capt}, it is not as simple.

$$\cfrac{\cfrac{\cfrac{\vdots}{\beta\,;\emptyset \vdash \Lambda\alpha.\, \Lambda\beta.\, \lambda x : \alpha \to \beta.\, x : \forall \alpha, \beta.\, (\alpha \to \beta) \to (\alpha \to \beta)}}{\beta\,;\emptyset \vdash (\Lambda\alpha.\, \Lambda\beta.\, \lambda x : \alpha \to \beta.\, x)\langle\beta\rangle : A_{\mathsf{subst}}} \qquad A_{\mathsf{subst}} = (\forall \beta.\, (\alpha \to \beta) \to (\alpha \to \beta))[\beta/\alpha]}{\emptyset\,;\emptyset \vdash E_{\mathsf{capt}} : \forall \beta.\, A_{\mathsf{subst}}}$$

In this derivation, A_{subst} will be the result of $(\forall \beta.\, (\alpha \to \beta) \to (\alpha \to \beta))[\beta/\alpha]$. A capture avoiding substitution would replace the bound variable β with another variable (e.g., β'), resulting in $A_{\mathsf{subst}} = \forall \beta'.\, (\beta \to \beta') \to (\beta \to \beta')$. In contrast, a capture incurring substitution such as the one we defined on terms in Section 1 will not do any renaming. Thus, it would result in $A_{\mathsf{subst}} = \forall \beta.\, (\beta \to \beta) \to (\beta \to \beta)$.

While capture-avoiding substitution is very easy for humans, it is extremely tedious to mechanize. When reasoning formally, in Coq, we will thus avoid this tedium, by switching to a variable representation that is easier to mechanize but arguably harder to read for humans: *De Bruijn indices*.

2.2 De Bruijn representation

In *De Bruijn representation*, variables are not strings that are bound at a *binder* (*e.g.*, λx, $\Lambda\alpha$, $\exists\alpha$, and $\forall\alpha$). Instead, variables are natural numbers which indicate to which binder they refer. More precisely, they explain how many binders we need to "skip" (going up the syntax tree) until we reach the binder for the variable. This abstract idea is best understood with a few examples:

- The type $\forall\alpha.\, \alpha \to \alpha$ is represented as $\forall.\, \widehat{0} \to \widehat{0}$. When looking for the binding occurrence of the variable α, we have to skip 0 binders, as the single type quantifier binds it.
- The type $\forall\alpha, \beta.\, \alpha \to \beta$ is represented as $\forall.\, \forall.\, \widehat{1} \to \widehat{0}$. When looking for the binding occurrence of α, we have to skip the quantifier introducing β.
- The type $\forall\alpha.\, (\exists\beta.\, \alpha \to \beta) \to \alpha$ is represented as $\forall.\, (\exists.\, \widehat{1} \to \widehat{0}) \to \widehat{0}$. Note that the term structure is important: the first occurrence of α is replaced by $\widehat{1}$, as it sits below an additional quantifier, while the second occurrence is not below further binders.

In De Bruijn representation, our types are:

$$A, B ::= \ \dots \ \mid \forall.\, A \mid \exists.\, A \mid \widehat{n}$$

Exercise 17 (De Bruijn) In this exercise, you will get some practice with translating between the different representations.

(a) Translate the following types using named binders to their De Bruijn representation.
  - $\forall\alpha.\, \alpha$
  - $\forall\alpha.\, \alpha \to \alpha$
  - $\forall\alpha, \beta.\, \alpha \to (\beta \to \alpha) \to \alpha$
  - $\forall\alpha.\, (\forall\beta.\, \beta \to \alpha) \to (\forall\beta, \delta.\, \beta \to \delta \to \alpha)$
  - $\forall\alpha, \beta.\, (\beta \to (\forall\alpha.\, \alpha \to \beta)) \to \alpha$

(b) Translate the following types in De Bruijn representation to a named representation. You may choose names arbitrarily, but without conflicts, so ensure that translating the resulting types back to De Bruijn representation would produce the original type again.
  - $\forall.\, \forall.\, \widehat{0}$
  - $\forall.\, (\forall.\, \widehat{1} \to \widehat{0})$
  - $\forall.\, \forall.\, (\forall.\, \widehat{1} \to \widehat{0})$
  - $\forall.\, (\forall.\, \widehat{0} \to \widehat{1}) \to \forall.\, \widehat{0} \to \widehat{1} \to \widehat{0}$

●

Exercise 18 (Named to De Bruijn) Write a function, which translates types in named representation to types in De Bruijn representation. Test your function on the examples from Exercise 17.
Hint: Define a function debruijn m A where m is a map keeping track of which De Bruijn indices the free variables in A point to. You will need to update this map as you move underneath binders. ●

Exercise 19 (DeBruijn to Named) If we attempt to define a function which maps types in De Bruijn representation to their named counterparts, we are faced with a decision. There are multiple different named types which map to the same De Bruijn representation. Give two named types that have the same De Bruijn representation. •

Substitution Let us now turn to substitution on types in De Bruijn representation. The substitution operation that we want for System F, denoted $A[\sigma]$, replaces the free variables in A with the types given by the *parallel* substitution σ. (We will derive a unary substitution $A[B/\ ]$ corresponding to $A[B/\alpha]$ from $A[\sigma]$ subsequently.) Defining $A[\sigma]$ can be done mechanically (*i.e.*, its definition follows a concrete recipe), but it can get a bit hairy. Thus, we use a tool in Coq, called Autosubst [9], which defines $A[\sigma]$ for us and provides us with tactics to reason about it. To understand *in principle* what is going on and how De Bruijn types work, we sketch how one can define the capture avoiding, parallel substitution $A[\sigma]$.

As a stepping stone, we define a broken, *capture incurring* substitution operation $A[\sigma]$:

$$\begin{aligned} \widehat{n}[\sigma] &:= \sigma(n) \\ \mathsf{int}[\sigma] &:= \mathsf{int} \\ (A \to B)[\sigma] &:= A[\sigma] \to B[\sigma] \\ (\forall.\, A)[\sigma] &:= \forall.\, A[\Uparrow \sigma] \\ (\exists.\, A)[\sigma] &:= \exists.\, A[\Uparrow \sigma] \end{aligned} \qquad \begin{aligned} \Uparrow \sigma &:= \widehat{0}.\sigma \\ (A.\sigma)\ 0 &:= A \\ (A.\sigma)\ (n+1) &:= \sigma(n) \end{aligned}$$

For variables, we simply look up in the map σ which type should be inserted for n. The interesting cases are binders. For binders such as $\forall.\, A$, we must change the substitution: we want the variable $\widehat{0}$ in A to remain unchanged, since it is bound to the quantifier $\forall$. For all other variables, we need to shift the substitution: $\widehat{0}$ outside of the binder $\forall$ is $\widehat{1}$ inside of the binder, so we need to replace $\widehat{1}$ with $\sigma(0)$ and not with $\sigma(1)$. That is what the lifting operation $\Uparrow \sigma$ does. $\Uparrow \sigma$ uses the so-called *cons* $A.\sigma$ to map $\widehat{0}$ to itself and shift the substitution σ. To see that this makes sense, consider the following example: $(\forall.\, \widehat{0} \to \widehat{1} \to \widehat{2})[\mathsf{bool}.\mathsf{int}.\mathsf{unit}.\ldots]$. If we compute the substitution, then we obtain the desired result $\forall.\, \widehat{0} \to \mathsf{bool} \to \mathsf{int}$ (and *not* $\forall.\, \mathsf{bool} \to \mathsf{int} \to \mathsf{unit}$ or $\forall.\, \widehat{0} \to \mathsf{int} \to \mathsf{unit}$).

As teased above, this substitution still has one fundamental flaw: it is capture incurring. For example, if we apply the identity substitution $\mathsf{id}(n) := \widehat{n}$ to $\forall.\, \widehat{0} \to \widehat{1}$, then one would intuitively expect the result $\forall.\, \widehat{0} \to \widehat{1}$. However, if we compute $(\forall.\, \widehat{0} \to \widehat{1})[\mathsf{id}]$ for the broken substitution above, we obtain $\forall.\, \widehat{0} \to \widehat{0}$, which is clearly wrong! The problem is that when we move underneath a binder (here $\forall$), we do not account for the free variables in the types in σ. For example, if $\sigma = \mathsf{id}$ inserts $\widehat{0}$ for 0 *outside of* the binder, then it needs to insert $\widehat{1}$ for 1 *underneath* the binder.

Thus, to make $A[\sigma]$ capture avoiding, we *rename* all the variables in the substitution σ as soon as we move underneath a binder: if they previously referred to n, then they should refer to $n+1$ underneath the binder. In the definition of $A[\sigma]$, this change is reflected by changing $\Uparrow \sigma$:

$$\Uparrow \sigma := \widehat{0}.(\mathsf{ren}\ \mathsf{S}\ \sigma) \quad \text{where} \quad (\mathsf{ren}\ \delta\ \sigma)(n) := \mathsf{ren}\ \delta\ (\sigma\ n) \text{ and } \mathsf{S}\ n := n+1$$

The renaming operation on types (with a map δ from variables to variables) is defined analogously to substitution by:

$$\begin{aligned}
\mathsf{ren}\ \delta\ \widehat{n} &:= \widehat{\delta(n)} & \Uparrow_{\mathsf{ren}}\ \delta &:= 0.(\mathsf{S}\circ\delta)\\
\mathsf{ren}\ \delta\ \mathsf{int} &:= \mathsf{int} & (n.\delta)\ 0 &:= n\\
\mathsf{ren}\ \delta\ (A\to B) &:= \mathsf{ren}\ \delta\ A \to \mathsf{ren}\ \delta\ B & (n.\delta)\ (m+1) &:= \delta(m)\\
\mathsf{ren}\ \delta\ (\forall.\ A) &:= \forall.\ \mathsf{ren}\ (\Uparrow_{\mathsf{ren}}\ \delta)\ A & (\delta\circ\delta')\ n &:= \delta(\delta' n)\\
\mathsf{ren}\ \delta\ (\exists.\ A) &:= \exists.\ \mathsf{ren}\ (\Uparrow_{\mathsf{ren}}\ \delta)\ A
\end{aligned}$$

Note that, when we move underneath a binder with $\Uparrow_{\mathsf{ren}}$, the variable renaming leaves the variable $\widehat{0}$ unchanged and increases all the renamed variables (coming from δ). That is, it shifts δ by one (through the cons) *and* increases its results by one (through the S). The former is required to replace the right variables underneath the binder and the latter to not screw up which binder the (free) variables in δ refer to.

Given the capture avoiding parallel substitution $A[\sigma]$, we can derive a single point substitution $A[B/\ ] := A[B.\mathsf{id}]$ needed for the type system. In the type system, we often want to replace the variable of a binder with a type (*e.g.*, in the case of BIGAPP)—that is what $A[B/\ ]$ does! It replaces the variable $\widehat{0}$ with A and decreases all other variables in A since the binder $\forall$ has been removed. To see that this makes sense, consider the rule BIGAPP for the example $E\ \langle\mathsf{int}\rangle$. If $\Delta\,;\Gamma\vdash E : \forall.\,\widehat{0}\to\widehat{1}$, then we have the desired $\Delta\,;\Gamma\vdash E\ \langle\mathsf{int}\rangle : \mathsf{int}\to\widehat{0}$ since $(\widehat{0}\to\widehat{1})[\mathsf{int}/\ ] = (\widehat{0}\to\widehat{1})[\mathsf{int}.\mathsf{id}] = \mathsf{int}\to\widehat{0}$.

Exercise 20 (De Bruijn Terms) For terms, we have opted for a named representation (*i.e.*, with strings as variables and binders which carry a variable). Define a De Bruijn representation of the terms and define a capture avoiding substitution operation on them.
●

2.3 System F with De Bruijn types

In the rest of these notes, we will pretend to work in ordinary System F with named binders, because it significantly improves readability. However, since doing the same in Coq is not an option (due to the broken type substitution), we will use a version System F with De Bruijn types in Coq. In the following, we make precise what System F with De Bruijn types looks like.

$$\begin{array}{rrcl}
\text{Types} & A, B & ::= & \ldots \mid \forall.\ A \mid \exists.\ A \mid \widehat{n}\\
\text{Type Variable Contexts} & \Delta & := & \mathbb{N}\\
\text{Source Terms} & E & ::= & \ldots \mid \Lambda.\ E \mid E\ \langle A\rangle \mid \mathsf{pack}\ [A, E]\ \mathsf{as}\ \exists.\ B\\
& & & \mid \mathsf{unpack}\ E\ \mathsf{as}\ [x]\ \mathsf{in}\ E'\\
\text{Runtime Terms} & e & ::= & \ldots \mid \Lambda.\ e \mid e\ \langle\rangle \mid \mathsf{pack}\ e \mid \mathsf{unpack}\ e\ \mathsf{as}\ x\ \mathsf{in}\ e'\\
\text{(Runtime) Values} & v & ::= & \ldots \mid \Lambda.\ e \mid \mathsf{pack}\ v\\
\text{Evaluation Contexts} & K & ::= & \ldots \mid K\ \langle\rangle \mid \mathsf{pack}\ K \mid \mathsf{unpack}\ K\ \mathsf{as}\ x\ \mathsf{in}\ e
\end{array}$$

Contextual operational semantics $\boxed{e_1 \leadsto_{\mathrm{b}} e_2}$

BIGBETA
$(\Lambda.\ e)\ \langle\rangle \leadsto_{\mathrm{b}} e$

UNPACK
$\mathsf{unpack}\ (\mathsf{pack}\ v)\ \mathsf{as}\ x\ \mathsf{in}\ e \leadsto_{\mathrm{b}} e[v/x]$

With De Bruijn types, our type variable contexts become natural numbers. Since binders use successive numbers, we just have to keep track of a bound on the maximum type variable used.

Type Well-Formedness $\boxed{\Delta \vdash A}$

$$\textsc{wf-int}\quad \Delta \vdash \mathsf{int} \qquad \textsc{wf-lam}\quad \frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \to B} \qquad \textsc{wf-tvar}\quad \frac{n < \Delta}{\Delta \vdash \widehat{n}} \qquad \textsc{wf-tforall}\quad \frac{1 + \Delta \vdash A}{\Delta \vdash \forall.\, A} \qquad \textsc{wf-texists}\quad \frac{1 + \Delta \vdash A}{\Delta \vdash \exists.\, A}$$

In the typing judgments, we have to take a bit of care when introducing new type variables to the context. We introduce new type variables when we move underneath a type binder (*i.e.*, $\exists.$ or $\forall.$). As with substitution, when we move underneath a binder, we must be careful to not screw up the mapping between free variables and the binders they refer to (in this case in the context Γ). That is, the variable $\widehat{0}$ will now point to the binder we just moved under, and all other variables have to be increased by one. We do the increase with the operation $\uparrow A := \mathsf{ren}\ \mathsf{S}\ A$ and lift it to typing contexts $\uparrow \Gamma$ pointwise.

Church-style typing $\boxed{\Delta\,; \Gamma \vdash E : A}$

$$\ldots \qquad \textsc{lam}\quad \frac{\Delta \vdash A \qquad \Delta\,; \Gamma, x : A \vdash E : B}{\Delta\,; \Gamma \vdash \lambda x : A.\, E : A \to B} \qquad \textsc{bigLam}\quad \frac{1 + \Delta; \uparrow \Gamma \vdash E : A}{\Delta\,; \Gamma \vdash \Lambda.\, E : \forall.\, A}$$

$$\textsc{bigApp}\quad \frac{\Delta, \Gamma \vdash E : \forall.\, B \qquad \Delta \vdash A}{\Delta\,; \Gamma \vdash E\ \langle A \rangle : B[A/\ ]} \qquad \textsc{pack}\quad \frac{1 + \Delta \vdash A \qquad \Delta\,; \Gamma \vdash E : A[B/\ ] \qquad \Delta \vdash B}{\Delta\,; \Gamma \vdash \mathsf{pack}\ [B, E]\ \mathsf{as}\ \exists.\, A : \exists.\, A}$$

$$\textsc{unpack}\quad \frac{\Delta\,; \Gamma \vdash E : \exists.\, A \qquad 1 + \Delta\,; (\uparrow \Gamma), x : A \vdash E' : \uparrow B \qquad \Delta \vdash B}{\Delta\,; \Gamma \vdash \mathsf{unpack}\ E\ \mathsf{as}\ [x]\ \mathsf{in}\ E' : B}$$

Curry-style typing $\boxed{\Delta\,; \Gamma \vdash e : A}$

$$\ldots \qquad \textsc{lam}\quad \frac{\Delta \vdash A \qquad \Delta\,; \Gamma, x : A \vdash e : B}{\Delta\,; \Gamma \vdash \lambda x.\, e : A \to B} \qquad \textsc{app}\quad \frac{\Delta\,; \Gamma \vdash e_1 : A \to B \qquad \Delta\,; \Gamma \vdash e_2 : A}{\Delta\,; \Gamma \vdash e_1\, e_2 : B}$$

$$\textsc{bigLam}\quad \frac{1 + \Delta; \uparrow \Gamma \vdash e : A}{\Delta\,; \Gamma \vdash \Lambda.\, e : \forall.\, A} \qquad \textsc{bigApp}\quad \frac{\Delta, \Gamma \vdash e : \forall.\, B \qquad \Delta \vdash A}{\Delta\,; \Gamma \vdash e\ \langle\rangle : B[A/\ ]}$$

$$\textsc{pack}\quad \frac{1 + \Delta \vdash A \qquad \Delta\,; \Gamma \vdash e : A[B/\ ] \qquad \Delta \vdash B}{\Delta\,; \Gamma \vdash \mathsf{pack}\ e : \exists.\, A}$$

$$\textsc{unpack}\quad \frac{\Delta\,; \Gamma \vdash e : \exists.\, A \qquad 1 + \Delta\,; (\uparrow \Gamma), x : A \vdash e' : \uparrow B \qquad \Delta \vdash B}{\Delta\,; \Gamma \vdash \mathsf{unpack}\ e\ \mathsf{as}\ x\ \mathsf{in}\ e' : B}$$

2.4 Type Safety

In this section, we prove type safety for System F. In the exercises, you will extend the proofs to also cover existential types. As already stated above, we will continue to work with named binders on paper from now on, and only use De Bruijn indices when working formally in Coq.

Theorem 23 (Progress)**.** *Theorem 10 remains valid: If $\vdash e : A$, then e is progressive.*

Proof. Remember we are doing induction on $\vdash e : A$.

Case 1: BIGLAM, $e = \Lambda.\, e_1$. e is a value.

Case 2: BIGAPP, $e = v\,\langle\rangle$ and $\vdash v : \forall\alpha.\, B$.
$v = \Lambda.\, e'$ for some e' by inversion (or canonical forms), and $e \rightsquigarrow e'$ by BIGBETA, CTX.

Case 3: BIGAPP, $e = e'\,\langle\rangle$ where e' is not a value.
By inversion and induction, we have that e' is progressive and hence there exists e'' s.t. $e' \rightsquigarrow e''$. Thus we have $e \rightsquigarrow e''\,\langle\rangle$.

Case 4: PACK, UNPACK. See Exercise 21. □

Lemma 24 (Type Substitution)**.**

If $\Delta, \alpha\,;\Gamma \vdash e : A$ and $\Delta \vdash B$, then $\Delta\,;\Gamma[B/\alpha] \vdash e : A[B/\alpha]$.

Technically speaking, we must update the composition and decomposition lemmas to handle the new evaluation contexts. However, we will omit this trivial proof.

Theorem 25 (Preservation)**.**

Theorem 14 remains valid: If $\vdash e : A$ and $e \rightsquigarrow e'$, then $\vdash e' : A$.

Proof. The proof of preservation remains the same. We only need to update the proof for *base* preservation (Lemma 13).

We have: To show:

Case: BIGBETA

Have $e = (\Lambda.\, e_1)\,\langle\rangle$ and $e' = e_1$ $\qquad\vdash e_1 : B[C/\alpha]$

By inversion on $\vdash e : A$, have $\vdash \Lambda.\, e_1 : \forall\alpha.\, B$ s.t. $A = B[C/\alpha]$, $\vdash C$.

By another inversion, have $\alpha\,;\emptyset \vdash e_1 : B$.

We are done by type substitution.

Case: UNPACK

See Exercise 21.

□

Exercise 21 Extend the proofs of progress and preservation to handle existential types.

●

Exercise 22 (Universal Fun) For this and the following exercises, we are working in System F with products and sums.

a) Define the type of function composition, and implement it.

b) Define a function swapping the first two arguments of any other function, and give its type.

c) Given two functions of type $A \to A'$ and $B \to B'$, it is possible to "map" these functions into products, obtaining an function $A \times B \to A' \times B'$. Write down such a mapping function and its type.

d) Do the same with sums.

●

Exercise 23 (Existential Fun) In your first semester at UdS, when you learned ML, you may have seen a signature very similar to this one:

```
signature ISET = sig
  type set
  val empty    : set
  val singleton: int -> set
  val union    : set -> set -> set
  val subset   : set -> set -> bool
end
```

Assume we have a primitive type bool in our language, with two literals for true and false. We also need a corresponding conditional $\mathsf{if}\, e_0 \,\mathsf{then}\, e_1 \,\mathsf{else}\, e_2$. Assume that besides addition, we also have subtraction and the comparison operators ($=$, $\neq$, $<$, $\leq$, $>$, $\geq$) on integers. Furthermore, assume we can write arbitrary recursive functions with $\mathsf{fix}_{A,B}\, f\, x.\, e$. The typing rule is

$$\textsc{Rec}\quad \frac{\Gamma, f : A \to B, x : A \vdash E : B}{\Gamma \vdash \mathsf{fix}_{A,B}\, f\, x.\, e : A \to B}$$

For example, the Fibonacci function could be written as follows:

$$\mathsf{fix}_{\mathsf{int},\mathsf{int}}\, \mathit{fib}\; x.\, \mathsf{if}\, x \leq 1 \,\mathsf{then}\, x \,\mathsf{else}\, \mathit{fib}\, ((x - \overline{2})) + \mathit{fib}\, (x - \overline{1})$$

This term has type $\mathsf{int} \to \mathsf{int}$. Finally, assume that the language has record types. Records are, syntactically, a bit of a mouthful:

Types	$A ::= \cdots$	$\mid \{(\mathit{lab} : A)^*\}$
Runtime Terms	$e ::= \cdots$	$\mid \{(\mathit{lab} := e)^*\} \mid e.\mathit{lab}$
(Runtime) Values	$v ::= \cdots$	$\mid \{(\mathit{lab} := v)^*\}$
Eval. Contexts	$K ::= \cdots$	$\mid \{(\mathit{lab} := v)^*, \mathit{lab} := K, (\mathit{lab} := e)^*\} \mid K.\mathit{lab}$

but their typing and primitive reduction rules are quite similar to those for the unit type and binary products:

$$\textsc{Record}\quad \frac{\Delta\,;\Gamma \vdash e_1 : A_1 \qquad \cdots \qquad \Delta\,;\Gamma \vdash e_n : A_n}{\Delta\,;\Gamma \vdash \{\mathit{lab}_1 := e_1, \ldots, \mathit{lab}_n := e_n\} : \{\mathit{lab}_1 : A_1, \ldots, \mathit{lab}_n : A_n\}}$$

$$\textsc{Project}\quad \{\mathit{lab}_1 := v_1, \ldots, \mathit{lab}_n := v_n\}.\mathit{lab}_i \leadsto_{\mathrm{b}} v_i \text{ when } 1 \leq i \leq n$$

Here, the metavariable *lab* ranges over a denumerable set of *labels* (disjoint from variables and type variables), and the notation $(X)^*$ denotes a finite list $X_1, X_2, \ldots, X_n$ of X's. The record $v := \{\mathsf{add} := \lambda x.\, \lambda y.\, x + y, \mathsf{sub} := \lambda x.\, \lambda y.\, x - y, \mathsf{neg} := \lambda x.\, \overline{0} - x\}$, for example, comprises components $v.\mathsf{add}$, $v.\mathsf{sub}$, and $v.\mathsf{neg}$ implementing integer addition, subtraction, and negation functions. We presuppose that the components of any record have distinct labels. Thus, $\{\mathsf{a} := \mathsf{true}, \mathsf{b} := \{\mathsf{a} := \mathsf{false}\}\}$ is syntactically well-formed but $\{\mathsf{a} := \mathsf{true}, \mathsf{a} := \mathsf{false}\}$ is not, due to the repetition of label a.

Now, let's do some programming with existential types.

a) Define a type A_{ISET} that corresponds to the signature `ISET` given above.

b) Define an implementation of A_{ISET}, with the operations actually doing what you would intuitively expect. Notice that you don't have lists, so you will have to find some other representation of finite sets. (The tricky part of this exercise is making sure that the subset check is a terminating function.)

c) Define a type A_{ISETE} that extends type A_{ISET} with a function that tests if two sets are equal. Define a function of type $A_{\mathsf{ISET}} \to A_{\mathsf{ISETE}}$ that transforms any arbitrary implementation of A_{ISET} into an implementation of A_{ISETE}, by adding an implementation of the equality function.

●

2.5 Church encodings

System F allows us to encode other types using universal types. These encodings are called Church encodings.

The empty type

$$\mathbf{0} := \forall \alpha.\, \alpha$$

The unit type

$$\mathbf{1} := \forall \alpha.\, \alpha \to \alpha$$
$$() := \Lambda.\, \lambda x.\, x$$

Booleans

$$\begin{aligned}
\mathsf{bool} &:= \forall \alpha.\, \alpha \to \alpha \to \alpha \\
\mathsf{true} &:= \Lambda.\, \lambda t.\, \lambda f.\, t \\
\mathsf{false} &:= \Lambda.\, \lambda t.\, \lambda f.\, f \\
\mathsf{if}_C\, v \,\mathsf{then}\, v_1 \,\mathsf{else}\, v_2 &:= v\, \langle C \rangle\, v_1\, v_2 \\
\mathsf{if}_C\, e \,\mathsf{then}\, e_1 \,\mathsf{else}\, e_2 &:= (e\, \langle \mathbf{1} \to C \rangle\, (\lambda ().\, e_1)\, (\lambda ().\, e_2))\, ()
\end{aligned}$$

The C type in $\mathsf{if}_C\, e \,\mathsf{then}\, e_1 \,\mathsf{else}\, e_2$ denotes the type of e_1 and e_2, which is the return type of the expression. Also note that e_1 and e_2 are "hidden" under a lambda abstraction to maintain a call-by-value semantics.

Statement 26. $\textsf{if false then } e_1 \textsf{ else } e_2 \rightsquigarrow^\star e_2$.

Proof.

$$\begin{aligned}
\textsf{if false then } e_1 \textsf{ else } e_2 &:= ((\Lambda.\, \lambda t.\, \lambda f.\, f)\ \langle \mathbf{1} \to C\rangle\ (\lambda().\, e_1)\ (\lambda().\, e_2))\ ()\\
&\rightsquigarrow ((\lambda t.\, \lambda f.\, f)\ (\lambda().\, e_1)\ (\lambda().\, e_2))\ ()\\
&\rightsquigarrow^\star (\lambda().\, e_2)\ ()\\
&\rightsquigarrow e_2
\end{aligned}$$

□

Product types

$$\begin{aligned}
A \times B &:= \forall \alpha.\, (A \to B \to \alpha) \to \alpha\\
\langle v_1, v_2\rangle &:= \Lambda\alpha.\, \lambda p : A \to B \to \alpha.\, p\, v_1\, v_2\\
\langle e_1, e_2\rangle &:= \textsf{let}\, x_2 = e_2 \textsf{ in let}\, x_1 = e_1 \textsf{ in } \langle x_1, x_2\rangle\\
\pi_1\, e &:= e\ \langle A\rangle\ (\lambda x : A, y : B.\, x)\\
\pi_2\, e &:= e\ \langle B\rangle\ (\lambda x : A, y : B.\, y)
\end{aligned}$$

Church numerals

$$\begin{aligned}
\textsf{nat} &:= \forall\alpha.\, \alpha \to (\alpha \to \alpha) \to \alpha\\
\textsf{zero} &:= \Lambda\alpha.\, \lambda z : \alpha.\, \lambda s : \alpha \to \alpha.\, z\\
\overline{n} &:= \Lambda\alpha.\, \lambda z : \alpha.\, \lambda s : \alpha \to \alpha.\, s^n(z)\\
\textsf{succ} &:= \lambda n : \textsf{nat}.\, \Lambda\alpha.\, \lambda z : \alpha.\, \lambda s : \alpha \to \alpha.\, s\ (n\ \langle\alpha\rangle\ z\ s)\\
\textsf{iter}_C &:= \lambda n : \textsf{nat}.\, \lambda z : C.\, \lambda s : C \to C.\, n\ \langle C\rangle\ z\ s
\end{aligned}$$

Statement 27. $\textsf{iter}_C\, \overline{n}\, z\, s \rightsquigarrow^\star$ *result of* $s^n(z)$.

Exercise 24 Define a Church encoding for sum types in System F.

a) Define the encoding of the type $A + B$.

b) Implement $\textsf{inj}_1\, v$, $\textsf{inj}_2\, v$, and $\textsf{match}\, v_0 \textsf{ of inj}_1\, x_1.\, e_1 \mid \textsf{inj}_2\, x_2.\, e_2 \textsf{ end}$.

c) Prove that your encoding has the same reduction behaviors as the built-in sum type.

d) Prove that your encoding also has the same typing rules as the built-in sum type.

●

Exercise 25 Lists in System F can be Church encoded as

$$\textsf{list}\, A := \forall\alpha.\, \alpha \to (A \to \alpha \to \alpha) \to \alpha$$

a) Implement $\textsf{nil}$, which represents the empty list, and $\textsf{cons}\, v_1\, v_2$, which constructs a new list by prepending v_1 of type A to the list v_2 of type $\textsf{list}\, A$.

b) Define the typing rules for lists, and prove that your encoding satisfies those rules.

c) Define a function head of type $\mathsf{list}\, A \to A + \mathbf{1}$. $\mathsf{head}\, l$ should evaluate to $\mathsf{inj}_1\, v$ if l evaluates to a list whose head is v, or $\mathsf{inj}_2\, ()$ if l evaluates to nil. You do not need to verify its correctness.

d) Define a function tail of type $\mathsf{list}\, A \to \mathsf{list}\, A$, which computes the tail of the list. You do not need to verify its correctness.

2.6 Termination

We extend the semantic model to handle universal and existential types. The naïve approach (*i.e.*, quantifying over arbitrary syntactic types in the interpretation of universals and existentials) does not yield a well-founded relation. The reason for this is that the type substituted in for the type variable may well be larger than the original universal or existential type. Instead, we quantify over so-called semantic types. To make this work, we need to introduce semantic type substitutions which map type variables to semantic types in our model (we assume that type substitutions δ are total; they may assign bogus semantic types, like the semantic type of all closed values, for type variables we do not care about).

Big-Step Semantics

$\boxed{e \downarrow v}$

$$\textsc{bigLambda}\quad \Lambda.\, e \downarrow \Lambda.\, e \qquad \textsc{bigApp}\quad \frac{e_1 \downarrow \Lambda.\, e \qquad e \downarrow v}{e_1\, \langle\rangle \downarrow v} \qquad \textsc{pack}\quad \frac{e \downarrow v}{\mathsf{pack}\, e \downarrow \mathsf{pack}\, v} \qquad \textsc{unpack}\quad \frac{e \downarrow \mathsf{pack}\, v \qquad e'[v/x] \downarrow v'}{\mathsf{unpack}\, e\, \mathsf{as}\, x\, \mathsf{in}\, e' \downarrow v'}$$

Semantic Types

$\boxed{\tau \in \mathit{SemType}}$

$$\begin{aligned} \mathit{SemType} &:= \mathbb{P}(\mathit{CVal}) \\ \mathit{CVal} &:= \{v \mid v \text{ closed}\} \end{aligned}$$

Value Relation

$\boxed{\mathcal{V}[\![A]\!]\delta}$

$$\begin{aligned} \mathcal{V}[\![\alpha]\!]\delta &:= \delta(\alpha) \\ \mathcal{V}[\![\mathsf{int}]\!]\delta &:= \{\overline{n} \mid n \in \mathbb{Z}\} \\ \mathcal{V}[\![A \to B]\!]\delta &:= \{(\lambda x.\, e) \in \mathit{CVal} \mid \forall v.\, v \in \mathcal{V}[\![A]\!]\delta \Rightarrow e[v/x] \in \mathcal{E}[\![B]\!]\delta\} \\ \mathcal{V}[\![\forall \alpha.\, A]\!]\delta &:= \{(\Lambda.\, e) \in \mathit{CVal} \mid \forall \tau \in \mathit{SemType}.\, e \in \mathcal{E}[\![A]\!](\delta, \alpha \mapsto \tau)\} \\ \mathcal{V}[\![\exists \alpha.\, A]\!]\delta &:= \{\mathsf{pack}\, v \mid \exists \tau \in \mathit{SemType}.\, v \in \mathcal{V}[\![A]\!](\delta, \alpha \mapsto \tau)\} \end{aligned}$$

Expression Relation

$\boxed{\mathcal{E}[\![A]\!]\delta}$

$$\mathcal{E}[\![A]\!]\delta := \{e \mid \exists v.\, e \downarrow v \wedge v \in \mathcal{V}[\![A]\!]\delta\}$$

Context Relation

$\boxed{\mathcal{G}[\![\Gamma]\!]\delta}$

$$\textsc{crel-empty}\quad \mathcal{G}[\![\Gamma]\!]\delta(\gamma) \qquad \textsc{crel-elem}\quad \frac{\mathcal{G}[\![\Gamma]\!]\delta(\gamma) \qquad v \in \mathcal{V}[\![A]\!]\delta}{\mathcal{G}[\![\Gamma, x : A]\!]\delta(\gamma, x \mapsto v)}$$

Semantic Typing $\boxed{\Delta ; \Gamma \vDash e : A}$

$$\Delta ; \Gamma \vDash e : A := \forall \delta. \forall \gamma \in \mathcal{G}[\![\Gamma]\!]\delta. \gamma(e) \in \mathcal{E}[\![A]\!]\delta$$

Theorem 28 (Semantic Soundness)**.**

Theorem 18 remains valid: If $\Delta ; \Gamma \vdash e : A$, then $\Delta ; \Gamma \vDash e : A$.

Proof. By induction on $\Delta ; \Gamma \vdash e : A$ and then using the compatibility lemmas. The existing cases need to be adapted to the extended model with type variable substitutions. This is a straightforward exercise. We present the cases for universal types in Lemma 29 and Lemma 30. You will finish the cases for existential types in Exercise 26. $\square$

We write our compatibility lemmas as inference rules. This is just a notational device; each is an implication from its premises to its conclusion.

Lemma 29 (Compatibility for type abstraction; cf. BIGLAM)**.**

$$\frac{\Delta, \alpha ; \Gamma \vDash e : A}{\Delta, \Gamma \vDash \Lambda.\, e : \forall \alpha.\, A}$$

Proof.

We have:	To show:
$\Delta, \alpha ; \Gamma \vDash e : A$	$\Delta, \Gamma \vDash \Lambda.\, e : \forall \alpha.\, A$
Suppose δ and $\gamma \in \mathcal{G}[\![\Gamma]\!]\delta$	$\gamma(\Lambda.\, e) \in \mathcal{E}[\![\forall \alpha.\, A]\!]\delta$
	By BIGLAMBDA: $\Lambda.\, \gamma(e) \in \mathcal{V}[\![\forall \alpha.\, A]\!]\delta$
Suppose $\tau \in \mathit{SemType}$	$\gamma(e) \in \mathcal{E}[\![A]\!](\delta, \alpha \mapsto \tau)$
Have: $\delta' = (\delta, \alpha \mapsto \tau)$ is a type substitution	
	By applying $\Delta, \alpha ; \Gamma \vDash e : A$, we only need to show $\gamma \in \mathcal{G}[\![\Gamma]\!]\delta'$

To finish the proof, we make use of an auxiliary lemma which we do not prove here:

Lemma (Boring Lemma 1)**.** *If δ_1 and δ_2 agree on the free type variables of Γ and A, then*

$$\mathcal{V}[\![A]\!]\delta_1 = \mathcal{V}[\![A]\!]\delta_2$$
$$\mathcal{G}[\![\Gamma]\!]\delta_1 = \mathcal{G}[\![\Gamma]\!]\delta_2$$
$$\mathcal{E}[\![A]\!]\delta_1 = \mathcal{E}[\![A]\!]\delta_2$$

$\square$

Lemma 30 (Compatibility for type application; cf. BIGAPP)**.**

$$\frac{\Delta ; \Gamma \vDash e : \forall \alpha.\, B \qquad \Delta \vdash A}{\Delta, \Gamma \vDash e\,\langle\rangle : B[A/\alpha]}$$

Proof.

We have:	To show:
$\Delta\,;\Gamma \vDash e : \forall\alpha.\, B$	$\Delta, \Gamma \vDash e\ \langle\rangle : B[A/\alpha]$
$\Delta \vdash A$	
Suppose δ is a type substitution, $\gamma \in \mathcal{G}\llbracket\Gamma\rrbracket\delta$	$\gamma(e\ \langle\rangle) \in \mathcal{E}\llbracket B[A/\alpha]\rrbracket\delta$
	$\gamma(e)\ \langle\rangle \in \mathcal{E}\llbracket B[A/\alpha]\rrbracket\delta$
	$\exists\hat{v}.\, \gamma(e)\ \langle\rangle \downarrow \hat{v} \in \mathcal{V}\llbracket B[A/\alpha]\rrbracket\delta$
	By BIGAPP: $\exists\hat{e}, \hat{v}.\, \gamma(e) \downarrow \Lambda.\, \hat{e}$ and $\hat{e} \downarrow \hat{v} \in \mathcal{V}\llbracket B[A/\alpha]\rrbracket\delta$
	$\exists\hat{e}.\, \gamma(e) \downarrow \Lambda.\, \hat{e}$ and $\hat{e} \in \mathcal{E}\llbracket B[A/\alpha]\rrbracket\delta$
From $\Delta\,;\Gamma \vDash e : \forall\alpha.\, B$:	
$\gamma(e) \in \mathcal{E}\llbracket\forall\alpha.\, B\rrbracket\delta$	
$\gamma(e) \downarrow v \in \mathcal{V}\llbracket\forall\alpha.\, B\rrbracket\delta$ for some v	
$v = \Lambda.\, e'$ and $\forall\tau \in \mathit{SemType}.\, e' \in \mathcal{E}\llbracket B\rrbracket(\delta, \alpha \mapsto \tau)$ for some e'	
Pick $\hat{e} := e'$	$e' \in \mathcal{E}\llbracket B[A/\alpha]\rrbracket\delta$
Set $\tau := \mathcal{V}\llbracket A\rrbracket\delta$ and $\delta' := (\delta, \alpha \mapsto \tau)$	$\mathcal{V}\llbracket A\rrbracket\delta \in \mathit{SemType}$
	$\mathcal{E}\llbracket B[A/\alpha]\rrbracket\delta = \mathcal{E}\llbracket B\rrbracket\delta'$

Again, to finish this proof we rely on auxiliary lemmas:

Lemma (Boring Lemma 2)**.**

$$\mathcal{V}\llbracket B\rrbracket(\delta, \alpha \mapsto \mathcal{V}\llbracket A\rrbracket\delta) = \mathcal{V}\llbracket B[A/\alpha]\rrbracket\delta$$
$$\mathcal{E}\llbracket B\rrbracket(\delta, \alpha \mapsto \mathcal{V}\llbracket A\rrbracket\delta) = \mathcal{E}\llbracket B[A/\alpha]\rrbracket\delta$$

Lemma (Boring Lemma 3)**.** *If δ is a type substitution and $\Delta \vdash A$, then $\mathcal{V}\llbracket A\rrbracket\delta \in \mathit{SemType}$.*

□

Exercise 26 Prove the compatibility lemmas for the cases of existential types. •

2.7 Free Theorems

Our model allows us to prove several theorems about specific universal types. This class of theorems was coined “free theorems” by Wadler [10].

Example ($\forall\alpha.\, \alpha$)**.** *We prove that there exists no term e s.t. $\vdash e : \forall\alpha.\, \alpha$.*

Proof.

We have:	To show:
Suppose, by way of contradiction, $\vdash e : \forall\alpha.\, \alpha$	$\bot$
Let $\delta_{\mathsf{emp}}(\beta) := \emptyset$ for any type variable β.	
By Theorem 28, $e \in \mathcal{E}\llbracket\forall\alpha.\, \alpha\rrbracket\delta_{\mathsf{emp}}$, so $e \downarrow v \in \mathcal{V}\llbracket\forall\alpha.\, \alpha\rrbracket\delta_{\mathsf{emp}}$	
Pick $\tau = \emptyset$, then $v = \Lambda.\, e'$ and $e' \in \mathcal{E}\llbracket\alpha\rrbracket(\delta_{\mathsf{emp}}[\alpha \mapsto \emptyset])$	
Hence $e' \downarrow v' \in \emptyset$ for some v'	

□

Example ($\forall\alpha.\, \alpha \to \alpha$)**.** *We prove that all inhabitants of $\forall\alpha.\, \alpha \to \alpha$ are identity functions, in the sense that given a closed term f of that type, for any closed value v we have $f\ \langle\rangle\ v \downarrow v$.*

Proof.

We have:	To show:
Suppose $\vdash f : \forall \alpha.\, \alpha \to \alpha$	$f\ \langle\rangle\ v \downarrow v$
Let $\delta_{\mathsf{emp}}(\beta) := \emptyset$ for any type variable β.	
By Theorem 28, $f \downarrow f_v \in \mathcal{V}[\![\forall \alpha.\, \alpha \to \alpha]\!]\delta_{\mathsf{emp}}$	
Pick $\tau = \{v\}$	
We have $f_v = \Lambda.\, e'$ and $e' \in \mathcal{E}[\![\alpha \to \alpha]\!](\delta_{\mathsf{emp}}[\alpha \mapsto \tau])$.	
(We sometimes write this as $\mathcal{E}[\![\tau \to \tau]\!]$.)	
From $v \in \tau$, we have $e'\ v \in \mathcal{E}[\![\tau]\!]$ and thus $e'\ v \downarrow v$.	
So, $f\ \langle\rangle\ v \rightsquigarrow^* f_v\ \langle\rangle\ v = (\Lambda.\, e')\ \langle\rangle\ v \rightsquigarrow e'\ v \downarrow v$	

□

Exercise 27 We consider System F with products. For each of the following types, state a property P that holds for all inhabitants of that type ($\forall f : A.\, P$) and then prove it. Your property should be as strong as possible.

a) $\forall \alpha, \beta.\, \alpha \to \beta \to \alpha \times \beta$

b) $\forall \alpha, \beta.\, \alpha \times \beta \to \alpha$

c) $\forall \alpha, \beta.\, \alpha \to \beta$

•

Exercise 28 Prove the following: Given a closed term f of type $\forall \alpha.\, \alpha \to \alpha \to \alpha$, and any two closed values v_1, v_2, we have either $f\ \langle\rangle\ v_1\ v_2 \downarrow v_1$ or $f\ \langle\rangle\ v_1\ v_2 \downarrow v_2$. •

Exercise 29 Suppose A_1, A_2, and A are closed types and f is a closed term of type $\forall \alpha.\, (A_1 \to A_2 \to \alpha) \to \alpha$ and g is a closed term of type $A_1 \to A_2 \to A$. Prove that if $f\ \langle\rangle\ g \downarrow v$, then $\exists v_1, v_2.\, g\, v_1\, v_2 \downarrow v$. (Essentially, this means that f can do nothing but call g with some arguments.) •

Limitation of the semantic model It is important to note that our semantic model in its current form is not strong enough to prove that our Church encodings are *faithful* encodings, in the sense that we cannot prove that our encodings of values for a type encapsulate the behaviors we expect for those values. As an example, we look at the encoding of bool values.

For any value v s.t. $\vdash v : \mathsf{bool} = \forall \alpha.\, \alpha \to \alpha \to \alpha$, we have a Free Theorem:

Statement 31 (Free Theorem for bool values)**.** $\forall v_1, v_2.\, \exists v'.\, v\ \langle\rangle\ v_1\ v_2 \downarrow v' \in \{v_1, v_2\}$.

Statement 31 allows us to say that $(\mathsf{if}\, v\, \mathsf{then}\, v_1\, \mathsf{else}\, v_2) \downarrow v' \in \{v_1, v_2\}$. This result is a bit weak: it does not guarantee that two executions of $\mathsf{if}\, v\, \mathsf{then}\, v_1\, \mathsf{else}\, v_2$ will evaluate to the same value, because one execution can evaluate to v_1, while the other to v_2. Thus the theorem does not allow us to distinguish true and false values. We actually want the following stronger lemma, which encapsulates the expected behaviors of bool values.

Statement 32 (Expected behaviors of bool values)**.**

$(\forall v_1, v_2.\, v\ \langle\rangle\ v_1\ v_2 \downarrow v_1)$, *or* $(\forall v_1, v_2.\, v\ \langle\rangle\ v_1\ v_2 \downarrow v_2)$.

The lemma states that the application of v either always produces the first value (the then branch), or always produces the second value (the else branch). We show that our semantic model, which is strong enough to show Statement 31, is not strong enough to prove Statement 32. We do this by adding an extension to our language, with which we can build a bool value that satisfies Statement 31 but violates Statement 32.

We extend the language with an expression $\mathsf{if0}(e_1, e_2)$ which checks if the result of e_1 is $\overline{0}$. If so, it returns $\overline{0}$ otherwise it returns the result of e_2:

$$\mathsf{if0}(\overline{0}, v) \rightsquigarrow_{\mathrm{b}} \overline{0} \qquad \frac{v \neq \overline{0}}{\mathsf{if0}(v, w) \rightsquigarrow_{\mathrm{b}} w} \qquad \frac{\Delta\,;\Gamma \vdash e_1 : A \qquad \Delta\,;\Gamma \vdash e_2 : A}{\Delta\,;\Gamma \vdash \mathsf{if0}(e_1, e_2) : A}$$

Even with the extension, type safety and the fundamental theorem still hold. However, we can now construct the following value:

$$v_{\mathsf{bad}} := \Lambda\alpha.\, \lambda x, y : \alpha.\, \mathsf{if0}(x, y)$$

We can see that v_{bad} has type bool and satisfies Statement 31 but not Statement 32. When instantiated with int and given the arguments $\overline{0}$ and $\overline{1}$, v_{bad} returns $\overline{0}$, so the first argument. In contrast, when instantiated with int and two arguments where the first one is not zero, then v_{bad} returns the second argument. For example, we have:

$$v_{\mathsf{bad}}\ \langle\mathsf{int}\rangle\ \overline{0}\,\overline{1} \rightsquigarrow^{\star} \overline{0} \qquad\qquad v_{\mathsf{bad}}\ \langle\mathsf{int}\rangle\ \overline{1}\,\overline{0} \rightsquigarrow^{\star} \overline{0}$$

Thus, our semantic model does not distinguish different bool values. In order to prove that our Church encodings are faithful encodings, we would need to extend our model to reasoning about *relational parametricity* originally proposed by Reynolds [7].

2.8 Existential types and invariants

To prove that existential types can serve to maintain invariants, we introduce a new construct to the language: assert. $\mathsf{assert}\, e$ gets stuck when e does not evaluate to true. This construct is not syntactically well-typed, but semantically safe when used correctly (*i.e.*, when there is some guarantee that the argument will never be false).

$$\begin{array}{rrcl} \text{Source Terms} & E & ::= & \ldots \mid \mathsf{assert}\, E \\ \text{Runtime Terms} & e & ::= & \ldots \mid \mathsf{assert}\, e \\ \text{Evaluation Contexts} & K & ::= & \ldots \mid \mathsf{assert}\, K \end{array}$$

Contextual operational semantics $\boxed{e_1 \rightsquigarrow_{\mathrm{b}} e_2}$

$$\ldots \qquad\qquad \frac{\textsc{assert-true}}{\mathsf{assert\ true} \rightsquigarrow_{\mathrm{b}} ()}$$

We can now use the assert construct to assert invariants of our implementations of existential types.

Consider the following signature.

$$\mathrm{BIT} := \exists\alpha.\ \{\mathrm{bit} : \alpha,\ \mathrm{flip} : \alpha \to \alpha,\ \mathrm{get} : \alpha \to \mathsf{bool}\}$$

We can implement this signature as follows.

$$\text{MyBit} := \textsf{pack}\,[\textsf{int}, \{\text{bit} := \overline{0},\ \text{flip} := \lambda x.\, \overline{1} - x,\ \text{get} := \lambda x.\, x > \overline{0}\}]\ \textsf{as}\ \text{BIT}$$

It is not hard to see that MyBit implements the signature and behaves like a boolean, assuming bit is always either 1 or 0. In fact, we can use Semantic Soundness (Theorem 28) and the new assert construct to prove that this is indeed the case.

For this, we change MyBit to assert the invariant within flip and get.

$$\begin{aligned}\text{MyBit} := \textsf{pack}\,[\textsf{int}, \{&\text{bit} := \overline{0}, \text{flip} := \lambda x.\, \textsf{assert}\,(x == \overline{0} \lor x == \overline{1})\,;\, \overline{1} - x,\\ &\text{get} := \lambda x.\, \textsf{assert}\,(x == \overline{0} \lor x == \overline{1})\,;\, x > \overline{0}\}]\ \textsf{as}\ \text{BIT}\end{aligned}$$

We encode the sequential composition $e_1\,;\,e_2$ as $\textsf{let}\ x = e_1\ \textsf{in}\ e_2$ where x is not free in e_2. This, in turn, is encoded as $(\lambda x.\, e_2)\, e_1$.

There is no typing rule for assert, so our new implementation is not well-typed. This is because it is not statically obvious that the assertion will always hold. We'll have to prove it! So the best we can hope for is semantic safety: $\vDash \text{MyBit} : \text{BIT}$.

Lemma (Semantically Safe Booleans)**.**
$\text{MyBit} \in \mathcal{V}[\![\text{BIT}]\!]$.

Proof.

We have:	To show:
	$\text{MyBit} \in \mathcal{V}[\![\text{BIT}]\!]$
Pick $\tau := \{\overline{0}, \overline{1}\}$	
	It suffices to show
	$\left\{\begin{array}{l}\text{bit} := \overline{0},\\ \text{flip} := \lambda x.\, \textsf{assert}\,(x == \overline{0} \lor x == \overline{1})\,;\, \overline{1} - x,\\ \text{get} := \lambda x.\, \textsf{assert}\,(x == \overline{0} \lor x == \overline{1})\,;\, x > \overline{0}\end{array}\right\}$ $\in \mathcal{V}[\![\{\text{bit} : \alpha,\ \text{flip} : \alpha \to \alpha,\ \text{get} : \alpha \to \textsf{bool}\}]\!](\alpha \mapsto \tau)$
Thus, we are left with three cases.	
Case: bit	$\overline{0} \in \mathcal{V}[\![\alpha]\!](\alpha \mapsto \tau) = \tau$
This is true, since $\overline{0} \in \{\overline{0}, \overline{1}\}$	
Case: flip	$(\lambda x.\, \textsf{assert}\,(x == \overline{0} \lor x == \overline{1})\,;\, \overline{1} - x) \in \mathcal{V}[\![\alpha \to \alpha]\!](\alpha \mapsto \tau)$
Suppose $v \in \tau$	
	$(\textsf{assert}\,(v == \overline{0} \lor v == \overline{1})\,;\, \overline{1} - v) \in \mathcal{E}[\![\alpha]\!](\alpha \mapsto \tau)$
Since $v \in \tau$,	
have $(\textsf{assert}\,(v == \overline{0} \lor v == \overline{1})\,;\, \overline{1} - v) \rightsquigarrow ((\,)\,;\, \overline{1} - v) \rightsquigarrow \overline{1} - v \rightsquigarrow \overline{1 - n}$	
where $v = \overline{n}$	
	$\overline{1 - n} \in \mathcal{V}[\![\alpha]\!](\alpha \mapsto \tau) = \tau$
We conclude since $\overline{1 - n} \in \tau$.	
Case: get	
With similar reasoning as above, we show that the assert succeeds.	

□

Note that all our structural syntactic safety rules extend to semantic safety. In other words, if some syntactically well-typed piece of code *uses* MyBit, then the entire program will be semantically safe because every syntactic typing rule preserves the semantic safety.

(The entire program cannot be syntactically well-typed, since it contains MyBit.) This is essentially what we prove when we prove Semantic Soundess (Theorem 28). Thus, proving an implementation like the one above to be semantically safe means that no code that makes use of the implementation can break the invariant. If the entire term that contains MyBit is semantically safe, the invariant will be maintained.

Note: It would not have been necessary to add a new primitive assert to the language. Instead, we could have defined it as

$$\mathsf{assert}\, E := \mathsf{if}\, E\, \mathsf{then}\, ()\, \mathsf{else}\, \overline{0}\, \overline{0}$$
$$\mathsf{assert}\, e := \mathsf{if}\, e\, \mathsf{then}\, ()\, \mathsf{else}\, \overline{0}\, \overline{0}$$

Clearly, these definitions have the same reduction behavior – and also the same typing rule, *i.e.*, none. This again explains why we have to resort to a *semantic* proof to show that the bad event, the crash (here, using $\overline{0}$ as a function) does not occur.

Exercise 30 Consider the following existential type:

$$A := \exists \alpha.\ \{\mathsf{zero} : \alpha, \mathsf{add2} : \alpha \to \alpha, \mathsf{toint} : \alpha \to \mathsf{int}\}$$

and the following implementation

$$E := \mathsf{pack}\,[\mathsf{int}, \big\{\mathsf{zero} := \overline{0}, \mathsf{add2} := \lambda x.\, x + \overline{2}, \mathsf{toint} := \lambda x.\, x\big\}]\ \mathsf{as}\ A$$

This exercise is about proving that toint will only ever yield even numbers. To that end, we assume the existence of a closed function *even* : int → bool testing whether a number is even. Assuming that we extended our calculus with a recursion operator, we could define it as follows:

$$\begin{aligned} \mathit{even} := {} & \mathsf{fix}_{\mathsf{int},\mathsf{int}}\, f\, x. \\ & \quad \mathsf{if}\, x = \overline{0}\, \mathsf{then\ true\ else\ if}\, x = 1\, \mathsf{then\ false\ else\ if}\, x < \overline{0}\, \mathsf{then}\, f\ (x + \overline{2})\ \mathsf{else}\, f\ (x - \overline{2}) \end{aligned}$$

a) Change E such that toint asserts evenness of the argument before it is returned.

b) Prove, using the semantic model, that your new value is safe (*i.e.*, that its type erasure is in $\mathcal{V}[\![A]\!]$). You may assume that *even* works as intended, but make sure you state this assumption formally.

●

Exercise 31 Consider the following existential type, which provides an interface to any implementation of the sum type.

$$\begin{aligned} \mathrm{SUM}(A, B) := \exists \alpha.\ \{ & \mathsf{myinj}_1 : A \to \alpha, \\ & \mathsf{myinj}_2 : B \to \alpha, \\ & \mathsf{mycase} : \forall \beta.\, \alpha \to (A \to \beta) \to (B \to \beta) \to \beta\} \end{aligned}$$

Of course, we could now implement this type using the sum type that we built into the language. But instead, we could also pick a different implementation – an implementation that is in some sense "daring", since it is not syntactically well-typed. However, thanks to the abstraction provided by existential type, we can be sure that no crash will occur at runtime (*i.e.*, the program will not get stuck).

We define such an implementation as follows:

$$\begin{aligned}\mathrm{MySum}(A,B) := \mathsf{pack}\,\{&\mathsf{myinj}_1 := \lambda x.\,\langle \overline{1}, x\rangle,\\ &\mathsf{myinj}_2 := \lambda x.\,\langle \overline{2}, x\rangle,\\ &\mathsf{mycase} := \Lambda.\,\lambda x, f_1, f_2.\,\mathsf{if}\,\pi_1\,x == \overline{1}\,\mathsf{then}\,f_1\,(\pi_2\,x)\,\mathsf{else}\,f_2\,(\pi_2\,x)\}\end{aligned}$$

Your task is to show that the implementation is safe: Prove that for all closed types A, B, we have $\mathrm{MySum}(A,B) \in \mathcal{V}[\![\mathrm{SUM}(A,B)]\!]$. ●

2.9 Contextual Equivalence and Relational Parametricity

We revisit the problem of showing that our Church encodings are *full* and *faithful* encodings. They are full in the sense that the encodings include all the canonical forms we expect as elements of the type, and they are faithful in the sense that the encodings do not include those that do not behave like one of the canonical forms.

In the particular case of bool, we want to show that our encoding of true and false actually behaves like boolean values true and false respectively. As a plan of attack, we want to show that if $\vdash v : \mathsf{bool}$, then v and $\eta(v) := \mathsf{if}\,v\,\mathsf{then}\,\mathsf{true}\,\mathsf{else}\,\mathsf{false}$ have the same behaviors. That is, we show that if we use v *in the way boolean values are intended for* to construct another boolean, then the new expression should have the same behavior as v.

In order to define what it means to "behaves like", we introduce *contextual equivalence* and, to prove it, Reynolds' *relational parametricity* [7]. Note that while we work with System F here, the results extend also to languages with more complex features.

Program Contexts To define contextual equivalence, we define *program contexts*—contexts with a hole in an arbitrary position (even underneath binders).

$$\begin{aligned}\text{Program Context}\quad C ::= {}& \bullet \mid C\,e \mid e\,C \mid \lambda x.\,C \mid \Lambda\alpha.\,C \mid C\,\langle A\rangle \mid C + e \mid e + C\\ & \mid \mathsf{pack}\,C \mid \mathsf{unpack}\,C\,\mathsf{as}\,x\,\mathsf{in}\,e \mid \mathsf{unpack}\,e\,\mathsf{as}\,x\,\mathsf{in}\,C \mid \cdots\end{aligned}$$

Similar to evaluation contexts, program contexts have a filling operation which plugs in an expression for the hole $C[e]$. The definition is straightforward (and not spelled out here).

Exercise 32

a) Are there any evaluation contexts that are not also program contexts? If yes, give 3 examples. If no, explain why not.

b) Are there any program contexts that are not also evaluation contexts? If yes, give 3 examples. If no, explain why not.

c) Are there any contexts that are both program and evaluation contexts? If yes, give 3 examples. If no, explain why not.

●

Program Context Typing $\boxed{C : (\Delta\,;\Gamma \vdash A) \rightsquigarrow (\Delta'\,;\Gamma' \vdash A')}$

$$\textsc{hole}\quad \bullet : (\Delta\,;\Gamma \vdash A) \rightsquigarrow (\Delta'\,;\Gamma' \vdash A)$$

$$\textsc{lam}\quad \frac{C : (\Delta\,;\Gamma \vdash A) \rightsquigarrow (\Delta'\,;\Gamma', x : A_1 \vdash A_2)}{\lambda x.\, C : (\Delta\,;\Gamma \vdash A) \rightsquigarrow (\Delta'\,;\Gamma' \vdash A_1 \to A_2)}$$

$$\textsc{app-l}\quad \frac{C : (\Delta\,;\Gamma \vdash A) \rightsquigarrow (\Delta'\,;\Gamma' \vdash A_2 \to A') \qquad \Delta'\,;\Gamma' \vdash e : A_2}{C\, e : (\Delta\,;\Gamma \vdash A) \rightsquigarrow (\Delta'\,;\Gamma' \vdash A')}$$

$$\textsc{app-r}\quad \frac{C : (\Delta\,;\Gamma \vdash A) \rightsquigarrow (\Delta'\,;\Gamma' \vdash A_2) \qquad \Delta'\,;\Gamma' \vdash e : A_2 \to A'}{e\, C : (\Delta\,;\Gamma \vdash A) \rightsquigarrow (\Delta'\,;\Gamma' \vdash A')}$$

$$\textsc{bigLam}\quad \frac{C : (\Delta\,;\Gamma \vdash A) \rightsquigarrow (\Delta', \alpha\,;\Gamma' \vdash A')}{\Lambda\alpha.\, C : (\Delta\,;\Gamma \vdash A) \rightsquigarrow (\Delta'\,;\Gamma' \vdash \forall\alpha.\, A')}$$

$$\textsc{bigApp}\quad \frac{C : (\Delta\,;\Gamma \vdash A) \rightsquigarrow (\Delta'\,;\Gamma' \vdash \forall\alpha.\, A') \qquad \Delta' \vdash A_1}{C\,\langle A_1\rangle : (\Delta\,;\Gamma \vdash A) \rightsquigarrow (\Delta'\,;\Gamma' \vdash A'[A_1/\alpha])}$$

$$\textsc{pack}\quad \frac{C : (\Delta\,;\Gamma \vdash A) \rightsquigarrow (\Delta'\,;\Gamma' \vdash A'[A_1/\alpha]) \qquad \Delta' \vdash A_1}{\mathsf{pack}\, C : (\Delta\,;\Gamma \vdash A) \rightsquigarrow (\Delta'\,;\Gamma' \vdash \exists\alpha.\, A')}$$

$$\textsc{unpackL}\quad \frac{C : (\Delta\,;\Gamma \vdash A) \rightsquigarrow (\Delta'\,;\Gamma' \vdash \exists\alpha.\, B) \qquad \Delta', \alpha\,;\Gamma', x : B \vdash e' : D \qquad \Delta' \vdash D}{\mathsf{unpack}\, C\, \mathsf{as}\, x\, \mathsf{in}\, e' : (\Delta\,;\Gamma \vdash A) \rightsquigarrow (\Delta'\,;\Gamma' \vdash D)}$$

$$\textsc{unpackR}\quad \frac{C : (\Delta\,;\Gamma \vdash A) \rightsquigarrow (\Delta', \alpha\,;\Gamma', x : B \vdash D) \qquad \Delta'\,;\Gamma' \vdash e : \exists\alpha.\, B \qquad \Delta' \vdash D}{\mathsf{unpack}\, e\, \mathsf{as}\, x\, \mathsf{in}\, C : (\Delta\,;\Gamma \vdash A) \rightsquigarrow (\Delta'\,;\Gamma' \vdash D)}$$

$$\textsc{sumL}\quad \frac{C : (\Delta\,;\Gamma \vdash A) \rightsquigarrow (\Delta'\,;\Gamma' \vdash \mathsf{int}) \qquad \Delta'\,;\Gamma' \vdash e : \mathsf{int}}{C + e : (\Delta\,;\Gamma \vdash A) \rightsquigarrow (\Delta'\,;\Gamma' \vdash \mathsf{int})}$$

$$\textsc{sumR}\quad \frac{C : (\Delta\,;\Gamma \vdash A) \rightsquigarrow (\Delta'\,;\Gamma' \vdash \mathsf{int}) \qquad \Delta'\,;\Gamma' \vdash e : \mathsf{int}}{e + C : (\Delta\,;\Gamma \vdash A) \rightsquigarrow (\Delta'\,;\Gamma' \vdash \mathsf{int})}$$

The judgement $C : (\Delta\,;\Gamma \vdash A) \rightsquigarrow (\Delta'\,;\Gamma' \vdash A')$ essentially says that if $\Delta\,;\Gamma \vdash e : A$, then $\Delta'\,;\Gamma' \vdash C[e] : A'$. However, in contrast to program contexts K, we define the judgement *intensionally* (*i.e.*, as an inductive definition) this time. The reason for the different definition will become clear below.

Contextual Equivalence $\boxed{\Delta\,;\Gamma \vdash e_1 \equiv_{\text{ctx}} e_2 : A}$

$$\begin{aligned}&\Delta\,;\Gamma \vdash e_1 \equiv_{\text{ctx}} e_2 : A :=\\ &\qquad \forall C : (\Delta\,;\Gamma \vdash A) \rightsquigarrow (\emptyset\,;\emptyset \vdash \mathsf{int}).\, \exists n.\, C[e_1] \downarrow \overline{n} \wedge C[e_2] \downarrow \overline{n}\end{aligned}$$

Logical Equivalence $\boxed{\Delta\,;\Gamma \vdash e_1 \approx e_2 : A}$

$$\Delta\,;\Gamma \vdash e_1 \approx e_2 : A := \forall \delta.\, \forall (\gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma]\!]\delta.\, (\gamma_1(e_1), \gamma_2(e_2)) \in \mathcal{E}[\![A]\!]\delta$$

Expression Relation $\boxed{\mathcal{E}[\![A]\!]\delta}$

$$\mathcal{E}[\![A]\!]\delta := \{(e_1, e_2) \mid \exists v_1, v_2.\, e_1 \downarrow v_1 \wedge e_2 \downarrow v_2 \wedge (v_1, v_2) \in \mathcal{V}[\![A]\!]\delta\}$$

Value Relation $\boxed{\mathcal{V}[\![A]\!]\delta}$

$$\begin{aligned}\mathrm{VRel} &:= \mathbb{P}(\mathit{CVal} \times \mathit{CVal})\\ \mathcal{V}[\![\alpha]\!]\delta &:= \delta(\alpha)\\ \mathcal{V}[\![\mathsf{int}]\!]\delta &:= \{(\overline{n}, \overline{n})\}\\ \mathcal{V}[\![A \to B]\!]\delta &:= \{(\lambda x_1.\, e_1, \lambda x_2.\, e_2) \in \mathit{CVal} \times \mathit{CVal} \mid\\ &\qquad\qquad \forall (v_1, v_2) \in \mathcal{V}[\![A]\!]\delta.\, (e_1[v_1/x_1], e_2[v_2/x_2]) \in \mathcal{E}[\![B]\!]\delta\}\\ \mathcal{V}[\![\forall \alpha.\, A]\!]\delta &:= \{(\Lambda.\, e_1, \Lambda.\, e_2) \in \mathit{CVal} \times \mathit{CVal} \mid \forall R \in \mathrm{VRel}.\, (e_1, e_2) \in \mathcal{E}[\![A]\!](\delta, \alpha \mapsto R)\}\\ \mathcal{V}[\![\exists \alpha.\, A]\!]\delta &:= \{(\mathsf{pack}\, v_1, \mathsf{pack}\, v_2) \mid \exists R \in \mathrm{VRel}.\, (v_1, v_2) \in \mathcal{V}[\![A]\!](\delta, \alpha \mapsto R)\}\end{aligned}$$

Context Relation $\boxed{\mathcal{G}[\![\Gamma]\!]\delta}$

$$\mathcal{G}[\![\Gamma]\!]\delta := \left\{(\gamma, \gamma') \,\middle|\, \forall x : A \in \Gamma. (\gamma(x), \gamma'(x)) \in \mathcal{V}[\![A]\!]\delta\right\}$$

Lemma 33 (Binary value inclusion)**.** *If $(v_1, v_2) \in \mathcal{V}[\![A]\!]\delta$, then also $(v_1, v_2) \in \mathcal{E}[\![A]\!]\delta$*

Exercise 33 Prove the relational compatibility lemmas for variables and for type abstraction. That is prove

$$\frac{x : A \in \Gamma}{\Delta\,;\Gamma \vdash x \approx x : A}$$

and

$$\frac{\Delta, \alpha\,;\Gamma \vdash e \approx e' : A}{\Delta\,;\Gamma \vdash \Lambda.\, e \approx \Lambda.\, e' : \forall \alpha.\, A}$$

●

Theorem 34 (Soundness of $\approx$ w.r.t. $\equiv_{\text{ctx}}$)**.**

If $\mathit{FV}(e_1) \subseteq \mathit{dom}(\Gamma)$ and $\mathit{FV}(e_2) \subseteq \mathit{dom}(\Gamma)$ and $\Delta\,;\Gamma \vdash e_1 \approx e_2 : A$, then $\Delta\,;\Gamma \vdash e_1 \equiv_{\text{ctx}} e_2 : A$.

Proof. Suppose $C : (\Delta ; \Gamma \vdash A) \rightsquigarrow (\emptyset ; \emptyset \vdash \mathsf{int})$.
By compatibility (Lemma 35), $\emptyset ; \emptyset \vdash C[e_1] \approx C[e_2] : \mathsf{int}$.
By adequacy (Lemma 36), we are done. □

Lemma 35 (Compatibility)**.**

If $\mathit{FV}(e_1) \subseteq \mathit{dom}(\Gamma)$ *and* $\mathit{FV}(e_2) \subseteq \mathit{dom}(\Gamma)$, *and* $\Delta ; \Gamma \vdash e_1 \approx e_2 : A$
and $C : (\Delta ; \Gamma \vdash A) \rightsquigarrow (\Delta' ; \Gamma' \vdash A')$,
then $\Delta' ; \Gamma' \vdash C[e_1] \approx C[e_2] : A'$.

Proof. By induction on C and then using the compatibility lemmas for each case. □

Lemma 36 (Adequacy)**.** *If* $(e_1, e_2) \in \mathcal{E}[\![\mathsf{int}]\!]$, *then* $e_1 \downarrow \overline{n}$ *and* $e_2 \downarrow \overline{n}$ *for some* n.

Proof. By definition. □

Example 37 (Representation Independence)**.**

$$\begin{aligned}
\text{BIT} &:= \exists\alpha.\ \{\text{bit} : \alpha,\ \text{flip} : \alpha \to \alpha,\ \text{get} : \alpha \to \mathsf{bool}\} \\
\text{IntBit} &:= \mathsf{pack}\ \{\ \text{bit} := \overline{0}, \text{flip} := \lambda x.\, \overline{1} - x, \text{get} := \lambda x.\, x > \overline{0}\ \} \\
\text{BoolBit} &:= \mathsf{pack}\ \{\ \text{bit} := \mathsf{false}, \text{flip} := \lambda x.\, \mathsf{not}\, x, \text{get} := \lambda x.\, x\ \}
\end{aligned}$$

Goal: $\vdash \text{IntBit} \approx \text{BoolBit} : \text{BIT}$.

Proof. Pick $R := \{(\overline{0}, \mathsf{false}), (\overline{1}, \mathsf{true})\}$. □

Theorem 38 (Fundamental Property of the Logical Relations)**.**

If $\Gamma \vdash e : A$ *then* $\Gamma \vdash e \approx e : A$.

Proof. By induction on e and the compatibility lemmas. □

Now we can go back to proving that bool is a full and faithful encoding.

Theorem 39. *If* $\vdash e : \mathsf{bool}$, *then* $\vdash e \equiv_{\text{ctx}} \eta(e) : \mathsf{bool}$.

Proof.

We have:	To show:
$\vdash e : \mathsf{bool}$	$\vdash e \equiv_{\text{ctx}} \eta(e) : \mathsf{bool}$
	By soundness (Theorem 34) $\vdash e \approx \eta(e) : \mathsf{bool}$
	$(e, \mathsf{if}\, e \,\mathsf{then\, true\, else\, false}) \in \mathcal{E}[\![\mathsf{bool}]\!]$

By the fundamental property (Theorem 38) and our assumption,
$(e, e) \in \mathcal{E}[\![\mathsf{bool}]\!]$
So $e \downarrow v$, and (i) $(v, v) \in \mathcal{V}[\![\mathsf{bool}]\!]$.
Also $\mathsf{if}\, e \,\mathsf{then\, true\, else\, false} \downarrow b \in \{\mathsf{true}, \mathsf{false}\}$.

$$(v, b) \in \mathcal{V}[\![\mathsf{bool}]\!]$$
$$(v, b) \in \mathcal{V}[\![\forall \alpha.\, \alpha \to \alpha \to \alpha]\!]$$

Suppose $R \in \text{VRel}$, $(v_1, v_2) \in R$, $(v_1', v_2') \in R$.

$$(v\ \langle\rangle\ v_1\ v_1', b\ \langle\rangle\ v_2\ v_2') \in \mathcal{E}[\![R]\!]$$

It suffices to show $(v\ \langle\rangle\ v_1\ v_1', (\mathsf{if}\, v \,\mathsf{then\, true\, else\, false})\ \langle\rangle\ v_2\ v_2') \in \mathcal{E}[\![R]\!]$

$$(v\ \langle\rangle\ v_1\ v_1', (v\ \langle\rangle\ \mathsf{true\, false})\ \langle\rangle\ v_2\ v_2') \in \mathcal{E}[\![R]\!]$$

Pick $S := \{(v_a, v_b) \mid (v_a, v_b\ \langle\rangle\ v_2\ v_2') \in \mathcal{E}[\![R]\!]\} \in \text{VRel}$.

It suffices to show $(v\ \langle\rangle\ v_1\ v_1', v\ \langle\rangle\ \mathsf{true\, false}) \in \mathcal{E}[\![S]\!]$

We apply (i).

$$(v_1, \mathsf{true}) \in \mathcal{V}[\![S]\!]$$

Follows from the definition of S and that $(v_1, v_2) \in R$.

$$(v_1', \mathsf{false}) \in \mathcal{V}[\![S]\!]$$

Follows from the definition of S and that $(v_1', v_2') \in R$.

□

Exercise 34 Prove that our church-encodings of natural numbers in Section 2.4 are full and faithful. That is, show that if $\vdash n : \mathsf{nat}$, then $\vdash n \equiv_{\text{ctx}} n\ \langle \mathsf{nat} \rangle\ \mathsf{zero\ succ} : \mathsf{nat}$. •

Exercise 35 In Exercise 31, we have encoded sums as tagged pairs. Of course, we can also define an instance of the existential type in a trivial way by using primitive sums:

$$\begin{aligned}\mathsf{SumSum}(A, B) := \mathsf{pack}\, \{&\mathsf{myinj_1} := \lambda x.\, \mathsf{inj_1} x,\\ &\mathsf{myinj_2} := \lambda x.\, \mathsf{inj_2} x,\\ &\mathsf{mycase} := \Lambda.\, \lambda x, f_1, f_2.\, \mathsf{match}\ x\ \mathsf{of}\ \mathsf{inj}_1\, x_1.f_1\ x_1 \mid \mathsf{inj}_2\, x_2.f_2\ x_2\ \mathsf{end}\}\end{aligned}$$

Show that the two implementations are contextually equivalent, *i.e.*, $\vdash \mathsf{SumSum} \equiv_{\text{ctx}} \text{MySum} : \text{SUM}(A, B)$. •

3 Recursive Types

We extend our language with recursive types. These types allow us to encode familiar recursive data structures, as well as the recursive functions needed to program with them. A familiar example is the type of integer lists:

$$\mathsf{list} \approx \mathbf{1} + A \times \mathsf{list}$$

Since list mentions itself, it is a recursive type. We use $\approx$ here, because we will not *define* list as the right-hand side. However, it will be isomorphic.

To support such types, we introduce a new type former and two constructs that witness the isomorphism between recursive types and their "definition".

$$\begin{array}{rrcll}
\text{Types} & A, B & ::= & \ldots & \mid \mu\alpha.\, A \\
\text{Source Terms} & E & ::= & \ldots & \mid \mathsf{roll}_{\mu\alpha.\,A}\, E \mid \mathsf{unroll}_{\mu\alpha.\,A}\, E \\
\text{Runtime Terms} & e & ::= & \ldots & \mid \mathsf{roll}\; e \mid \mathsf{unroll}\; e \\
\text{(Runtime) Values} & v & ::= & \ldots & \mid \mathsf{roll}\; v \\
\text{Evaluation Contexts} & K & ::= & \ldots & \mid \mathsf{roll}\; K \mid \mathsf{unroll}\; K
\end{array}$$

Church-style typing $\boxed{\Delta\,;\Gamma \vdash E : A}$

$$\cdots \qquad \overset{\text{ROLL}}{\dfrac{\Delta\,;\Gamma \vdash E : A[\mu\alpha.\,A/\alpha]}{\Delta\,;\Gamma \vdash \mathsf{roll}_{\mu\alpha.\,A}\, E : \mu\alpha.\,A}} \qquad \overset{\text{UNROLL}}{\dfrac{\Delta\,;\Gamma \vdash E : \mu\alpha.\,A}{\Delta\,;\Gamma \vdash \mathsf{unroll}_{\mu\alpha.\,A}\, E : A[\mu\alpha.\,A/\alpha]}}$$

Curry-style typing $\boxed{\Delta\,;\Gamma \vdash e : A}$

$$\cdots \qquad \overset{\text{ROLL}}{\dfrac{\Delta\,;\Gamma \vdash e : A[\mu\alpha.\,A/\alpha]}{\Delta\,;\Gamma \vdash \mathsf{roll}\; e : \mu\alpha.\,A}} \qquad \overset{\text{UNROLL}}{\dfrac{\Delta\,;\Gamma \vdash e : \mu\alpha.\,A}{\Delta\,;\Gamma \vdash \mathsf{unroll}\; e : A[\mu\alpha.\,A/\alpha]}}$$

Contextual operational semantics $\boxed{e_1 \leadsto_{\mathrm{b}} e_2}$

$$\cdots \qquad \overset{\text{UNROLL}}{\mathsf{unroll}\;(\mathsf{roll}\; v) \leadsto_{\mathrm{b}} v}$$

Note that roll and unroll witness the isomorphism between $\mu\alpha.\,A$ and $A[\mu\alpha.\,A/\alpha]$.

With this machinery, we are now able to define list and its constructors as follows.

$$\begin{aligned}
\mathsf{list} &:= \mu\alpha.\, \mathbf{1} + \mathsf{int} \times \alpha \\
&\approx (\mathbf{1} + \mathsf{int} \times \alpha)[\mathsf{list}/\alpha] \\
&= \mathbf{1} + \mathsf{int} \times \mathsf{list} \\
\mathsf{nil} &:= \mathsf{roll}_{\mathsf{list}}\,(\mathsf{inj}_1())\\
\mathsf{cons}(h, t) &:= \mathsf{roll}_{\mathsf{list}}\,(\mathsf{inj}_2\langle h, t\rangle)
\end{aligned}$$

Exercise 36 Prove that this definition of lists enjoys the expected typing rules. •

One might wonder why we do not take list to be equivalent to its unfolding. There are two approaches to recursive types in the literature.

a) **equi**-recursive types.
   This approach makes recursive types and their (potentially) infinite set of unfoldings **equivalent**. While it may be more convenient for the programmer, it also substantially complicates the metatheory of the language. The reason for this is that the notion of equivalence has to be co-inductive to account for infinite unfoldings.

b) **iso**-recursive types.
   This the approach we are taking here. It makes recursive types and their unfolding **isomorphic**. While it may seem like it puts the burden of rolling and unrolling on the programmer, this can be (and is) hidden in practice. It does not complicate the metatheory (much).

Exercise 37 Extend the proof of type safety (*i.e.*, progress and preservation) to handle recursive types. ●

3.1 Untyped Lambda Calculus

Recursive types allow us to encode the untyped λ-calculus. The key idea here is that instead of thinking about it as "untyped", we should rather think about it as "uni-typed". We will call that type D (for *dynamic*).

To clearly separate between the host language and the language to be encoded, we introduce new notation for abstraction and application. The following typing and reduction rules should be fulfilled by these constructs.

$$\frac{\Gamma, x : D \vdash e : D}{\Gamma \vdash \mathsf{lam}\, x.\, e : D} \qquad \frac{\Gamma \vdash e_1 : D \qquad \Gamma \vdash e_2 : D}{\Gamma \vdash \mathsf{app}(e_1, e_2) : D}$$

$$\frac{e_2 \leadsto e_2'}{\mathsf{app}(e_1, e_2) \leadsto \mathsf{app}(e_1, e_2')} \qquad \frac{\vdash v_2 : D \qquad e_1 \leadsto e_1'}{\mathsf{app}(e_1, v_2) \leadsto \mathsf{app}(e_1', v_2)} \qquad \mathsf{app}(\mathsf{lam}\, x.\, e, v) \leadsto^\star e[v/x]$$

From the typing rule for application, it is clear that we will somehow have to convert from D to $D \to D$. We chose $D \approx D \to D$, *i.e.*,

$$D := \mu\alpha.\, \alpha \to \alpha.$$

With this, the remaining definitions are straight-forward.

$$\begin{aligned} \mathsf{lam}\, x.\, e &:= \mathsf{roll}_D\, (\lambda x : D.\, e) \\ \mathsf{app}(e_1, e_2) &:= (\mathsf{unroll}\, e_1)\, e_2 \end{aligned}$$

These definitions respect the typing rules given above. It remains to show that the reduction rules also hold. Here, the most interesting case is that of beta reduction.

$$\mathsf{app}(\mathsf{lam}\, x.\, e, v) = (\mathsf{unroll}\, (\mathsf{roll}\, (\lambda x.\, e)))\, v \leadsto (\lambda x.\, e)\, v \leadsto e[v/x]$$

We can now show that our language with recursive types has a well-typed divergent term. To do this, we define a term Ω that reduces to itself.

$$\begin{aligned}
&\omega : D := \mathsf{lam}\, x.\, \mathsf{app}(x,x) = \mathsf{roll}\ (\lambda x.\ (\mathsf{unroll}\ x)\ x)\\
&\Omega : D := \mathsf{app}(\omega,\omega) = (\mathsf{unroll}\ \omega)\ \omega \leadsto (\lambda x.\ (\mathsf{unroll}\ x)\ x)\ \omega \leadsto (\mathsf{unroll}\ \omega)\ \omega = \Omega
\end{aligned}$$

Exercise 38 (Keep Rollin') Use the roll and unroll primitives introduced in class to encode *typed* fixpoints. Specifically: Suppose you are given types A and B well-formed in Δ.

Define a value form $\mathsf{fix}_{A,B}\ f\ x.\ e$ satisfying the following:

$$\frac{\Delta\,;\Gamma, f : A \to B, x : A \vdash e : B}{\Delta\,;\Gamma : \mathsf{fix}_{A,B}\ f\ x.\ e : A \to B}$$

and (when A, B closed)

$$(\mathsf{fix}_{A,B}\ f\ x.\ e)\ v \leadsto^* e[\mathsf{fix}_{A,B}\ f\ x.\ e/f, v/x]$$

●

3.2 Girard's Typecast Operator ("J")

We will now show that we can obtain divergence—and encode a fixed-point combinator—by other, possibly surprising, means. We assume a typecast operator cast and a default-value operator O with the following types and semantics. (Technically, we have to change runtime terms and the base reduction rules to have types in them. We will not spell out that change here.)

$$\frac{}{\mathsf{cast} : \forall\alpha.\,\forall\beta.\ \alpha \to \beta} \qquad \frac{}{\mathsf{O} : \forall\alpha.\ \alpha}$$

$$\frac{A = B}{\mathsf{cast}\ \langle A\rangle\ \langle B\rangle\ v \leadsto v} \qquad \frac{A \neq B}{\mathsf{cast}\ \langle A\rangle\ \langle B\rangle\ v \leadsto \mathsf{O}\ \langle B\rangle}$$

$$\mathsf{O}\ \langle \mathsf{int}\rangle \leadsto \overline{0} \qquad \mathsf{O}\ \langle A \to B\rangle \leadsto \lambda x.\ \mathsf{O}\ \langle B\rangle \qquad \mathsf{O}\ \langle \forall\alpha.\ A\rangle \leadsto \Lambda\alpha.\ \mathsf{O}\ \langle A\rangle$$

Again, we encode the untyped λ-calculus. This time, we pick $D := \forall\alpha.\ \alpha \to \alpha$. It suffices to simulate roll and unroll from the previous section for the type D.

$$\begin{aligned}
\mathsf{unroll}_D &:= \lambda x : D.\ x\ \langle D\rangle\\
\mathsf{roll}_D &:= \lambda f : D \to D.\ \Lambda\alpha.\ \mathsf{cast}\ \langle D \to D\rangle\ \langle \alpha \to \alpha\rangle\ f
\end{aligned}$$

It remains to check the reduction rule for unroll (roll v).

$$\begin{aligned}
\mathsf{unroll}_D\ (\mathsf{roll}_D\ v) &\leadsto \mathsf{unroll}_D\ (\Lambda\alpha.\ \mathsf{cast}\ \langle D \to D\rangle\ \langle \alpha \to \alpha\rangle\ v)\\
&\leadsto^\star \mathsf{cast}\ \langle D \to D\rangle\ \langle D \to D\rangle\ v \leadsto v
\end{aligned}$$

Exercise 39 Encode the roll and unroll primitives using Girard's cast operator. Specifically: Suppose you are given a type A s.t. $\Delta, \alpha \vdash A$ for some Δ.

Encode $\mu\alpha.\, A$ as a type R_A together with intro and elim forms roll and unroll, satisfying the following properties, where $U_A := A[R_A/\alpha]$:

$$\Delta \vdash R_A$$

$$\Delta\,;\emptyset \vdash \mathsf{roll}_A : U_A \to R_A$$

$$\Delta\,;\emptyset \vdash \mathsf{unroll}_A : R_A \to U_A$$

and, if A is closed,

$$\mathsf{unroll}_A\,(\mathsf{roll}_A\, v) \leadsto^* v$$

●

3.3 Semantic model: Step-indexing

In this section, we want to develop a semantic model of our latest type system, including recursive types. Clearly, we will no longer be able to use this model to show termination, since we saw that we can now write diverging well-typed terms. However, as we saw in in the discussion about semantic existential types in Section 2.8, a semantic model can be helpful even if it does not prove termination: We can use it to show that ill-typed code, like MyBit and MySum, is actually semantically well-typed and hence safe to use from well-typed code.

However, when we try to naively define the value relation for recursive types, it becomes immediately clear that we have a problem: The type in the recursive occurrence of $\mathcal{V}[\![A]\!]\delta$ does not become smaller. To mitigate this, we resort to the technique of *step-indexing* [2, 1]. The core idea is to index our relations (in particular, $\mathcal{V}[\![A]\!]\delta$ and $\mathcal{E}[\![A]\!]\delta$) by the "number of steps of computation that the program may perform". This intuition is not entirely correct, but it is close enough.

$\mathcal{V}[\![A]\!]\delta$ is now a predicate over both a natural number $k \in \mathbb{N}$ and a closed value v. Intuitively, $(k, v) \in \mathcal{V}[\![A]\!]\delta$ means that no well-typed program using v at type A will "go wrong" in k steps (or less). This intuition also explains why we want these relations to be *monotone* or *downwards-closed* with respect to the step-index: If $(k, v) \in \mathcal{V}[\![A]\!]\delta$, then $\forall j \le k.\, (j, v) \in \mathcal{V}[\![A]\!]\delta$.

We will need the new notion of a program terminating in k steps with some final term:

Step-indexed termination $\boxed{e \searrow^{k} e'}$

$$\frac{\forall e'.\, e \not\leadsto e'}{e \searrow^{0} e} \qquad\qquad \frac{e \leadsto e' \qquad e' \searrow^{k} e''}{e \searrow^{k+1} e''}$$

The judgment $e \searrow^{k} e'$ means that e reduces to e' in k steps, and that e' is irreducible. Notice that, unlike the $e \downarrow v$ evaluation relation, e' does *not have to be a value.* All $e \searrow^{k} e'$ says is that e will stop computing after k steps, and it will end up in term e'. It could either be stuck (*i.e.*, have crashed), or arrived at a value.

When showing semantic soundness of step-indexing, we will rely on a few lemmas stating basic properties of the reduction relations.

Exercise 40 Prove the following statements.

Lemma 40. *If $K[e]$ is a value, then so is e.*

Lemma 41. *If e is not a value, and $K[e] \rightsquigarrow e'$, then there exists an e'' such that $e' = K[e'']$ and $e \rightsquigarrow e''$.*

Lemma 42. *If $K[e] \searrow^{k} e'$, then there exists $j \leq k$ and an e'' such that $e \searrow^{j} e''$ and $K[e''] \searrow^{k-j} e'$.*

•

Now, we can define our semantic model.

Semantic Types $\boxed{\tau \in \mathit{SemType}}$

$$\begin{aligned} \mathit{SemType} &:= \{\tau \in \mathbb{P}(\mathbb{N} \times \mathit{CVal}) \mid \forall (k,v) \in \tau.\, \forall j < k.\, (j,v) \in \tau\} \\ \mathit{CVal} &:= \{v \mid v \text{ closed}\} \end{aligned}$$

Value Relation $\boxed{\mathcal{V}[\![A]\!]\delta}$

$$\begin{aligned} \mathcal{V}[\![\alpha]\!]\delta &:= \delta(\alpha) \\ \mathcal{V}[\![\mathsf{int}]\!]\delta &:= \{(k, \overline{n})\} \\ \mathcal{V}[\![A \to B]\!]\delta &:= \{(k, \lambda x.\, e) \mid (\lambda x.\, e) \in \mathit{CVal} \wedge \forall j \leq k, v.\, (j,v) \in \mathcal{V}[\![A]\!]\delta \Rightarrow (j, e[v/x]) \in \mathcal{E}[\![B]\!]\delta\} \\ \mathcal{V}[\![\forall \alpha.\, A]\!]\delta &:= \{(k, \Lambda.\, e) \mid (\Lambda.\, e) \in \mathit{CVal} \wedge \forall \tau \in \mathit{SemType}.\, (k,e) \in \mathcal{E}[\![A]\!](\delta, \alpha \mapsto \tau)\} \\ \mathcal{V}[\![\exists \alpha.\, A]\!]\delta &:= \{(k, \mathsf{pack}\, v) \mid \exists \tau \in \mathit{SemType}.\, (k,v) \in \mathcal{V}[\![A]\!](\delta, \alpha \mapsto \tau)\} \\ \mathcal{V}[\![\mu \alpha.\, A]\!]\delta &:= \{(k, \mathsf{roll}\, v) \mid \forall j < k.\, (j,v) \in \mathcal{V}[\![A[\mu\alpha.\, A/\alpha]]\!]\delta\} \end{aligned}$$

Expression Relation $\boxed{\mathcal{E}[\![A]\!]\delta}$

$$\mathcal{E}[\![A]\!]\delta := \left\{(k,e) \,\middle|\, \forall j < k, e'.\, e \searrow^{j} e' \Rightarrow (k-j, e') \in \mathcal{V}[\![A]\!]\delta\right\}$$

Context Relation $\boxed{\mathcal{G}[\![\Gamma]\!]\delta}$

$$\mathcal{G}[\![\Gamma]\!]\delta := \{(k,\gamma) \mid \forall x : A \in \Gamma.(k, \gamma(x)) \in \mathcal{V}[\![A]\!]\delta\}$$

Semantic Typing $\boxed{\Delta\,;\Gamma \vDash e : A}$

$$\Delta\,;\Gamma \vDash e : A := \forall \delta.\, \forall (k,\gamma) \in \mathcal{G}[\![\Gamma]\!]\delta.\, (k, \gamma(e)) \in \mathcal{E}[\![A]\!]\delta$$

Notice that the value and expression relations are defined mutually recursively by induction over first the step-index, and then the type.

Furthermore, notice that the new model can cope well with non-deterministic reductions. In the old model, the assumption of determinism was pretty much built into $\mathcal{E}$: We demanded that the expression evaluates to *some* well-formed value. If there had been non-determinism, then it could have happened that some non-deterministic branches diverge or get stuck, as long as one of them ends up being a value. The new model can, in general, cope well with non-determinism: $\mathcal{E}$ is defined based on *all* expressions satisfying $\searrow$, *i.e.*, it takes into account any way that the program could compute. We no longer care about termination. If the program gets stuck after k steps on *any* non-deterministic execution, then it cannot be in the expression relation at step-index $k+1$. Since the semantic typing demands being in the relation at *all* step-indices, this means that semantically well-typed programs cannot possibly get stuck.

Definition 43 (Safety)**.** *A program e is* safe *if it does not get stuck, i.e., if for all k and e' such that $e \searrow^{k} e'$, e' is a value.*

By this definition, clearly, all semantically well-typed programs are safe. Now that the model no longer proves termination, safety is the primary motivation for even having a semantic model: As we saw in section 2.8, there are programs that are not well-typed, but thanks to the abstraction provided by existential types, they are still *safe*. Remember that "getting stuck" is our way to model the semantics of a crashing program, so what this really is all about is showing that our programs *do not crash*. Proving this is a worthwhile goal even for language that lack a termination guarantee.

Exercise 41 (Monotonicity) Prove that the expression relation $\mathcal{E}\llbracket A\rrbracket\delta$ is monotone with respect to step-indices:

- If $(k, e) \in \mathcal{E}\llbracket A\rrbracket\delta$, then $\forall j \leq k.\, (j, e) \in \mathcal{E}\llbracket A\rrbracket\delta$.

•

The first and very important lemma we show about this semantic model is the following:

Lemma 44 (Bind)**.**

If $(k, e) \in \mathcal{E}\llbracket A\rrbracket\delta$, and $\forall j \leq k.\, \forall v.\, (j, v) \in \mathcal{V}\llbracket A\rrbracket\delta \Rightarrow (j, K[v]) \in \mathcal{E}\llbracket B\rrbracket\delta$,
then $(k, K[e]) \in \mathcal{E}\llbracket B\rrbracket\delta$.

Proof.

We have:	To show:
(i) $(k, e) \in \mathcal{E}\llbracket A\rrbracket\delta$	
(ii) $\forall j \leq k.\, \forall v.\, (j, v) \in \mathcal{V}\llbracket A\rrbracket\delta \Rightarrow (j, K[v]) \in \mathcal{E}\llbracket B\rrbracket\delta$	$(k, K[e]) \in \mathcal{E}\llbracket B\rrbracket\delta$
Suppose $j < k$, $K[e] \searrow^{j} e'$	$(k-j, e') \in \mathcal{V}\llbracket B\rrbracket\delta$
By Lemma 42,	
there exist e_1 and $j_1 \leq j$ s.t. $e \searrow^{j_1} e_1$ and $K[e_1] \searrow^{j-j_1} e'$.	
By (i), $(k - j_1, e_1) \in \mathcal{V}\llbracket A\rrbracket\delta$.	
By (ii), $(k - j_1, K[e_1]) \in \mathcal{E}\llbracket B\rrbracket\delta$.	
With $K[e_1] \searrow^{j-j_1} e'$,	
we get $(k - j_1 - (j - j_1), e') \in \mathcal{V}\llbracket B\rrbracket\delta$, so we are done.	

□

Lemma 44 lets us zap subexpressions down to values, if we know that those subexpressions are in the relation. This is extremely helpful when proving that composite terms are semantically well-typed.

Next, we will re-prove a lemma that we already established for our initial version of the semantic model: Closure under Expansion. It should be noted that this lemma relies on determinism of the reduction relation.

Lemma 45 (Closure under Expansion)**.**
If e reduces deterministically for j steps, and if $e \leadsto^j e'$ and $(k, e') \in \mathcal{E}[\![A]\!]\delta$, then $(k + j, e) \in \mathcal{E}[\![A]\!]\delta$.

Proof.

We have:	To show:
(i) e reduces deterministically for j steps.	
(ii) $e \leadsto^j e'$	
(iii) $(k, e') \in \mathcal{E}[\![A]\!]\delta$	$(k + j, e) \in \mathcal{E}[\![A]\!]\delta$
Suppose $i < k + j$ and $e \searrow^i e''$	$(k + j - i, e'') \in \mathcal{V}[\![A]\!]\delta$
By (i) and (ii), $e' \searrow^{i-j} e''$.	
We have $i - j < k$.	
Thus, by (iii), $(k - (i - j), e'') \in \mathcal{V}[\![A]\!]\delta$, and we are done.	

□

Lemma 46 (Value Inclusion)**.** *If $(k, e) \in \mathcal{V}[\![A]\!]\delta$, then $(k, e) \in \mathcal{E}[\![A]\!]\delta$.*

Proof sketch. Then, e a value, so inverting $e \searrow^j e'$ gives us $j = 0$ and $e' = e$. □

These lemmas will be extremely helpful in our next theorem.

Theorem 47 (Semantic Soundness)**.** *If $\Delta ; \Gamma \vdash e : A$, then $\Delta ; \Gamma \vDash e : A$.*

Proof. Again, we do induction on $\Delta ; \Gamma \vdash e : A$ and then use the compatibility lemmas. We prove a few compatibility lemmas in Lemma 48, Lemma 49, Lemma 50, and Lemma 51. □

Lemma 48 (Compatibility for lambda abstraction; cf. LAM)**.**

$$\frac{\Delta \vdash A \qquad \Delta ; \Gamma, x : A \vDash e : B}{\Delta ; \Gamma \vDash \lambda x.\, e : A \to B}$$

Proof.

We have:	To show:
(i) $\Delta \vdash A$	
(ii) $\Delta ; \Gamma, x : A \vDash e : B$	$\Delta ; \Gamma \vDash \lambda x.\, e : A \to B$
Let $\delta, (k, \gamma) \in \mathcal{G}[\![\Gamma]\!]\delta$	$(k, \gamma(\lambda x.\, e)) \in \mathcal{E}[\![A \to B]\!]\delta$
	$(k, \lambda x.\, \gamma(e)) \in \mathcal{E}[\![A \to B]\!]\delta$
	By value inclusion, $(k, \lambda x.\, \gamma(e)) \in \mathcal{V}[\![A \to B]\!]\delta$
Suppose $j \le k$ and $(j, v) \in \mathcal{V}[\![A]\!]\delta$	$(j, \gamma(e[v/x])) \in \mathcal{E}[\![B]\!]\delta$
Let $\gamma' := \gamma[x \mapsto v]$	$(j, \gamma'(e)) \in \mathcal{E}[\![B]\!]\delta$
	By (ii), $(j, \gamma') \in \mathcal{G}[\![\Gamma, x : A]\!]\delta$
Suppose $y : C \in \Gamma, x : A$	$(j, \gamma'(y)) \in \mathcal{V}[\![C]\!]\delta$
Case: $y : C \in \Gamma$, so $y \in \mathsf{dom}(\gamma)$	
We have $\gamma'(y) = \gamma(y)$ and, by assumption, $(k, \gamma(y)) \in \mathcal{V}[\![C]\!]\delta$.	
By monotonicity, $(j, \gamma(y)) \in \mathcal{V}[\![C]\!]\delta$.	
Case: $y = x$ and $C = A$	
We have $\gamma'(x) = v$ and, by assumption, $(j, v) \in \mathcal{V}[\![A]\!]\delta$.	

□

Lemma 49 (Compatibility for function application; cf. APP)**.**

$$\frac{\Delta\,;\Gamma \vDash e_1 : A \to B \qquad \Delta\,;\Gamma \vDash e_2 : A}{\Delta\,;\Gamma \vDash e_1\, e_2 : B}$$

Proof.

We have:	To show:
(i) $\Delta\,;\Gamma \vDash e_1 : A \to B$	
(ii) $\Delta\,;\Gamma \vDash e_2 : A$	$\Delta\,;\Gamma \vDash e_1\, e_2 : B$
Let $\delta, (k, \gamma) \in \mathcal{G}\llbracket \Gamma \rrbracket \delta$	$(k, \gamma(e_1\, e_2)) \in \mathcal{E}\llbracket B \rrbracket \delta$
	$(k, (\gamma\, e_1)\,(\gamma\, e_2)) \in \mathcal{E}\llbracket B \rrbracket \delta$
$(k, \gamma\, e_1) \in \mathcal{E}\llbracket A \to B \rrbracket \delta$ by (i)	
Suppose $j \le k, (j, v_1) \in \mathcal{V}\llbracket A \to B \rrbracket \delta$	$(j, v_1\,(\gamma\, e_2)) \in \mathcal{E}\llbracket B \rrbracket \delta$
by bind with $K = \bullet\,(\gamma\, e_2)$	
$(k, \gamma\, e_2) \in \mathcal{E}\llbracket A \rrbracket \delta$ by (ii)	
$(j, \gamma\, e_2) \in \mathcal{E}\llbracket A \rrbracket \delta$ by monotonicity	
Suppose $i \le j, (i, v_2) \in \mathcal{V}\llbracket A \rrbracket \delta$	$(i, v_1\, v_2) \in \mathcal{E}\llbracket B \rrbracket \delta$
by bind with $K = v_1\, \bullet$	
$v_1 = \lambda x.\, e_1$ and $(i, e_1[v_2/x]) \in \mathcal{E}\llbracket B \rrbracket \delta$ for some e_1	
from $(j, v_1) \in \mathcal{V}\llbracket A \to B \rrbracket \delta$, $(i, v_2) \in \mathcal{V}\llbracket A \rrbracket \delta$, and $i \le j$	
$(i + 1, v_1\, v_2) \in \mathcal{E}\llbracket B \rrbracket \delta$	
by closure under expansion with BETA deterministic	
We're done by monotonicity.	

□

Lemma 50 (Compatibility for roll; cf. ROLL)**.**

$$\frac{\Delta\,;\Gamma \vDash e : A[\mu\alpha.\, A/\alpha]}{\Delta\,;\Gamma \vDash \mathsf{roll}\; e : \mu\alpha.\, A}$$

Proof.

We have:	To show:
(i) $\Delta\,;\Gamma \vDash e : A[\mu\alpha.\, A/\alpha]$	$\Delta\,;\Gamma \vDash \mathsf{roll}\; e : \mu\alpha.\, A$
Let $\delta, (k, \gamma) \in \mathcal{G}\llbracket \Gamma \rrbracket \delta$	$(k, \gamma(\mathsf{roll}\; e)) \in \mathcal{E}\llbracket \mu\alpha.\, A \rrbracket \delta$
	$(k, \mathsf{roll}\;(\gamma\, e)) \in \mathcal{E}\llbracket \mu\alpha.\, A \rrbracket \delta$
$(k, \gamma\, e) \in \mathcal{E}\llbracket A[\mu\alpha.\, A/\alpha] \rrbracket \delta$ by (i)	
Suppose $j \le k, (j, v) \in \mathcal{V}\llbracket A[\mu\alpha.\, A/\alpha] \rrbracket \delta$	$(j, \mathsf{roll}\; v) \in \mathcal{E}\llbracket \mu\alpha.\, A \rrbracket \delta$
by bind with $K = \mathsf{roll}\; \bullet$	
	By value inclusion, $(j, \mathsf{roll}\; v) \in \mathcal{V}\llbracket \mu\alpha.\, A \rrbracket \delta$
Suppose $i < j$	$(i, v) \in \mathcal{V}\llbracket A[\mu\alpha.\, A/\alpha] \rrbracket \delta$
We're done by monotonicity.	

□

Lemma 51 (Compatibility for unroll; cf. UNROLL)**.**

$$\frac{\Delta\,;\Gamma \vDash e : \mu\alpha.\, A}{\Delta\,;\Gamma \vDash \mathsf{unroll}\; e : A[\mu\alpha.\, A/\alpha]}$$

Proof.

We have:	To show:
(i) $\Delta\,;\Gamma \vDash e : \mu\alpha.\, A$	$\Delta\,;\Gamma \vDash \mathsf{unroll}\; e : A[\mu\alpha.\, A/\alpha]$
Let $\delta, (k, \gamma) \in \mathcal{G}[\![\Gamma]\!]\delta$	$(k, \gamma(\mathsf{unroll}\; e)) \in \mathcal{E}[\![A[\mu\alpha.\, A/\alpha]]\!]\delta$
	$(k, \mathsf{unroll}\;(\gamma\, e)) \in \mathcal{E}[\![A[\mu\alpha.\, A/\alpha]]\!]\delta$
$(k, \gamma\, e) \in \mathcal{E}[\![\mu\alpha.\, A]\!]\delta$ by (i)	
Suppose $j \leq k, (j, v) \in \mathcal{V}[\![\mu\alpha.\, A]\!]\delta$	$(j, \mathsf{unroll}\; v) \in \mathcal{E}[\![A[\mu\alpha.\, A/\alpha]]\!]\delta$
by bind with $K = \mathsf{unroll}\; \bullet$	
$v = \mathsf{roll}\; v'$ and (ii) $\big(\forall i < j.\, (i, v') \in \mathcal{V}[\![A[\mu\alpha.\, A/\alpha]]\!]\delta\big)$ for some v'	
from $(j, v) \in \mathcal{V}[\![\mu\alpha.\, A]\!]\delta$	$(j, \mathsf{unroll}\; (\mathsf{roll}\; v')) \in \mathcal{E}[\![A[\mu\alpha.\, A/\alpha]]\!]\delta$
Case: $j = 0$	
Trivial by definition of $\mathcal{E}[\![\cdot]\!]$.	
Case: $j > 0$ (we'll take a step)	
	By closure under expansion and UNROLL deterministic,
	$(j-1, v') \in \mathcal{E}[\![A[\mu\alpha.\, A/\alpha]]\!]\delta$
	By value inclusion, $(j-1, v') \in \mathcal{V}[\![A[\mu\alpha.\, A/\alpha]]\!]\delta$
We conclude by applying (ii) with $i = j - 1$.	

□

Remark. Note how the case of unroll crucially depends on us being able to take a step. From v being a safe value, we obtain that v' is safe *for $i < j$ steps*. This is crucial to ensure that our model is well-founded: Since the type may become larger, the step-index has to get smaller. If we had built an equi-recursive type system without explicit coercions for roll and unroll, we would need v' to be safe for j steps, and we would be stuck in the proof. But thanks to the coercions, there is a step being taken here, and we can use our assumption for v'.

Exercise 42 The step-indexed value relation for the sum type is—unsurprisingly—defined as follows:

$$\mathcal{V}[\![A + B]\!]\delta := \{(k, \mathsf{inj}_1\, v) \mid (k, v) \in \mathcal{V}[\![A]\!]\delta\} \cup \{(k, \mathsf{inj}_2\, v) \mid (k, v) \in \mathcal{V}[\![B]\!]\delta\}$$

Based on this, prove semantic soundness of the typing rules for inj_i and case. •

Exercise 43 Consider the following existential type describing an interface for lists:

$$\begin{aligned}\mathrm{LIST}(A) := \exists\alpha.\, \{&\mathsf{mynil} : \alpha,\\ &\mathsf{mycons} : A \to \alpha \to \alpha,\\ &\mathsf{mylistcase} : \forall\beta.\, \alpha \to \beta \to (A \to \alpha \to \beta) \to \beta\}\end{aligned}$$

One possible implementation of this interface represents a list $[v_1, v_2, \ldots, v_n]$ and its length n as nested pairs $\langle n, \langle v_1, \langle v_2, \langle \ldots, \langle v_n, () \rangle \ldots \rangle\rangle\rangle\rangle$. There is no type in our language that can express this, but when hidden behind the above interface, this representation can be used in a type-safe manner. The implementations of mynil and myconst are as follows:

$$\begin{aligned}\mathsf{mynil} &:= \langle \overline{0}, () \rangle \\ \mathsf{mycons} &:= \lambda a, l.\, \langle \overline{1} + \pi_1\, l, \langle a, \pi_2\, l \rangle \rangle\end{aligned}$$

a) Implement mylistcase such that

$$\begin{aligned}\mathsf{mylistcase}\,\langle\rangle\ \mathsf{mynil}\, v\, f &\rightsquigarrow^* v \\ \mathsf{mylistcase}\,\langle\rangle\ (\mathsf{mycons}\, a\, l)\, v\, f &\rightsquigarrow^* f\, a\, l\end{aligned}$$

where a, l, v, f are values.

You may assume an operation for testing integer equality:

$$\frac{\Delta\,;\Gamma \vdash e_i : \mathsf{int}}{\Delta\,;\Gamma \vdash e_1 == e_2 : \mathsf{bool}} \qquad \overline{n} == \overline{n} \rightsquigarrow_{\mathrm{b}} \mathsf{true} \qquad \frac{n \neq m}{\overline{n} == \overline{m} \rightsquigarrow_{\mathrm{b}} \mathsf{false}}$$

b) Why do we have to store the total length of the list, in addition to the bunch of nested pairs?

c) Prove that your code never crashes. To this end, prove that for any closed A, your implementation $\mathrm{MyList}(A)$ is semantically well-typed:

$$\forall k.\, (k, \mathrm{MyList}(A)) \in \mathcal{V}[\![\mathrm{LIST}(A)]\!]$$

You will need the definition of the value relation for pairs, which goes as follows:

$$\mathcal{V}[\![A \times B]\!]\delta := \{(k, \langle v_1, v_2 \rangle) \mid (k, v_1) \in \mathcal{V}[\![A]\!]\delta \wedge (k, v_2) \in \mathcal{V}[\![B]\!]\delta\}$$

●

3.4 The Curious Case of the Ill-Typed but Safe Z-Combinator

We have seen the well-typed fixpoint combinator $\mathsf{fix}_{A,B}\, f\, x.\, e$. In this section, we will look at a closely-related combinator called Z. Fix an expression e and define

$$\begin{aligned}Z &:= \lambda x.\, g\, g\, x \\ g &:= \lambda r.\, \mathsf{let}\, f = \lambda x.\, r\, r\, x\, \mathsf{in}\, \lambda x.\, e\end{aligned}$$

Z does not have type in our language, as it can be used to write diverging terms without using recursive types. In this sense, it is similar to the assert instruction (assuming we make sure that the assertion always ends up being true). However, Z is a perfectly safe runtime expression that reduces without getting stuck:

$$\begin{aligned}Z\, v &\rightsquigarrow g\, g\, v \\ &\rightsquigarrow (\mathsf{let}\, f = \lambda x.\, g\, g\, x\, \mathsf{in}\, \lambda x.\, e)\, v \\ &\rightsquigarrow (\lambda x.\, e[Z/f])\, v \\ &\rightsquigarrow e[Z/f, v/x]\end{aligned}$$

The way we will prove Z safe is most peculiar. At one point in the proof, we will arrive at what seems to be a circularity: In the process of proving a certain expression safe, the proof obligation reduces to showing that same expression to be safe. It seems like we made no progress at all. However, the step-indices are not the same. In fact, during the proof, we will take steps that decrease the step index of the final goal. The way out of this conundrum is to simply assume the expression to be safe at all lower step indices, by initiating an induction over the step-index. This technique is known as Löb induction, and later on in the course we will see why it has earned its special name over natural induction.

To harness the full power of Löb induction, we will eventually apply it to expressions that are functions. In these cases, our goal will often be $(k, \lambda x.\, e) \in \mathcal{V}[\![A \to B]\!]$. Our Löb induction hypothesis, however, will be $(k', \lambda x.\, e) \in \mathcal{E}[\![A \to B]\!]$. To make use of the induction hypothesis, we need a way to go from $\mathcal{E}[\![A \to B]\!]$ to $\mathcal{V}[\![A \to B]\!]$. This is a fairly trivial lemma.

Exercise 44 Prove the following lemma:

Lemma 52.
If $(k, \lambda x.\, e) \in \mathcal{E}[\![A \to B]\!]\delta$ *and* $(\lambda x.\, e) \in \mathit{CVal}$,
then $(k, \lambda x.\, e) \in \mathcal{V}[\![A \to B]\!]\delta$*.*

●

Before we get to the safety proof for Z, we first introduce yet another helpful lemma that will make our life easier. It extracts the core of the compatibility lemma for application.

Lemma 53 (Semantic Application)**.** *If* $(k, e_1) \in \mathcal{E}[\![A \to B]\!]\delta$ *and* $(k, e_2) \in \mathcal{E}[\![A]\!]\delta$*, then* $(k, e_1\, e_2) \in \mathcal{E}[\![B]\!]\delta$*.*

Finally, we proceed with the proof of safety of Z.

Lemma 54 (Z is safe)**.**

$$\frac{\Delta\,;\Gamma, f : A \to B, x : A \vDash e : B}{\Delta\,;\Gamma \vDash Z : A \to B}$$

Proof.

We have:	To show:
(i) $\Delta\,;\Gamma, f : A \to B, x : A \vDash e : B$	$\Delta\,;\Gamma \vDash Z : A \to B$
Let $\delta, (k, \gamma) \in \mathcal{G}[\![\Gamma]\!]\delta$	$(k, \gamma\, Z) \in \mathcal{E}[\![A \to B]\!]\delta$
Set $g' := \lambda r.\, \mathsf{let}\, f = \lambda x.\, r\, r\, x\, \mathsf{in}\, \lambda x.\, \gamma\, e$	$(k, \lambda x.\, g'\, g'\, x) \in \mathcal{E}[\![A \to B]\!]\delta$
(ii) $\forall k' < k.\, (k', \lambda x.\, g'\, g'\, x) \in \mathcal{E}[\![A \to B]\!]\delta)$	
by strong induction over k (Löb induction).	
	By value inclusion, $(k, \lambda x.\, g'\, g'\, x) \in \mathcal{V}[\![A \to B]\!]\delta$
Suppose $j \leq k$ and $(j, v_1) \in \mathcal{V}[\![A]\!]\delta$	$(j, g'\, g'\, v_1) \in \mathcal{E}[\![B]\!]\delta$
We apply Lemma 53 and handle the goals in reverse order.	
(2)	$(j, v_1) \in \mathcal{E}[\![A]\!]\delta$
By value inclusion and assumption.	
(1)	$(j, g'\, g') \in \mathcal{E}[\![A \to B]\!]\delta$
Have $g'\, g' \rightsquigarrow \mathsf{let}\, f = \lambda x.\, g'\, g'\, x\, \mathsf{in}\, \lambda x.\, \gamma\, e \rightsquigarrow \lambda x.\, \gamma\, e[\lambda x.\, g'\, g'\, x/f]$.	
	By closure under expansion, $(j - 2, \lambda x.\, \gamma\, e[\lambda x.\, g'\, g'\, x/f]) \in \mathcal{E}[\![A \to B]\!]\delta$
	By value inclusion, $(j - 2, \lambda x.\, \gamma\, e[\lambda x.\, g'\, g'\, x/f]) \in \mathcal{V}[\![A \to B]\!]\delta$
Suppose $i \leq j - 2, (i, v_2) \in \mathcal{V}[\![A]\!]$	$(i, \gamma\, e[\lambda x.\, g'\, g'\, x/f][v_2/x]) \in \mathcal{E}[\![B]\!]\delta$
Set $\gamma' = \gamma[f \mapsto \lambda x.\, g'\, g'\, x, x \mapsto v_2]$	$(i, \gamma' e) \in \mathcal{E}[\![B]\!]\delta$
	By applying (i), $(i, \gamma') \in \mathcal{G}[\![\Gamma, f : A \to B, x : A]\!]$
Suppose $y : C \in \Gamma, f : A \to B, x : A$	$(i, \gamma'(y)) \in \mathcal{V}[\![C]\!]\delta$
Case: $y : C \in \Gamma$	
Have $\gamma'(y) = \gamma(y)$	$(i, \gamma(y)) \in \mathcal{V}[\![C]\!]\delta$
From $(k, \gamma) \in \mathcal{G}[\![\Gamma]\!]\delta$, we have $(k, \gamma(y)) \in \mathcal{V}[\![C]\!]\delta$.	
By monotonicity, we are done.	
Case: $y = x, C = A$	$(i, \gamma'(x)) \in \mathcal{V}[\![A]\!]\delta$
Have $\gamma'(x) = v_2$	$(i, v_2) \in \mathcal{V}[\![A]\!]\delta$
By assumption.	
Case: $y = f, C = A \to B$	$(i, \gamma'(f)) \in \mathcal{V}[\![A \to B]\!]\delta$
	$(i, \lambda x.\, g'\, g'\, x) \in \mathcal{V}[\![A \to B]\!]\delta$
	By Lemma 52, $(i, \lambda x.\, g'\, g'\, x) \in \mathcal{E}[\![A \to B]\!]\delta$
We apply our Löb induction hypothesis (ii), with $k' := i \leq j - 2 < j \leq k$.	

□

Essentially, what this proof demonstrates is that we can just assume our goal of the form $\mathcal{E}[\![A]\!]\delta$ to hold, without any work—but only at smaller step-indices. So before we can use the induction hypothesis, we have to let the program do some computation.

4 Mutable State

In this chapter, we extend the language with references. We add the usual operations on references: allocation, dereferencing, assignment. Interestingly, we do not need to talk about locations (think of them as addresses) in the source terms—just like we usually do not have raw addresses in our code. Only when we define what the allocation operation reduces to, do we need to introduce them. Consequently, they are absent from the source terms and do not have a typing rule in the Church-style typing relation.

$$\begin{array}{rrcl}
\text{Locations} & \ell & & \\
\text{Heaps} & h & \in & Loc \xrightharpoonup{\mathsf{fin}} Val \\
\text{Types} & A, B & ::= & \ldots \mid \mathsf{ref}\, A \\
\text{Source Terms} & E & ::= & \ldots \mid \mathsf{new}\, E \mid\ !\, E \mid E_1 \leftarrow E_2 \\
\text{Runtime Terms} & e & ::= & \ldots \mid \ell \mid \mathsf{new}\, e \mid\ !\, e \mid e_1 \leftarrow e_2 \\
\text{(Runtime) Values} & v & ::= & \ldots \mid \ell \\
\text{Evaluation Contexts} & K & ::= & \ldots \mid \mathsf{new}\, K \mid\ !\, K \mid e \leftarrow K \mid K \leftarrow v
\end{array}$$

Contextual operational semantics We need to extend our reduction relations with heaps (also called stores in the literature), which are finite partial functions from locations to values tracking allocated locations and their contents. We use $\emptyset$ to denote the empty heap. Most base reduction rules lift to the new judgment in the expected way: They work for any heap, and do not change it; for example, the rule for β-reduction now reads

$$h\,;(\lambda x.\, e)\, v \leadsto_{\mathrm{b}} h\,; e[v/x]$$

The base reduction rules for allocation, dereference, and assignment, however, interact with the heap:

Base reduction $\boxed{h_1\,; e_1 \leadsto_{\mathrm{b}} h_2\,; e_2}$

$$\ldots \qquad \frac{\ell \notin \mathsf{dom}(h)}{h\,; \mathsf{new}\, v \leadsto_{\mathrm{b}} h[\ell \mapsto v]\,; \ell}\,\textsc{New} \qquad \frac{h(\ell) = v}{h\,;!\,\ell \leadsto_{\mathrm{b}} h\,; v}\,\textsc{Deref} \qquad \frac{\ell \in \mathsf{dom}(h)}{h\,; \ell \leftarrow v \leadsto_{\mathrm{b}} h[\ell \mapsto v]\,; ()}\,\textsc{Assign}$$

Reduction $\boxed{h_1\,; e_1 \leadsto h_2\,; e_2}$

$$\frac{h\,; e \leadsto_{\mathrm{b}} h'\,; e'}{h\,; K[e] \leadsto h'\,; K[e']}\,\textsc{Ctx}$$

Notice that if we say $h(\ell) = v$, this implicitly also asserts that $\ell \in \mathsf{dom}(h)$. Furthermore, observe that NEW is our first non-deterministic reduction rule: There are many (in fact, infinitely many) possible choices for ℓ.

Church-style typing $\boxed{\Delta\,; \Gamma \vdash E : A}$

$$\ldots \qquad \frac{\Delta\,; \Gamma \vdash E : A}{\Delta\,; \Gamma \vdash \mathsf{new}\, E : \mathsf{ref}\, A}\,\textsc{New} \qquad \frac{\Delta\,; \Gamma \vdash E : \mathsf{ref}\, A}{\Delta\,; \Gamma \vdash !\, E : A}\,\textsc{Deref} \qquad \frac{\Delta\,; \Gamma \vdash E_1 : \mathsf{ref}\, A \qquad \Delta\,; \Gamma \vdash E_2 : A}{\Delta\,; \Gamma \vdash E_1 \leftarrow E_2 : \mathbf{1}}\,\textsc{Assign}$$

The typing rules for source terms are straight-forward, as locations do not arise in the source language.

To type runtime terms, we extend the typing rules with a heap typing context

$$\text{Heap Typing} \quad \Sigma ::= \emptyset \mid \Sigma, \ell : \mathsf{ref}\, A$$

tracking the types of locations. (The other rules just carry Σ around, but are otherwise unchanged.)

Curry-style typing $\boxed{\Sigma ; \Delta ; \Gamma \vdash e : A}$

$$\dots \qquad \frac{\Sigma ; \Delta ; \Gamma \vdash e : A}{\Sigma ; \Delta ; \Gamma \vdash \mathsf{new}\, e : \mathsf{ref}\, A}\,\text{NEW} \qquad \frac{\Sigma ; \Delta ; \Gamma \vdash e_1 : \mathsf{ref}\, A \qquad \Sigma ; \Delta ; \Gamma \vdash e_2 : A}{\Sigma ; \Delta ; \Gamma \vdash e_1 \leftarrow e_2 : \mathbf{1}}\,\text{ASSIGN}$$

$$\frac{\Sigma ; \Delta ; \Gamma \vdash e : \mathsf{ref}\, A}{\Sigma ; \Delta ; \Gamma \vdash\, !\, e : A}\,\text{DEREF} \qquad \frac{\ell : \mathsf{ref}\, A \in \Sigma}{\Sigma ; \Delta ; \Gamma \vdash \ell : \mathsf{ref}\, A}\,\text{LOC}$$

4.1 Examples

Counter. Consider the following program, which uses references and local variables to effectively hide the implementation details of a counter. This also goes to show that even values with a type that does not even mention references, like $\mathbf{1} \to \mathsf{int}$, can now have external behavior that was impossible to produce previously—namely, the function returns a different value on each invocation. Finally, the care we took previously to define the left-to-right evaluation order using evaluation contexts now really pays off: With a heap, the order in which expressions are reduced *does* matter.

$$\begin{aligned} p := \quad & \mathsf{let}\,\mathrm{cnt} = \mathsf{let}\, x = \mathsf{new}\, \overline{0}\, \mathsf{in} \\ & \qquad\qquad \lambda y.\, x \leftarrow !\, x + 1 \,;\, !\, x \\ & \mathsf{in} \\ & \mathrm{cnt}\,() + \mathrm{cnt}\,() \end{aligned}$$

To illustrate the operational semantics of the newly introduced operations, we investigate the execution of this program under an arbitrary heap h:

$$\begin{aligned} h ; p &\leadsto^* h[\ell \mapsto \overline{0}] ; (\lambda y.\, \ell \leftarrow !\, \ell + 1 ; !\, \ell)\,() + (\lambda y.\, \ell \leftarrow !\, \ell + 1 ; !\, \ell)\,() \qquad \ell \notin \mathsf{dom}(h) \\ &\leadsto^* h[\ell \mapsto \overline{0}] ; (\lambda y.\, \ell \leftarrow !\, \ell + 1 ; !\, \ell)\,() + (\ell \leftarrow 1 ; !\, \ell) \\ &\leadsto^* h[\ell \mapsto \overline{1}] ; (\ell \leftarrow 2 ; !\, \ell) + (\overline{1}) \\ &\leadsto^* h[\ell \mapsto \overline{2}] ; (!\, \ell) + \overline{1} \\ &\leadsto^* h[\ell \mapsto \overline{2}] ; \overline{2} + \overline{1} \\ &\leadsto h[\ell \mapsto \overline{2}] ; \overline{3} \end{aligned}$$

Exercise 45 We are (roughly) translating the following Java class into our language:

```
class Stack<T> {
  private ArrayList<T> l;
  public Stack() { this.l = new ArrayList<T>(); }
  public void push(T t) { l.add(t); }
  public T pop() {
    if l.isEmpty() { return null; } else { return l.remove(l.size() - 1) }
  }
}
```

To do so, we first translate the interface provided by the class (*i.e.*, everything public that is provided) into an existential type:

$$\begin{aligned} \mathrm{STACK}(A) := \exists \beta.\, \{&\mathsf{new} : () \to \beta, \\ &\mathsf{push} : \beta \to A \to (), \\ &\mathsf{pop} : \beta \to \mathbf{1} + A\} \end{aligned}$$

Just like the Java class, we are going to implement this interface with lists, but we will hide that fact and make sure clients can only use that list in a stack-like way. We assume that a type $\mathsf{list}\, A$ of the usual, functional lists with nil, cons and $\mathsf{listcase}$ is provided.

Define MyStack(A) such that the following term is well-typed of type $\forall \alpha.\, \mathrm{STACK}(\alpha)$, and behaves like the Java class (*i.e.*, like an imperative stack).

$$\Lambda \alpha.\, \mathsf{pack}\, [\mathsf{ref}\, \mathsf{list}\, \alpha, \mathrm{MyStack}(\alpha)]\, \mathsf{as}\, \mathrm{STACK}(\alpha)$$

(Of course, you do not have to take into account implementation details of the above Java class.) ●

Exercise 46 (Obfuscated Code) With references, our language now has a new feature: Obfuscated code!

Execute the following program in the empty heap, and give its result. You do not have to write down every single reduction step, but make sure the overall execution behavior is clear.

$$\begin{aligned} E := {} & \mathsf{let}\, x = \mathsf{new}\, (\lambda x : \mathsf{int}.\, x + x)\, \mathsf{in} \\ & \mathsf{let}\, f = (\lambda g : \mathsf{int} \to \mathsf{int}.\, \mathsf{let}\, f = {!}\, x\, \mathsf{in}\, x \leftarrow g\, ;\, f\, \overline{11})\, \mathsf{in} \\ & f\, (\lambda x : \mathsf{int}.\, f\, (\lambda y : \mathsf{int}.\, x)) + f\, (\lambda x : \mathsf{int}.\, x + \overline{9}) \end{aligned}$$

●

Exercise 47 (Not a big Challenge) Using references, it is possible to write a (syntactically) well-typed closed term that does not use roll or unroll, and that diverges. Find such a term. ●

4.2 Recursion via state

In the last chapter we have seen how to get recursion using recursive types. With state there is, however, another way to define recursive functions. In the simply typed lambda calculus and in system F recursion was not possible because for a recursive call a function needed to be in its own context. The only way to achieve this is by passing the function as an argument to itself. This could, however, not be done in a type-safe way. Recursive types solved this problem by allowing to roll the type of a function into some recursive type variable α which could then be taken as an argument.

With state there is another option: We do not need to have the function in the context at all, we just need to be able to access it via a reference. So there is no typing issue. This does of course leave open the question of how we are going to use a reference storing our recursive function in the recursive function we are currently trying to define. It turns out there is a trick: First store a dummy function in the reference. Once we have defined the function we can update the reference with the function we actually want to use. This way we can define a function recursively computing the sum from 0 to the function argument n like this:

$$\begin{aligned}&\mathsf{let}\ x = \mathsf{new}\ \lambda x.\ \overline{0}\ \mathsf{in}\\ &\mathsf{let}\ f = \lambda n.\ \mathsf{if}\ n = \overline{0}\ \mathsf{then}\ \overline{0}\ \mathsf{else}\ n + (!\,x)(n + (\overline{-1}))\ \mathsf{in}\\ &x \leftarrow f\ ;f\end{aligned}$$

This works well, but it is quite annoying to have to do the reference handling manually every time. Luckily it can be factored out into a separate function. This approach to recursion is know as *Landin's knot*.

$$\begin{aligned}\mathit{knot} :=\, &\lambda f.\\ &\quad \mathsf{let}\ x = \mathsf{new}\ \lambda x.\ \overline{0}\ \mathsf{in}\\ &\quad \mathsf{let}\ g = f\ (\lambda y.\ (!\,x)\ y)\ \mathsf{in}\\ &\quad x \leftarrow g\ ;g\end{aligned}$$

We can now define the sum function from above using *knot*.

$$\mathit{knot}\ \lambda f n.\ \mathsf{if}\ n = \overline{0}\ \mathsf{then}\ \overline{0}\ \mathsf{else}\ n + f\ (n + (\overline{-1}))$$

Note that in the definition of *knot* we are using $\lambda y.\ (!\,x)\ y$. An obvious question is whether we can just use $!\,x$ instead, as this is already a function of the same type. Our intuition from pure functional programming would suggest, that it should indeed be irrelevant whether we use a function or its η-expansion. This is, however, not true in the presence of side-effects. If we just used $!\,x$ the dereference would be executed immediately and we would look up the dummy function before it has been replaced by the actual function. The Λ-abstractions is a guard deferring the lookup to the point when the function is being used. At that point the dummy function will have been replaced by the correct function.

One might then ask why we do not guard the access to x by a λ in our first program. The reason is that the access is already guarded by the outer λ in the definition of f.

Exercise 48 Write a function computing the n-th fibonacci number. Do not use recursive types or church numerals. •

Exercise 49 In the lecture you have seen how to implement a simple counter. Give a definition of a counter that can be reset to 0 using a function $\mathit{reset} : \mathbf{1} \to \mathbf{1}$ •

4.3 Type Safety

We need to do a little work to extend our proof of syntactic type safety for state.

Our contextual typing judgment must now track the types of locations:

Contextual Typing $\boxed{\Sigma \vdash K : A \Rightarrow B}$

$$\Sigma \vdash K : A \Rightarrow B := \forall e, \Sigma'.\, \Sigma \subseteq \Sigma' \Rightarrow \Sigma' \vdash e : A \;\Rightarrow\; \Sigma' \vdash K[e] : B$$

We can now reprove composition and decomposition, with heap contexts.

Lemma 55 (Composition).
If $\Sigma\,;\emptyset\,;\emptyset \vdash e : B$ and $\Sigma \vdash K : B \Rightarrow A$, then $\Sigma\,;\emptyset\,;\emptyset \vdash K[e] : A$.

Lemma 56 (Decomposition).
If $\Sigma\,;\emptyset\,;\emptyset \vdash K[e] : A$, then $\Sigma\,;\emptyset\,;\emptyset \vdash e : B$ and $\Sigma \vdash K : B \Rightarrow A$ for some B.

Our proof of preservation will require two weakening lemmas that allow us to consider larger heap contexts.

Lemma 57 (Σ-Weakening). *If $\Sigma\,;\Delta\,;\Gamma \vdash e : A$ and $\Sigma' \supseteq \Sigma$, then $\Sigma'\,;\Delta\,;\Gamma \vdash e : A$.*

Lemma 58 (Contextual Σ-Weakening).
If $\Sigma \vdash K : A \Rightarrow B$ and $\Sigma' \supseteq \Sigma$, then $\Sigma' \vdash K : A \Rightarrow B$.

We need a significant change for progress and preservation to go through. Let's begin by adding heaps and heap contexts to our original formulation of preservation:

If $\Sigma\,;\emptyset\,;\emptyset \vdash e : A$ and $h\,;e \leadsto h'\,;e'$,
then $\Sigma\,;\emptyset\,;\emptyset \vdash e' : A$.

Notice that the heap context Σ does not change: this formulation does not account for allocation! The fix is to show that e' is well-typed against some potentially larger heap context Σ':

If $\Sigma\,;\emptyset\,;\emptyset \vdash e : A$ and $h\,;e \leadsto h'\,;e'$,
then there exists $\Sigma' \supseteq \Sigma$ s.t. $\Sigma'\,;\emptyset\,;\emptyset \vdash e' : A$.

A problem remains. Due to *dereference,* we have to ensure that *existing* locations remain in the heap typing, and keep their type. If we had a judgment $h : \Sigma$ that somehow ties the values in heaps to the types in heap contexts, we could formulate preservation as

If $\Sigma\,;\emptyset\,;\emptyset \vdash e : A$ and $h : \Sigma$ and $h\,;e \leadsto h'\,;e'$,
then there exists $\Sigma' \supseteq \Sigma$ s.t. $\Sigma'\,;\emptyset\,;\emptyset \vdash e' : A$ and $h' : \Sigma'$.

and our proof would go through. So let's define this heap typing judgment.

Heap Typing $\boxed{h : \Sigma}$

$$h : \Sigma := \forall \ell : \mathsf{ref}\, A \in \Sigma.\, \Sigma\,;\emptyset\,;\emptyset \vdash h(\ell) : A$$

Notice that the values stored in the heap, can use the *entire* heap to justify their well-typedness. In particular, the value stored at some location ℓ can itself refer to ℓ, since it is type-checked in a heap typing that contains ℓ.

To reformulate progress, we assume that the initial heap is well-typed. Additionally, we now (of course) quantify existentially over the heap that we end up with after taking a step (just like we quantify existentially over the expression we reduce to):

Lemma 59 (Progress)**.**
If $\Sigma\,;\emptyset\,;\emptyset \vdash e : A$ *and* $h : \Sigma$,
then e *is a value or there exist* e', h' *s.t.* $h\,; e \rightsquigarrow h'\,; e'$.

Proof. By induction on the typing derivation of e. Existing cases remain unchanged.

Case 1: $e = \ell$. ℓ is a value.

Case 2: $e = \mathsf{new}\, e'$ and $A = \mathsf{ref}\, B$ and $\Sigma\,;\emptyset\,;\emptyset \vdash e' : B$. By induction, we have

Subcase 1: $h\,; e' \rightsquigarrow h'\,; e''$.
$h\,; e'_1 \rightsquigarrow_{\mathrm{b}} h'\,; e''_1$ and $e' = K[e'_1]$ and $e'' = K[e''_1]$ by inversion.
Thus $e = (\mathsf{new}\, K)[e'_1]$ and we have $h\,; e \rightsquigarrow h'\,; (\mathsf{new}\, K)[e''_1]$ by CTX.

Subcase 2: e' is a value.
As h is finite, we may pick $\ell \notin \mathsf{dom}(h)$.
Thus $h\,; \mathsf{new}\, e' \rightsquigarrow h[\ell \mapsto e']\,; \ell$ by NEW, CTX.

Case 3: $e = \,!\, e'$ and $\Sigma\,;\emptyset\,;\emptyset \vdash e' : \mathsf{ref}\, A$. By induction, we have

Subcase 1: $h\,; e' \rightsquigarrow h'\,; e''$.
$h\,; e'_1 \rightsquigarrow_{\mathrm{b}} h'\,; e''_1$ and $e' = K[e'_1]$ and $e'' = K[e''_1]$ by inversion.
Thus $e = (!\, K)[e'_1]$ and we have $h\,; e \rightsquigarrow h'\,; (!\, K)[e''_1]$ by CTX.

Subcase 2: e' is a value.
$e' = \ell$ and $\ell : \mathsf{ref}\, A \in \Sigma$ by inversion (or canonical forms) with $\Sigma\,;\emptyset\,;\emptyset \vdash e' : \mathsf{ref}\, A$.
$\ell \in \mathsf{dom}(h)$ by $h : \Sigma$.
Thus $h\,; !\, e' \rightsquigarrow h\,; h(\ell)$ by DEREF, CTX.

Case 4: $e = e_1 \leftarrow e_2$ and $\Sigma\,;\emptyset\,;\emptyset \vdash e_1 : \mathsf{ref}\, B$ and $\Sigma\,;\emptyset\,;\emptyset \vdash e_2 : B$. By induction, we have

Subcase 1: $h\,; e_2 \rightsquigarrow h'\,; e'_2$.
$h\,; e_3 \rightsquigarrow_{\mathrm{b}} h'\,; e'_3$ and $e_2 = K[e_3]$ and $e'_2 = K[e'_3]$ by inversion.
Thus $e = (e_1 \leftarrow K)[e_3]$ and we have $h\,; e \rightsquigarrow h'\,; (e_1 \leftarrow K)[e'_3]$ by CTX.

Subcase 2: e_2 is a value and $h\,; e_1 \rightsquigarrow h'\,; e'_1$.
$h\,; e_3 \rightsquigarrow_{\mathrm{b}} h'\,; e'_3$ and $e_1 = K[e_3]$ and $e'_1 = K[e'_3]$ by inversion.
Thus $e = (K \leftarrow e_2)[e_3]$ and we have $h\,; e \rightsquigarrow h'\,; (K \leftarrow e_2)[e'_3]$ by CTX.

Subcase 3: e_1 and e_2 are values.
$e_1 = \ell$ and $\ell : \mathsf{ref}\, B \in \Sigma$ by inversion (or canonical forms) with $\Sigma\,;\emptyset\,;\emptyset \vdash e_1 : \mathsf{ref}\, B$.
$\ell \in \mathsf{dom}(h)$ by $h : \Sigma$.
Thus $h\,; e \rightsquigarrow h[\ell \mapsto e_2]\,; ()$ by ASSIGN, CTX. □

Lemma 60 (Base preservation)**.**
If $\Sigma\,;\emptyset\,;\emptyset \vdash e : A$ *and* $h : \Sigma$ *and* $h\,; e \rightsquigarrow_{\mathrm{b}} h'\,; e'$,
then there exists $\Sigma' \supseteq \Sigma$ *s.t.* $h' : \Sigma'$ *and* $\Sigma'\,;\emptyset\,;\emptyset \vdash e' : A$.

Proof. By cases on on the reduction of $h\,;e$. Existing cases remain mostly unchanged. (We have $h' = h$ and pick $\Sigma' = \Sigma$.) We leave base preservation for the new reduction rules as an exercise. □

Exercise 50 Prove base preservation for new, $!$, and $\leftarrow$. •

Lemma 61 (Preservation).
If $\Sigma\,;\emptyset\,;\emptyset \vdash e : A$ *and* $h : \Sigma$ *and* $h\,; e \rightsquigarrow h'\,; e'$,
then there exists $\Sigma' \supseteq \Sigma$ *s.t.* $h' : \Sigma'$ *and* $\Sigma'\,;\emptyset\,;\emptyset \vdash e' : A$.

Proof.

We have:	To show:
$\Sigma\,;\emptyset\,;\emptyset \vdash e : A,\ h : \Sigma,\ h\,; e \rightsquigarrow h'\,; e'$	$\exists \Sigma' \supseteq \Sigma.\, h' : \Sigma' \wedge \Sigma'\,;\emptyset\,;\emptyset \vdash e' : A$
$h\,; e_1 \rightsquigarrow_{\mathrm{b}} h'\,; e_1'$ and $e = K[e_1]$ and $e' = K[e_1']$ by inversion	
$\Sigma\,;\emptyset\,;\emptyset \vdash e_1 : B$ and $\Sigma \vdash K : B \Rightarrow A$ by decomposition	
$\Sigma' \supseteq \Sigma$ and $h' : \Sigma'$ and $\Sigma'\,;\emptyset\,;\emptyset \vdash e_1' : B$ by base preservation	
Pick Σ'	$\Sigma'\,;\emptyset\,;\emptyset \vdash K[e_1'] : A$
	by composition, $\Sigma' \vdash K : B \Rightarrow A$
We're done by weakening.	

□

4.4 Weak Polymorphism and the Value Restriction

There is a problem with the combination of implicit polymorphism (as it is implemented in ML) and references. Consider the following example program (in SML syntax).

$$\begin{array}{rr}
\mathsf{let\ val}\ x = \mathsf{ref\ nil} & x : \forall\alpha.\ \alpha\ \mathsf{list\ ref} \\
\mathsf{in}\ x := [5,6]; & x : \mathsf{int\ list\ ref} \\
(\mathsf{hd}\ (!x))\ (7) & x : (\mathsf{int} \to \mathsf{int})\ \mathsf{list\ ref}
\end{array}$$

The initial typing for x is due to implicit let polymorphism. The following typings are valid instantiations of that type. However, the program clearly should not be well-typed, as the last line will call an integer as a function.

The initial response to this problem is a field of research called weak polymorphism. We do not discuss these endeavours here. Instead, we want to mention a practical solution to the problem given by Andrew Wright in 1995 in a paper titled "Simple Impredicative Polymorphism". The solution is called the *value restriction* as it restricts implicit let polymorphism to values. To see why this solves the problem, consider the translation of the example above into System F with references.

$$\begin{array}{rr}
\mathsf{let}\ x = \Lambda.\ \mathsf{ref\ nil} & x : \forall\alpha.\ \mathsf{ref}\ (\mathsf{list}\ \alpha) \\
\mathsf{in}\ x\ \langle\rangle \leftarrow [5,6]; & x\ \langle\rangle : \mathsf{ref}\ (\mathsf{list\ int}) \\
(\mathsf{hd}\ (!(x\ \langle\rangle)))\ (7) & x\ \langle\rangle : \mathsf{ref}\ (\mathsf{list\ int} \to \mathsf{int})
\end{array}$$

We can see now why the original program could be considered well-typed. Note that the semantics are different from what the initial program's semantics seemed to be. In particular, x is a thunk in the System F version, so every instantiation generates a new, distinct, empty list.

In the case of values, however, the thunk will always evaluate to the same value. The "intended" semantics of the original program and the semantics of the translation coincide, so let polymorphism can be soundly applied.

4.5 Data Abstraction via Local State

References, specifically local references, give us yet another way of ensuring data abstraction. Consider the following signature for a mutable boolean value and corresponding implementation.

$$\begin{aligned}
\text{MUTBIT} &:= \{\text{flip} : \mathbf{1} \to \mathbf{1},\ \text{get} : \mathbf{1} \to \mathsf{bool}\}\\
\text{MyMutBit} &:= \mathsf{let}\, x = \mathsf{new}\,\overline{0}\\
&\qquad\quad \mathsf{in}\,\{\text{flip} := \lambda y.\, x \leftarrow \overline{1} - !\,x,\\
&\qquad\qquad\ \ \text{get} := \lambda y.\ !\,x > 0\}
\end{aligned}$$

As with previous examples, we would like to maintain an invariant on the value of x, namely that its content is either 0 or 1. The abstraction provided by the local reference will guarantee that no client code can violate this invariant. As before, we will extend the implementation with corresponding assert statements to ensure that the program *crashes* if the invariant is violated. As a consequence, by proving the program crash-free, we showed that the invariant is maintained.

$$\begin{aligned}
\text{MyMutBit} &:= \mathsf{let}\, x = \mathsf{new}\,\overline{0}\\
&\qquad\quad \mathsf{in}\,\{\text{flip} := \lambda y.\, \mathsf{assert}\,(!\,x == 0 \lor !\,x == 1)\,;\, x \leftarrow \overline{1} - !\,x,\\
&\qquad\qquad\ \ \text{get} := \lambda y.\, \mathsf{assert}\,(!\,x == 0 \lor !\,x == 1)\,;\, !\,x > 0\}
\end{aligned}$$

Once we extend our semantic model to handle references, we will be able to prove that this code is semantically well-typed, *i.e.*, does not crash—and as a consequence, we know that the assertions will always hold true.

4.6 Semantic model

We extend our previous semantic model with a notion of "possible worlds". These worlds are meant to encode the possible shapes of the physical state, *i.e.*, the heap that our program will produce over the course of its execution. When we allocate fresh references, we are allowed to add additional invariants that will be preserved in the remainder of the execution. This is the key reasoning principle that justifies the example from the previous section: We can have invariants on parts of the heap, like on the location used for x above.

$$\begin{array}{rrcl}
\text{Invariants} & \mathit{Inv} & := & \mathbb{P}(\text{Heap})\\
\text{World} & W & \in & \bigcup_n \mathit{Inv}^n\\
\text{World Extension} & W' \sqsupseteq W & := & \exists n.\, |W'| \geq |W| \land |W| = n \land \forall i \in 1 \ldots n.\, W'[i] = W[i]\\
\text{World Satisfaction} & h : W & := & \exists n.\, |W| = n \land \exists h_1 \ldots h_n.\, h \supseteq h_1 \uplus \ldots \uplus h_n \land\\
& & & \forall i \in 1 \ldots n.\, h_i \in W[i]
\end{array}$$

We update our step-indexed termination judgment for state.

Step-indexed termination $\boxed{h\,;e \searrow^k h'\,;e'}$

$$\frac{\forall h', e'.\, h\,;e \not\leadsto h'\,;e'}{h\,;e \searrow^0 h\,;e} \qquad \frac{h\,;e \leadsto h'\,;e' \qquad h'\,;e' \searrow^k h''\,;e''}{h\,;e \searrow^{k+1} h''\,;e''}$$

Semantic Types $\boxed{\tau \in \mathit{SemType}}$

$$\mathit{SemType} := \left\{ \tau \in \mathbb{P}(\mathbb{N} \times \text{World} \times \mathit{CVal}) \;\middle|\; \begin{array}{l} \forall (k, W, v) \in \tau.\, \forall k' \leq k, W' \sqsupseteq W. \\ \quad (k', W', v) \in \tau \end{array} \right\}$$

Notice that semantic types have to be closed with respect to *smaller* step-indices and *larger* worlds.

First-Order References Before we get to the meat of the model, the value relation, we have to impose an important restriction on our language. From here on, our language has only first-order references. Put differently, the heap is not allowed to contain functions, polymorphic or existential types, or references. If the heap contains these higher-order types, the semantic model presented below will not work.

First-Order Types $\boxed{a}$

$$\begin{array}{rrcl} \text{First-order Types} & a & ::= & \mathsf{int} \mid \mathsf{bool} \mid a_1 + a_2 \mid a_1 \times a_2 \\ \text{Types} & A & ::= & \dots \mid \mathsf{ref}\, a \end{array}$$

Value Relation $\boxed{\mathcal{V}[\![A]\!]\delta}$

$$\begin{aligned}
\mathcal{V}[\![\alpha]\!]\delta &:= \delta(\alpha) \\
\mathcal{V}[\![\mathsf{int}]\!]\delta &:= \{(k, W, \overline{n})\} \\
\mathcal{V}[\![A \times B]\!]\delta &:= \{(k, W, \langle v_1, v_2 \rangle) \mid (k, W, v_1) \in \mathcal{V}[\![A]\!]\delta \wedge (k, W, v_2) \in \mathcal{V}[\![B]\!]\delta\} \\
\mathcal{V}[\![A \to B]\!]\delta &:= \{(k, W, \lambda x.\, e) \;\mid\; (\lambda x.\, e) \in \mathit{CVal} \wedge \forall j \leq k, W' \sqsupseteq W, v. \\
&\qquad\qquad (j, W', v) \in \mathcal{V}[\![A]\!]\delta \Rightarrow (j, W', e[v/x]) \in \mathcal{E}[\![B]\!]\delta\} \\
\mathcal{V}[\![\mathsf{ref}\, a]\!]\delta &:= \{(k, W, \ell) \mid \exists i.\, W[i] = \{[\ell \mapsto v] \mid \vdash v : a\}\} \\
\mathcal{V}[\![\forall \alpha.\, A]\!]\delta &:= \{(k, W, \Lambda.\, e) \mid (\Lambda.\, e) \in \mathit{CVal} \wedge \forall \tau \in \mathit{SemType}.\, (k, W, e) \in \mathcal{E}[\![A]\!](\delta, \alpha \mapsto \tau)\} \\
\mathcal{V}[\![\exists \alpha.\, A]\!]\delta &:= \{(k, W, \mathsf{pack}\, v) \mid \exists \tau \in \mathit{SemType}.\, (k, W, v) \in \mathcal{V}[\![A]\!](\delta, \alpha \mapsto \tau)\} \\
\mathcal{V}[\![\mu \alpha.\, A]\!]\delta &:= \{(k, W, \mathsf{roll}\, v) \mid \forall j < k.\, (j, W, v) \in \mathcal{V}[\![A[\mu\alpha.\, A/\alpha]]\!]\delta\}
\end{aligned}$$

Expression Relation $\boxed{\mathcal{E}[\![A]\!]\delta}$

$$\begin{aligned}
\mathcal{E}[\![A]\!]\delta := \{(k, W, e) \;\mid\; & \forall j < k, W' \sqsupseteq W, e', h : W', h'. \\
& h\, ;\, e \searrow^{j} h'\, ;\, e' \Rightarrow \exists W'' \sqsupseteq W'.\, h' : W'' \wedge (k - j, W'', e') \in \mathcal{V}[\![A]\!]\delta\}
\end{aligned}$$

The definition of the ref case might seem odd at first glance. More concretely, one might ask why we refer to the syntactic typing judgment. The following lemma explains why this particular definition makes sense.

Lemma 62 (First-order types are simple)**.** $\vdash v : a \Leftrightarrow (k, W, v) \in \mathcal{V}[\![a]\!]\delta$.

In other words, for first-order types, syntactic and semantic well-typedness coincide. Furthermore, the step-index and the worlds are irrelevant.

Example Recall our implementation of the MUTBIT signature:

$$\begin{aligned}\text{MyMutBit} := {} & \mathsf{let}\, x = \mathsf{new}\,\overline{0} \\ & \mathsf{in}\, \{\text{flip} := \lambda y.\, \mathsf{assert}\,(!\,x == 0 \vee !\,x == 1)\,;\, x \leftarrow \overline{1} - !\,x, \\ & \quad \text{get} := \lambda y.\, \mathsf{assert}\,(!\,x == 0 \vee !\,x == 1)\,;\, !\,x > 0\}\end{aligned}$$

We will now prove that this implementation is safe with respect to the signature.

Theorem 63.

$$\forall k, W.\, (k, W, \text{MyMutBit}) \in \mathcal{E}[\![\text{MUTBIT}]\!].$$

Proof.

$$\begin{aligned}\text{Let } e_0(\ell) = {} & \{\text{flip} := \lambda y.\, \mathsf{assert}\,(!\,\ell == 0 \vee !\,\ell == 1)\,;\, \ell \leftarrow \overline{1} - !\,\ell, \\ & \text{get} := \lambda y.\, \mathsf{assert}\,(!\,\ell == 0 \vee !\,\ell == 1)\,;\, !\,\ell > 0\}.\end{aligned}$$

We have: To show:

$$(k, W, \text{MyMutBit}) \in \mathcal{E}[\![\text{MUTBIT}]\!]$$

Suppose $j < k$, $W_1 \sqsupseteq W$, $h : W_1$, $h \,;\, \text{MyMutBit} \searrow^{j} h' \,;\, e'$.

$$\exists W_2 \sqsupseteq W_1.\, h' : W_2 \wedge (k - j, W_2, e') \in \mathcal{V}[\![\text{MUTBIT}]\!]$$

We have $j = 2$, $h' = h[\ell \mapsto \overline{0}]$ (for some $\ell \notin \mathsf{dom}(h)$), and $e' = e_0(\ell)$.

$$\exists W_2 \sqsupseteq W_1.\, h' : W_2 \wedge (k - 2, W_2, e_0(\ell)) \in \mathcal{V}[\![\text{MUTBIT}]\!]$$

Let $n := |W_1|$. Pick $W_2 := W_1 +\!\!+ \left\{[\ell \mapsto \overline{0}],\ [\ell \mapsto \overline{1}]\right\}$.

$$W_2 \sqsupseteq W_1$$

Trivial.

$$h' : W'$$

From $h : W_1$ we have $h = h_1 \uplus \ldots \uplus h_n$.

Since $\ell \notin \mathsf{dom}(h)$, we have $\forall i \in 1 \ldots n.\, \ell \notin \mathsf{dom}(h_i)$.

Thus, $h' = h \uplus h_{n+1} \supseteq h_1 \uplus \ldots \uplus h_n \uplus h_{n+1}$ where $h_{n+1} := [\ell \mapsto \overline{0}]$.

$$\forall i \in 1 \ldots |W_2|.\, h_i \in W_2[i]$$

With $h : W_1$

$$h_{n+1} \in W_2[n+1]$$

$$[\ell \mapsto \overline{0}] \in \left\{[\ell \mapsto \overline{0}],\ [\ell \mapsto \overline{1}]\right\}$$

Trivial.

$$(k - 2, W_2, e_0(\ell)) \in \mathcal{V}[\![(\mathbf{1} \to \mathbf{1}) \times (\mathbf{1} \to \mathsf{bool})]\!]$$

We only do the case for flip here.

$$(k - 2, W_2, \lambda y.\, \mathsf{assert}\ (!\,\ell == 0 \vee !\,\ell == 1)\,;\, \ell \leftarrow \overline{1} - !\,\ell) \in \mathcal{V}[\![\mathbf{1} \to \mathbf{1}]\!]$$

Suppose $j \leq k - 2$, and $W_3 \sqsupseteq W_2$.

$$(j, W_3, \mathsf{assert}\ (!\,\ell == 0 \vee !\,\ell == 1)\,;\, \ell \leftarrow \overline{1} - !\,\ell) \in \mathcal{E}[\![\mathbf{1}]\!]$$

Suppose $j'' < j$, $W_4 \sqsupseteq W_3$, $h'' : W_4$, and $h'' \,;\, e'' \searrow^{j''} h''' \,;\, e'''$.

$$\exists W_5 \sqsupseteq W_4.\, h''' : W_5 \wedge (j - j'', W_5, e''') \in \mathcal{V}[\![\mathbf{1}]\!]$$

From $h'' : W_4$, $h'' \supseteq h_1 \uplus \ldots \uplus h_n \uplus h_{n+1} \uplus h_{n+2} \uplus \ldots h_{n'}$
with $\forall i \in 1 \ldots n'.\, h_i \in W_4[i]$.

Since $W_4 \sqsupseteq W_3 \sqsupseteq W_2$ we have $h_{n+1} = [\ell \mapsto \overline{0}]$ or $h_{n+1} = [\ell \mapsto \overline{1}]$.

Thus, $h''(\ell) = \overline{0}$ or $h''(\ell) = \overline{1}$.

So $h''' = h''[\ell \mapsto \overline{1 - h''(\ell)}]$, $e''' = ()$.

$$\exists W_5 \sqsupseteq W_4.\, h''' : W_5 \wedge (j - j'', W_5, ()) \in \mathcal{V}[\![\mathbf{1}]\!]$$

We pick $W_5 = W_4 \sqsupseteq W_4$.

$$h''' : W_5 \wedge (j - j'', W_5, ()) \in \mathcal{V}[\![\mathbf{1}]\!]$$

$$(j - j'', W_5, ()) \in \mathcal{V}[\![\mathbf{1}]\!]$$

Trivial.

$$h''' : W_5$$

From $h'' : W_4$, it suffices to show $[\ell \mapsto \overline{1 - h''(\ell)}] \in W_4[n+1]$

$$[\ell \mapsto \overline{1 - h''(\ell)}] \in \left\{[\ell \mapsto \overline{0}],\ [\ell \mapsto \overline{1}]\right\}$$

This follows from $h''(\ell) = \overline{0}$ or $h''(\ell) = \overline{1}$.

□

As before, the expression relation $\mathcal{E}[\![A]\!]\delta$ and value relation $\mathcal{V}[\![A]\!]\delta$ are monotone (downwards-closed) with respect to step-indices:

- If $(k, W, v) \in \mathcal{V}[\![A]\!]\delta$, then $\forall j \leq k.\, (j, W, v) \in \mathcal{V}[\![A]\!]\delta$.
- If $(k, W, e) \in \mathcal{E}[\![A]\!]\delta$, then $\forall j \leq k.\, (j, W, e) \in \mathcal{E}[\![A]\!]\delta$.

In addition, they are monotone (upwards-closed) with respect to worlds:

- If $(k, W, v) \in \mathcal{V}[\![A]\!]\delta$, then $\forall W' \sqsupseteq W.\, (k, W', v) \in \mathcal{V}[\![A]\!]\delta$.
- If $(k, W, e) \in \mathcal{E}[\![A]\!]\delta$, then $\forall W' \sqsupseteq W.\, (k, W', e) \in \mathcal{E}[\![A]\!]\delta$.

Lemma 64 (Decomposition of step-indexed termination).
If $h\,;K[e] \searrow^{k} h'\,;e'$,
then $\exists j \leq k, e'', h''.\, h\,;e \searrow^{j} h''\,;e'' \wedge h''\,;K[e''] \searrow^{k-j} h'\,;e'$.

Exercise 51 Prove the bind lemma. You may use the decomposition of step-indexed termination lemma.

Lemma 65 (Bind).
If $(k, W, e) \in \mathcal{E}[\![A]\!]\delta$,
and $\forall j \leq k.\, \forall W' \sqsupseteq W.\, \forall v.\, (j, W', v) \in \mathcal{V}[\![A]\!]\delta \Rightarrow (j, W', K[v]) \in \mathcal{E}[\![B]\!]\delta$,
then $(k, W, K[e]) \in \mathcal{E}[\![B]\!]\delta$.

•

Exercise 52 Prove closure under expansion.

Lemma 66 (Closure under Expansion).
If e *reduces deterministically for* j *steps under any heap,*
and if $\forall h.\, h\,;e \leadsto^{j} h\,;e'$ *and* $(k, W, e') \in \mathcal{E}[\![A]\!]\delta$,
then $(k + j, W, e) \in \mathcal{E}[\![A]\!]\delta$.

•

Exercise 53 Consider this interface for a counter

$$\begin{aligned}\text{COUNTER} := \{&\mathsf{inc} : \mathbf{1} \to \mathbf{1},\\ &\mathsf{get} : \mathbf{1} \to \mathsf{int}\}\end{aligned}$$

and the following, extra-safe implementation of the counter that stores the current count *twice*, just to be sure that it does not mis-count or gets invalidated by cosmic radiation:

$$\begin{aligned}&\text{SafeCounter} : \mathbf{1} \to \text{COUNTER}\\ &\text{SafeCounter} := \lambda\_.\, \mathsf{let}\, c1 = \mathsf{new}\, \overline{0}\, \mathsf{in}\, \mathsf{let}\, c2 = \mathsf{new}\, \overline{0}\, \mathsf{in}\\ &\qquad \{\mathsf{inc} = \lambda\_.\, c1 \leftarrow !\,c1 + \overline{1}\,;c2 \leftarrow !\,c2 + \overline{1}\\ &\qquad\; \mathsf{get} = \lambda\_.\, \mathsf{let}\, v1 = !\,c1\, \mathsf{in}\, \mathsf{let}\, v2 = !\,c2\, \mathsf{in}\\ &\qquad\qquad \mathsf{assert}\,(v1 == v2)\,;v1\}\end{aligned}$$

Prove that SafeCounter is semantically well-typed. In other words, prove that for $\forall k, W.\, (k, W, \text{SafeCounter}) \in \mathcal{V}[\![\mathbf{1} \to \text{COUNTER}]\!]$. •

Proof conventions. As a convention, we omit some of the nitty-gritty details of these proofs. In particular, we omit all step-indices. We also omit the accounting of names for invariants. Furthermore, we use disjoint union to express heaps, which makes reasoning about world satisfaction much easier. Below, we show the proof of the inc case for the SafeCounter.

Proof.

We have:	To show:
$e_{\text{ctr}} = \mathsf{let}\ c1 = \mathsf{new}\ \overline{0}\ \mathsf{in}\ \mathsf{let}\ c2 = \mathsf{new}\ \overline{0}\ \mathsf{in}\ \ldots$	
W_0	$(\_, W_0, \lambda\_.\, e_{\text{ctr}}) \in \mathcal{V}[\![\mathbf{1} \to \text{COUNTER}]\!]$
$W_1 \sqsupseteq W_0$	
$(\_, W_1, v_1) \in \mathcal{V}[\![\mathbf{1}]\!]$	$(\_, W_1, e_{\text{ctr}}) \in \mathcal{E}[\![\text{COUNTER}]\!]$
$W_2 \sqsupseteq W_1$	
$h_1 : W_2$	
$h_1 ; e_{\text{ctr}} \searrow h_2 ; e_2$	$\exists W_3 \sqsupseteq W_2.\, h_2 : W_3 \wedge (\_, W_3, e_2) \in \mathcal{V}[\![\text{COUNTER}]\!]$
$h_2 = h_1 \uplus \overbrace{[\ell_1 \mapsto \overline{0}, \ell_2 \mapsto \overline{0}]}^{h_{\text{ctr}}}$	
$e_2 = \{\mathsf{inc} = \lambda\_.\, e_{\text{inc}}, \mathsf{get} = \lambda\_.\, e_{\text{get}}\}$	
$e_{\text{inc}} := \ell_1 \leftarrow !\,\ell_1 + \overline{1} ; \ell_2 \leftarrow !\,\ell_2 + \overline{1}$	
$e_{\text{get}} := \mathsf{let}\ v1 = !\,\ell_1\ \mathsf{in}\ \mathsf{let}\ v2 = !\,\ell_2\ \mathsf{in}\ \mathsf{assert}\ (v1 == v2) ; v1$	
Pick W_3 with new invariant i:	
$W_3[i] := H_{\text{ctr}} := \{h \mid \exists n.\, h(\ell_1) = h(\ell_2) = \overline{n}\}$	
	$h_2 : W_3$ (done by $h_{\text{ctr}} \in H_{\text{ctr}}$)
	$(\_, W_3, e_2) \in \mathcal{V}[\![\text{COUNTER}]\!]$
Case inc**:**	$(\_, W_3, \lambda\_.\, e_{\text{inc}}) \in \mathcal{V}[\![\mathbf{1} \to \mathbf{1}]\!]$
$W_4 \sqsupseteq W_3$	$(\_, W_4, e_{\text{inc}}) \in \mathcal{E}[\![\mathbf{1}]\!]$
$W_5 \sqsupseteq W_4$	
$h_3 : W_5$	
$h_3 ; e_{\text{inc}} \searrow h_4 ; e_4$	
	$\exists W_6 \sqsupseteq W_5.\, h_4 : W_6 \wedge (\_, W_6, e_4) \in \mathcal{V}[\![\mathbf{1}]\!]$
By world satisfaction: $W_5[i] = H_{\text{ctr}}$	
$h_3 = h_3' \uplus \underbrace{[\ell_1 \mapsto \overline{n}, \ell_2 \mapsto \overline{n}]}_{\in H_{\text{ctr}}}$	
By reduction, we know the code can execute safely and we get	
$e_4 = (), h_4 = h_3' \uplus [\ell_1 \mapsto \overline{n+1}, \ell_2 \mapsto \overline{n+1}]$	
Pick $W_6 := W_5$	
	$h_4 : W_6$ (done by definition of H_{ctr})
	$(\_, W_6, ()) \in \mathcal{V}[\![\mathbf{1}]\!]$ (trivial)

□

Semantic soundness. Once again, we re-establish semantic soundness. We are only interested in actual programs here, so we will assume that e does not contain any locations. First, we supplement the missing definitions:

Context Relation $\boxed{\mathcal{G}[\![\Gamma]\!]\delta}$

$$\mathcal{G}[\![\Gamma]\!]\delta := \{(k, W, \gamma) \mid \forall x : A \in \Gamma.(k, W, \gamma(x)) \in \mathcal{V}[\![A]\!]\delta\}$$

Semantic Typing $\boxed{\Delta ; \Gamma \vDash e : A}$

$$\Delta ; \Gamma \vDash e : A := \forall k, W.\, \forall \delta.\, \forall \gamma.\, (k, W, \gamma) \in \mathcal{G}[\![\Gamma]\!]\delta \Rightarrow (k, W, \gamma(e)) \in \mathcal{E}[\![A]\!]\delta$$

Now we can prove the core theorem.

Theorem 67 (Semantic Soundness)**.** *If* $\Delta\,;\Gamma \vdash e : A$*, then* $\Delta\,;\Gamma \vDash e : A$*.*

Proof. As before, we proceed by induction on the typing derivation for e, applying compatibility lemmas in each case. We proved compatibility for dereferencing in class and prove compatibility for new below. We leave compatibility for assignment as an exercise. □

Lemma 68 (Compatibility for first-order allocation (cf. NEW))**.**

$$\frac{\Delta\,;\Gamma \vDash e : a}{\Delta\,;\Gamma \vDash \mathsf{new}\,e : \mathsf{ref}\,a}$$

Proof.

We have:	To show:
(i) $\Delta\,;\Gamma \vDash e : a$	$\Delta\,;\Gamma \vDash \mathsf{new}\,e : \mathsf{ref}\,a$
Let $\delta, (k, W, \gamma) \in \mathcal{G}\llbracket\Gamma\rrbracket\delta$	$(k, W, \gamma(\mathsf{new}\,e)) \in \mathcal{E}\llbracket\mathsf{ref}\,a\rrbracket\delta$
	$(k, W, \mathsf{new}\,\gamma(e)) \in \mathcal{E}\llbracket\mathsf{ref}\,a\rrbracket\delta$
$(k, W, \gamma(e)) \in \mathcal{E}\llbracket a\rrbracket\delta$ by (i)	
$j \leq k, W' \sqsupseteq W, (j, W', v) \in \mathcal{V}\llbracket a\rrbracket\delta$	$(j, W', \mathsf{new}\,v) \in \mathcal{E}\llbracket\mathsf{ref}\,a\rrbracket\delta$
by bind with $K = \mathsf{new}\,\bullet$	
$j' < j, W'' \sqsupseteq W', h : W'', h\,;\mathsf{new}\,v \searrow^{j'} h'\,;e''$	
	$\exists W''' \sqsupseteq W''.\, h' : W''' \wedge (j - j', W''', e'') \in \mathcal{V}\llbracket\mathsf{ref}\,a\rrbracket\delta$
$j' = 1, h' = h[\ell \mapsto v], e'' = \ell, \ell \notin \mathsf{dom}(h)$ by inversion	
	$\exists W''' \sqsupseteq W''.\, h' : W''' \wedge (j - 1, W''', \ell) \in \mathcal{V}\llbracket\mathsf{ref}\,a\rrbracket\delta$
Pick $W''' = W'' +\!\!+ \{[\ell \mapsto \hat{v}] \mid \vdash \hat{v} : a\}$.	
	$W''' \sqsupseteq W''$
Trivial.	
	$h' : W'''$
This follows from $h : W''$, our assumption on v, and Lemma 62.	
	$(j - 1, W''', \ell) \in \mathcal{V}\llbracket\mathsf{ref}\,a\rrbracket\delta$
By definition of $\mathcal{V}\llbracket\mathsf{ref}\,a\rrbracket$.	

□

Exercise 54 Prove compatibility for first-order assignment.

Lemma 69 (Compatibility for first-order assignment)**.**

$$\frac{\Delta\,;\Gamma \vDash e : \mathsf{ref}\,a \qquad \Delta\,;\Gamma \vDash e' : a}{\Delta\,;\Gamma \vDash e \leftarrow e' : \mathbf{1}}$$

•

Exercise 55 Prove compatibility for first-order dereferencing.

Lemma 70 (Compatibility for first-order dereferencing)**.**

$$\frac{\Delta\,;\Gamma \vDash e : \mathsf{ref}\,a}{\Delta\,;\Gamma \vDash\, !\,e : a}$$

•

Exercise 56 In the lecture we have seen that extending the value relation for references in the naive way to allow higher-order state does not work because it breaks monotonicity. Here is a slightly different version of the naive approach:

$$\mathcal{V}[\![\mathsf{ref}\ A]\!]\delta := \{(k, W, \ell) \mid \exists i.\, W[i] \subseteq \{[\ell \mapsto v] \mid (k, W, v) \in \mathcal{V}[\![A]\!]\delta\}\}$$

Does this change preserve monotonicity? If yes, is it a sensible definition of the logical relation for references? •

4.7 Protocols

While the model presented above lets us prove interesting examples, we will now see its limitations. Consider the following program.

$$\begin{aligned} e :=\ & \mathsf{let}\, x = \mathsf{new}\, \overline{4} \\ & \quad \mathsf{in}\, \lambda f.\, f\,()\,;\, \mathsf{assert}\,(!\,x == 4) \\ & \quad\ : (\mathbf{1} \to \mathbf{1}) \to \mathbf{1} \end{aligned}$$

Picking the following invariant allows us to prove this program safe.

$$\{[\ell \mapsto \overline{4}]\}$$

Now consider the following program.

$$\begin{aligned} e :=\ & \mathsf{let}\, x = \mathsf{new}\, \overline{3} \\ & \quad \mathsf{in}\, \lambda f.\, x \leftarrow \overline{4}\,;\, f\,()\,;\, \mathsf{assert}\,(!\,x == 4) \\ & \quad\ : (\mathbf{1} \to \mathbf{1}) \to \mathbf{1} \end{aligned}$$

While the programs are similar in nature, we struggle to come up with an invariant that remains true throughout the execution, and still allows us to prove the program safe.

To solve this problem, we can extend our model with a stronger notion of invariants, which we refer to as "protocols" or "state transition systems". We parameterize our model by an arbitrary set of states *State*. For every island in the world ("invariant" would no longer be the right term), we store the legal transitions on this abstract state space, the current state, and which heap invariant is enforced at each abstract state. The world extension relation makes sure that the invariants and the transitions never change, but the current state may change according to the transitions. World satisfaction then enforces the invariants given by the current states of all islands.

$$\begin{array}{rrcl}
\text{Invariants} & \mathit{Inv} & := & \{\ \Phi : \mathbb{P}(\mathit{State} \times \mathit{State})\ (\Phi \text{ reflexive, transitive}), \\
 & & & \quad \mathrm{c} : \mathit{State}, \\
 & & & \quad \mathrm{H} : \mathit{State} \to \mathbb{P}(\mathrm{Heap})\ \} \\
\text{World} & W & \in & \bigcup_n \mathit{Inv}^n \\
\text{World Extension} & W' \sqsupseteq W & := & |W'| \geq |W| \land |W| = n \\
 & & & \land\, \forall i \in 1 \ldots n. \quad W'[i].\Phi = W[i].\Phi \\
 & & & \qquad\qquad\qquad \land\, W'[i].\mathrm{H} = W[i].\mathrm{H} \\
 & & & \qquad\qquad\qquad \land\, (W[i].\mathrm{c}, W'[i].\mathrm{c}) \in W[i].\Phi \\
\text{World Satisfaction} & h : W & := & |W| = n \land \exists h_1 \ldots h_n.\, h \supseteq h_1 \uplus \ldots \uplus h_n \\
 & & & \land\, \forall i \in 1 \ldots n.\, h_i \in W[i].\mathrm{H}(W[i].\mathrm{c})
\end{array}$$

Lemma 71 (World Extension is a pre-order). $\sqsupseteq$ *is reflexive and transitive.*

Value Relation $\boxed{\mathcal{V}[\![A]\!]\delta}$

$$\begin{aligned}
\mathcal{V}[\![\alpha]\!]\delta &:= \delta(\alpha)\\
\mathcal{V}[\![\mathsf{int}]\!]\delta &:= \{(k, W, \overline{n})\}\\
\mathcal{V}[\![A \times B]\!]\delta &:= \{(k, W, \langle v_1, v_2\rangle) \mid (k, W, v_1) \in \mathcal{V}[\![A]\!]\delta \land (k, W, v_2) \in \mathcal{V}[\![B]\!]\delta\}\\
\mathcal{V}[\![A \to B]\!]\delta &:= \{(k, W, \lambda x.\, e) \;\mid\; (\lambda x.\, e) \in \mathit{CVal} \land \forall j \le k, W' \sqsupseteq W, v.\\
&\qquad\qquad (j, W', v) \in \mathcal{V}[\![A]\!]\delta \Rightarrow (j, W', e[v/x]) \in \mathcal{E}[\![B]\!]\delta\}\\
\mathcal{V}[\![\mathsf{ref}\; a]\!]\delta &:= \{(k, W, \ell) \mid \exists i.\, W[i] = \{\ \Phi := \emptyset^\star, \mathrm{c} := s_0, \mathrm{H} := \lambda\_.\ \{[\ell \mapsto v] \mid \vdash v : a\}\ \}\}\\
\mathcal{V}[\![\forall \alpha.\, A]\!]\delta &:= \{(k, W, \Lambda.\, e) \mid (\Lambda.\, e) \in \mathit{CVal} \land \forall \tau \in \mathit{SemType}.\, (k, W, e) \in \mathcal{E}[\![A]\!](\delta, \alpha \mapsto \tau)\}\\
\mathcal{V}[\![\exists \alpha.\, A]\!]\delta &:= \{(k, W, \mathsf{pack}\, v) \mid \exists \tau \in \mathit{SemType}.\, (k, W, v) \in \mathcal{V}[\![A]\!](\delta, \alpha \mapsto \tau)\}\\
\mathcal{V}[\![\mu \alpha.\, A]\!]\delta &:= \{(k, W, \mathsf{roll}\, v) \mid \forall j < k.\, (j, W, v) \in \mathcal{V}[\![A[\mu\alpha.\, A/\alpha]]\!]\delta\}
\end{aligned}$$

For the case of reference types, we assert that there exists an invariant that makes sure the location always contains data of the appropriate type. To this end, we assume there is some fixed state s_0 which this invariant will be in, and the transition relation is the reflexive, transitive closure of the empty relation – in other words, the state cannot be changed.

The remaining definitions (expression relation, context relation, semantic typing) remain unchanged.

Exercise 57 (In $F^{\mu\,!}$ + STSs) This exercise operates in $F^{\mu\,!}$, that is System F – universal and existential types – plus recursive types (μ) and state (!). Furthermore, we work with the semantic model that is based on state-transition-systems (STSs).

Show the compatibility lemmas for the cases for first-order references: $\mathsf{new}\, e$, $!\, e$, $e_1 \leftarrow e_2$.

●

This model now is strong enough to prove safety of the example above, and of many interesting real-world programs.

Lemma 72. *Recall the example program from above which we used to motivate the introduction of state transition systems.*

$$\begin{aligned}
e := {}& \mathsf{let}\; x = \mathsf{new}\; \overline{3}\\
& \quad \mathsf{in}\; \lambda f.\, x \leftarrow \overline{4}\,;\, f\,()\,;\, \mathsf{assert}\,(!\, x == 4)\\
& \quad : (\mathbf{1} \to \mathbf{1}) \to \mathbf{1}
\end{aligned}$$

We can now show that $(\_, W, e) \in \mathcal{E}[\![(\mathbf{1} \to \mathbf{1}) \to \mathbf{1}]\!]$.

Proof.

We have: | To show:

$(\_, W_1, e) \in \mathcal{E}[\![(\mathbf{1} \to \mathbf{1}) \to \mathbf{1}]\!]$

$W_2 \sqsupseteq W_1, h : W_2$

$h \,;\, e \searrow^{-} h_0 \,;\, e_0$

$\exists W_3 \sqsupseteq W_2.\, h_0 : W_3 \wedge (\_, W_3, e_0) \in \mathcal{V}[\![(\mathbf{1} \to \mathbf{1}) \to \mathbf{1}]\!]$

$\ell \notin \mathsf{dom}(h)$

$h_0 = h \uplus [\ell \mapsto \overline{3}]$

$e_0 = \lambda f.\, \ell \leftarrow \overline{4} \,;\, f\,() \,;\, \mathsf{assert}\,(*\ell == \overline{4})$

Pick W_3 that extends W_2 with a new invariant i:

$W_3[i] := \{\ \Phi := \{(s_3, s_4)\}^\star, \mathrm{c} := s_3, \mathrm{H} := \lambda s_n.\ \{[\ell \mapsto n]\}\ \}$

$W_3 \sqsupseteq W_2$ (trivial)

$h_0 : W_3$ (done by $[\ell \mapsto \overline{3}] \in W_3[i].\mathrm{H}(s_3)$)

$(\_, W_3, e_0) \in \mathcal{V}[\![(\mathbf{1} \to \mathbf{1}) \to \mathbf{1}]\!]$

$W_4 \sqsupseteq W_3$

$(\_, W_4, v) \in \mathcal{V}[\![\mathbf{1} \to \mathbf{1}]\!]$

Let $e_1 := \ell \leftarrow \overline{4} \,;\, v\,() \,;\, \mathsf{assert}\,(*\ell == \overline{4})$ $\qquad (\_, W_4, e_1) \in \mathcal{E}[\![\mathbf{1}]\!]$

$W_5 \sqsupseteq W4, h_1 : W_5$

$h_1 \,;\, e_1 \searrow^{-} h_2 \,;\, e_2$

$\exists W_f \sqsupseteq W_5.\, h_2 : W_f \wedge (\_, W_f, e_2) \in \mathcal{V}[\![\mathbf{1}]\!]$

In order to pick W_f, we need to know what h_2 and e_2 can be,
so we need to symbolically execute e_1.
From $W_5 \sqsupseteq W_4 \sqsupseteq W_3$, $W_5[i] = W_3[i]$ except that $W_5[i].\mathrm{c}$ could be s_4.
From $h_1 : W_5$ and the invariant i, $\ell \in \mathsf{dom}(h_1)$.
So $h_1 \,;\, e_1 \rightsquigarrow^\star h_1[\ell \mapsto \overline{4}] \,;\, (v\,() \,;\, \mathsf{assert}\,(*\ell == \overline{4}))$
and (i) $h_1[\ell \mapsto \overline{4}] \,;\, (v\,() \,;\, \mathsf{assert}\,(*\ell == \overline{4})) \searrow^{-} h_2 \,;\, e_2$.
Now, we need to execute $v\,() \,;\, \mathsf{assert}\,(*\ell == \overline{4})$.
For that, we need to come up with a new world.
Let $W_6[i] = W_5[i]$ with $\mathrm{c} := s_4$
(ii) $h_1[\ell \mapsto \overline{4}] : W_6$
By Lemma 73, along with (i) and (ii),
we get $\exists W_7 \sqsupseteq W_6.\, h_2 : W_7 \wedge (\_, W_7, e_2) \in \mathcal{V}[\![\mathbf{1}]\!]$
Pick W_7 as given.
By transitivity of $\sqsupseteq$, $W_7 \sqsupseteq W_5$. □

Lemma 73 (Auxiliary "Lemma")**.** $(\_, W_5, (v\,() \,;\, \mathsf{assert}\,(*\ell == \overline{4}))) \in \mathcal{E}[\![\mathbf{1}]\!]$

Proof.

We have: To show:

From $(\_, W_4, v) \in \mathcal{V}[\![\mathbf{1} \to \mathbf{1}]\!]$, by monotonicity
$(\_, W_5, v) \in \mathcal{V}[\![\mathbf{1} \to \mathbf{1}]\!]$

$$(\_, W_5, (v\ () \mathbin{;} \mathsf{assert}\ (*\ell == \overline{4}))) \in \mathcal{E}[\![\mathbf{1}]\!]$$

By assumption, and def. of $\mathcal{V}[\![\mathbf{1} \to \mathbf{1}]\!]$,
$(\_, W_5, v\ ()) \in \mathcal{E}[\![\mathbf{1}]\!]$
We apply Lemma 65 (the bind lemma) with $K := \bullet \mathbin{;} \mathsf{assert}\ (!\,\ell == \overline{4})$

$$\forall W_6 \sqsupseteq W_5.\, (\_, W_6, \mathsf{assert}\ !\,\ell == \overline{4}) \in \mathcal{E}[\![\mathbf{1}]\!]$$

$W_7 \sqsupseteq W_6, W_6 \sqsupseteq W_5$
$h_4 : W_7$
$h_4 \mathbin{;} \mathsf{assert}\ (!\,\ell == \overline{4}) \searrow^{-} h_5 \mathbin{;} e_5$

$$\exists W_8 \sqsupseteq W_7.\, h_5 : W_8 \wedge (\_, W_8, e_5) \in \mathcal{V}[\![\mathbf{1}]\!]$$

$h_5(\ell) = \overline{4}$ because $W_7 \sqsupseteq W_5$ and $W_5[i].\mathrm{c} = s_4$
$h_5 = h_4 \wedge e_5 = ()$

$$\exists W_8 \sqsupseteq W_7.\, h_4 : W_8 \wedge (\_, W_8, ()) \in \mathcal{V}[\![\mathbf{1}]\!]$$

Trivial by picking $W_8 := W_7$. □

4.8 State & Existentials

We present several examples that combine references and existential types to encode stateful abstract data types.

4.8.1 Symbol ADT

Consider the following signature and the corresponding (stateful) implementation.

$$\begin{aligned}
\mathrm{SYMBOL} &:= \exists \alpha.\, \{\ \mathsf{mkSym} : \mathbf{1} \to \alpha, \\
&\qquad\qquad \mathsf{check} : \alpha \to \mathbf{1}\ \} \\
\mathrm{Symbol} &:= \mathsf{let}\ c = \mathsf{new}\ \overline{0}\ \mathsf{in} \\
&\qquad \mathsf{pack}\left\langle \mathsf{int}, \begin{array}{l}\{\ \mathsf{mkSym} := \lambda\_.\ \mathsf{let}\ x = !\,c\ \mathsf{in}\ c \leftarrow x + 1 \mathbin{;} x, \\ \quad \mathsf{check} := \lambda x.\ \mathsf{assert}\ (x < !\,c)\ \}\end{array} \right\rangle \mathsf{as}\ \mathrm{SYMBOL}
\end{aligned}$$

Intuitively, the function check guarantees that only mkSym can generate values of type α.

The proof of safety for Symbol showcases how semantic types and invariants can work together in interesting ways. Instead of going through the proof, we give the invariant that will be associated with the location ℓ_c correspding to the reference c, and the semantic type for the type variable α.

$$\begin{aligned}
W[i] := \{\ & \Phi := \{(s_{n_1}, s_{n_2}) \mid (n_2 \geq n_1)\}\,, \\
& \mathrm{c} := s_0, \\
& \mathrm{H} := \lambda s_n.\ \{[\ell_c \mapsto n]\}\ \} \\
\alpha \mapsto\ & \{(\_, W, \overline{n}) \mid 0 \leq n \wedge W[i].\mathrm{c} = s_m \wedge n < m\}
\end{aligned}$$

Note that the semantic type assigned to α is defined in terms of the transition system that governs ℓ_c. Consequently, the semantic type will grow over time as the value in ℓ_c increases. This is possible because when we define the interpretation of α, we already picked the new world, and hence we know which index our island will have. This is similar to how, when we define the island, we already know the actual location ℓ_c picked by the program.

In the proof of safety of Symbol, we know that both mkSym and check will only be called in worlds future to the one in which we established our invariant. Consequently, we can rely on the existence of the transition we set up. The transitions we take mirror the change to c in the program.

Proof Sketches. We refer to the above as a *proof sketch.* In such a sketch, we only give the invariants that are picked, the interpretations of semantic types, and how we update the state of STSs.

4.8.2 Twin Abstraction

The following signature represents an ADT that can generate two different types of values (red and blue). The check function guarantees that values from distinct "colors" will never be equal. Interestingly, the ADT is implemented by picking both red and blue values from an ever-increasing counter.

$$\begin{aligned}
\text{TWIN} := \exists\alpha.\,\exists\beta.\,\{\ \ &\text{mkRed} : \mathbf{1} \to \alpha,\\
&\text{mkBlue} : \mathbf{1} \to \beta,\\
&\ \ \text{check} : (\alpha, \beta) \to \mathbf{1}\ \ \}\\
\text{Twin} := \mathsf{let}\, c = \mathsf{new}\, \overline{0}\ \mathsf{in}\ \ &\\
\mathsf{pack}\,\mathsf{pack}\,\{\ \ &\text{mkRed} := \lambda\_.\ \mathsf{let}\, x = {!\,c}\ \mathsf{in}\ c \leftarrow x + 1\,;\, x,\\
&\text{mkBlue} := \lambda\_.\ \mathsf{let}\, x = {!\,c}\ \mathsf{in}\ c \leftarrow x + 1\,;\, x,\\
&\ \ \text{check} := \lambda(x, y).\ \mathsf{assert}\,(x \neq y)\ \ \}
\end{aligned}$$

Note how the implementation does not keep track of the assignment of numbers to the red or blue type. Nonetheless, we are able to prove this implementation semantically safe. It is perhaps not surprising that we cannot rely entirely on physical state to guarantee the separation of the red and blue type. We make use of so-called "ghost state", which is auxiliary state that only exists in the verification of the program.

In addition to the value of the counter value n, our transition system also has to keep track of two sets which we suggestively call R and B. These sets are disjoint represent the part of generated values (between 0 and n) that belong to the red and blue type, respectively.

Again, we tie the semantic types for α and β to the transition system to ensure that their interpretation can grow over time. This time, however, we additionally require that the values in those semantic types belong to R or B, respectively.

$$W[i] := \left\{ \begin{array}{l} \Phi := \left\{ (s_{n,R,B}, s_{n',R',B'}) \;\middle|\; \begin{array}{l} R \uplus B = \{0 \dots n-1\} \\ R' \uplus B' = \{0 \dots n'-1\} \\ n \leq n',\ R \subseteq R',\ B \subseteq B' \end{array} \right\}, \\ \mathrm{c} := s_{0,\emptyset,\emptyset}, \\ \mathrm{H} := \lambda s_{n,R,B}.\ \{[\ell_c \mapsto n]\} \end{array} \right\}$$

$$\alpha \mapsto \{(\_, W, \overline{n}) \mid W[i].\mathrm{c} = s_{m,R,B} \wedge n \in R\}$$
$$\beta \mapsto \{(\_, W, \overline{n}) \mid W[i].\mathrm{c} = s_{m,R,B} \wedge n \in B\}$$

Given these definitions, proving safety of Twin is straight-forward. When we give out a red value, we update the R component of our state. Accordingly, when we give out a blue value, we update B. In both cases, we increase the n component by one.

Exercise 58 (In $F^{\mu\,!}$ + STSs) Consider the following stateful abstract data type. We assume we are given some *first-order* types a and b.

$$\begin{array}{rl} \mathrm{SUM} := \exists \beta.\ \{ & \mathsf{setA} : a \to \beta, \\ & \mathsf{setB} : b \to \beta, \\ & \mathsf{getA} : \beta \to \mathbf{1} + a, \\ & \mathsf{getB} : \beta \to \mathbf{1} + b\ \} \end{array}$$

$$\begin{array}{l} \mathrm{MySum} := \lambda\_.\ \mathsf{let}\ x = \mathsf{new}\ \langle \overline{1}, \overline{1} \rangle\ \mathsf{in} \\ \qquad \mathsf{pack}\ \{\ \mathsf{setA} := \lambda y.\ x \leftarrow \langle \overline{1}, y \rangle, \\ \qquad\qquad \mathsf{setB} := \lambda y.\ x \leftarrow \langle \overline{2}, y \rangle, \\ \qquad\qquad \mathsf{getA} := \lambda\_.\ \mathsf{let}\ (c, d) = !\,x\ \mathsf{in} \\ \qquad\qquad\qquad \mathsf{if}\ c == \overline{1}\ \mathsf{then}\ \mathsf{inj}_2\ d\ \mathsf{else}\ \mathsf{inj}_1\ (), \\ \qquad\qquad \mathsf{getB} := \lambda\_.\ \mathsf{let}\ (c, d) = !\,x\ \mathsf{in} \\ \qquad\qquad\qquad \mathsf{if}\ c == \overline{2}\ \mathsf{then}\ \mathsf{inj}_2\ d\ \mathsf{else}\ \mathsf{inj}_1\ ()\ \} \end{array}$$

The client can only obtain an element of β once they called one of the two setters. This means that the getters can rely on the data having been initialized.

Prove that this implementation is semantically well-typed:

$$\forall k, W.\ (k, W, \mathrm{MySum}) \in \mathcal{V}[\![\mathbf{1} \to \mathrm{SUM}]\!]$$

Give only a proof *sketch* (*i.e.*, give only the invariants, the semantic type picked for β, and how the STSs are updated). •

Exercise 59 (In $F^{\mu\,!}$ + STSs) Consider the following code:

$$\begin{array}{l} e_{\mathrm{ctr}} := \mathsf{let}\ c = \mathsf{new}\ \overline{0}\ \mathsf{in} \\ \qquad \lambda n.\ \mathsf{if}\ n > !\,c\ \mathsf{then}\ c \leftarrow n\ \mathsf{else}\ (); \\ \qquad\quad \lambda\_.\ \mathsf{assert}\ (!\,c \geq n) \end{array}$$

Intuitively, this function allows everyone to *increment* the counter to values of their choice.

After every increment, a little stub is returned that asserts that the counter will never be below the given n again.

Show that e_{ctr} is safe:

$$\forall k, W.\, (k, W, e_{\mathrm{ctr}}) \in \mathcal{E}[\![\mathsf{int} \to (\mathbf{1} \to \mathbf{1})]\!]$$

Give a proof *outline*, not a proof sketch (following the conventions described previously, in which step-indices are ignored). •

Exercise 60 (In $F^{\mu\,!}$ + Inv) This exercise operates in $F^{\mu\,!}$ and the semantic model with plain invariants, *i.e.*, no STSs.

When we started building a semantic model for our language with state, we restricted the *type system* to only allow storing first-order data into the heap. This was necessary for the proof of semantic soundness of the model. However, we did not change the operational semantics of the language: We can still write programs that use higher-order state, we just do not automatically obtain their safety. In some specific cases though, the use of higher-order state can actually be justified in our model.

Consider the following simple program:

$e_{\mathrm{id}} := \lambda f.\, \mathsf{let}\, x = \mathsf{new}\, f \,\mathsf{in}\, \lambda y.\ !\,x\, y$

Show that e_{id} is semantically well-typed: For any closed types A, B, prove

$$\forall k, W.\, (k, W, e_{\mathrm{id}}) \in \mathcal{V}[\![(A \to B) \to (A \to B)]\!]$$

Give a proof *outline*, not a proof sketch, following the conventions described previously, in which step-indices are ignored. •

4.9 Semantically well-typed expressions are safe

The following theorem tells us that, in order to to prove that a closed expression is safe—or *progressive* in our old tongue, it is *adequate* to show that the expression is semantically well-typed.

Theorem 74 (Adequacy)**.**

If $\vDash e : A$ *then*
for all h, e', h' *such that* $h\,; e \rightsquigarrow^{\star} e'\,; h'$*, either* e' *is a value or* $h'\,; e'$ *can make a step.*

Proof.

We have:	To show:
(i) $\vDash e : A$	
$h\,;e \rightsquigarrow^{\star} e'\,;h'$	e' is a value or $h'\,;e'$ can make a step
Suppose $h'\,;e'$ cannot take a step any more.	
	e' is a value
(Here we exploit the fact that the reduction relation $\rightsquigarrow$ is decidable.)	
So $h\,;e \searrow^{j} h'\,;e'$.	
From (i), have $\forall k, W.\, (k, W, e) \in \mathcal{E}[\![A]\!]$.	
We instantiate this with $k := j + 1$ and $W, W' := [\,]$.	
It is trivial to see that $h : W$.	
So we have some W'' s.t. $W'' \sqsupseteq W' \wedge h' : W'' \wedge (1, W'', e') \in \mathcal{V}[\![A]\!]$.	
Thus e' is a value.	$\square$

5 Program Logic

For every language up to this point, we first defined its operational semantics and then used the operational semantics to reason about terms of the language. We now turn to a different approach sometimes referred to as *axiomatic semantics*: the axiomatic semantics assigns meaning to programs in the form of a program logic—a set of rules that we can use to modularly reason about programs.

Why axiomatic semantics? In contrast to language models based on operational semantics and types, axiomatic semantics are aimed at compositional *program verification.* For example, consider verifying the Euclidian algorithm for computing the greatest common divisor:

$$\begin{aligned} \mathrm{euclid}(a, b) &:= \textsf{if}\, b == 0 \,\textsf{then}\, a \,\textsf{else}\, \mathrm{euclid}(b, \mathrm{mod}(a, b)) \\ \mathrm{mod}(a, b) &:= a - (a \,\textsf{div}\, b) * b \end{aligned}$$

In a typed model based on operational semantics, we can at best type both functions as $\textsf{int} \times \textsf{int} \to \textsf{int}$. But even for the type $\textsf{int} \times \textsf{int} \to \textsf{int}$, we have the problem that $a \,\textsf{div}\, b$ could be stuck for $b = 0$. So how can we say anything useful about these two functions? With a program logic (*i.e.*, an axiomatic semantics), we can give compositional *specifications* for these functions. For example, we can express the specification as *Hoare triples*:

$$\{a \geq 0 \land b > 0\}\, \mathrm{mod}(a, b)\, \{c.\, \exists k \geq 0.\, a = b \cdot k + c \land 0 \leq c < b\}$$

$$\{a \geq 0 \land b \geq 0\}\, \mathrm{euclid}(a, b)\, \{c.\, \mathrm{gcd}(a, b, c)\}$$

where gcd is a predicate expressing that the c is the greatest common divisor of a and b. Such a triple $\{P\}\, e\, \{v.\, Q(v)\}$ expresses that if the *precondition* P is true, then e will execute to a value v (if it terminates) satisfying the *postcondition* $Q(v)$. As we will see below, when we prove such triples, we learn information (*e.g.*, from case distinctions), which we can use to justify subsequent reasoning steps (*e.g.*, the application of mod).

5.1 Hoare Logic

We start with the canonical example of a program logic: *Hoare logic* [4]. In our setting, we discuss a Hoare logic with Hoare triples $\{P\}\, e\, \{v.\, Q(v)\}$ for terms from an extended λ-calculus[1]:

$$\begin{array}{lrcl} \text{Propositions} & P, Q, R & ::= & \phi \mid \exists x.\, P(x) \mid \forall x.\, P(x) \mid P \land Q \mid P \lor Q \\ \text{Terms} & e & ::= & x \mid v \mid \textsf{fix}\, f\, x.\, e \mid e_1\, e_2 \mid e_1 \,o\, e_2 \mid \textsf{if}\, e_1 \,\textsf{then}\, e_2 \,\textsf{else}\, e_3 \\ & & \mid & (e_1, e_2) \mid \pi_1 e \mid \pi_2 e \mid \textsf{inj}_1 e \mid \textsf{inj}_2 e \\ & & \mid & \textsf{match}\, e_1 \,\textsf{of}\, \textsf{inj}_1\, x \Rightarrow e_2 \mid \textsf{inj}_2\, y \Rightarrow e_3 \,\textsf{end} \\ \text{Values} & v & ::= & \textsf{fix}\, f\, x.\, e \mid \overline{n} \mid (v_1, v_2) \mid \textsf{inj}_1 v \mid \textsf{inj}_2 v \mid \textsf{true} \mid \textsf{false} \end{array}$$

where ϕ is an arbitrary, simple meta level proposition (*e.g.*, $\textsf{True}$, $\textsf{False}$, $n < 0$, or $\textsf{even}(k)$).

[1]As a consequence of considering the λ-calculus instead of an imperative language (as done in more traditional formulations of Hoare logic), we do not have explicit variable assignments in the language. Fittingly, our pre- and postconditions do not contain *variable assignments* and we use substitution instead of updating pre- and postconditions.

Proposition Proof Rules

$$\boxed{P \vdash Q}$$

$$\textsc{Refl}\quad P \vdash P \qquad \textsc{Trans}\ \frac{P \vdash Q \qquad Q \vdash R}{P \vdash R} \qquad \textsc{Pure}\ \frac{\phi}{P \vdash \phi} \qquad \textsc{FromPure}\ \frac{P \vdash \phi \qquad \phi \Rightarrow (P \vdash Q)}{P \vdash Q} \qquad \textsc{AndElimL}\quad P \land Q \vdash P$$

$$\textsc{AndElimR}\quad P \land Q \vdash Q \qquad \textsc{AndIntro}\ \frac{P \vdash Q \qquad P \vdash R}{P \vdash Q \land R} \qquad \textsc{OrIntroL}\quad P \vdash P \lor Q \qquad \textsc{OrIntroR}\quad Q \vdash P \lor Q \qquad \textsc{OrElim}\ \frac{P \vdash R \qquad Q \vdash R}{P \lor Q \vdash R}$$

$$\textsc{AllIntro}\ \frac{\forall x : X.\, (P \vdash Q(x))}{P \vdash \forall x : X.\, Q(x)} \qquad \textsc{AllElim}\ \frac{y : X}{(\forall x : X.\ P(x)) \vdash P(y)} \qquad \textsc{ExistIntro}\ \frac{y : X \qquad P \vdash Q(y)}{P \vdash \exists x : X.\, Q(x)}$$

$$\textsc{ExistElim}\ \frac{\forall x : X.\, (P(x) \vdash Q)}{\exists x : X.\, P(x) \vdash Q}$$

Here, we write $\phi \Rightarrow \psi$ for a *meta-level* implication. (You can think of it as a Coq implication.)

Exercise 61 Derive some common proof principles:

$$\textsc{Weakening}\ \frac{P \vdash R}{P \land Q \vdash R} \qquad \textsc{True}\quad P \vdash \mathsf{True} \qquad \textsc{False}\quad \mathsf{False} \vdash P \qquad \textsc{AndComm}\quad P \land Q \vdash Q \land P \qquad \textsc{OrComm}\quad P \lor Q \vdash Q \lor P$$

$$\textsc{AllComm}\quad \forall x, y.\, P(x,y) \vdash \forall y, x.\, P(x,y) \qquad \textsc{ExComm}\quad \exists x, y.\, P(x,y) \vdash \exists y, x.\, P(x,y)$$

●

Hoare Rules

$$\boxed{\{P\}\, e\, \{v.\, Q(v)\}}$$

$$\textsc{Value}\quad \{P(v)\}\, v\, \{w.\, P(w)\} \qquad \textsc{Consequence}\ \frac{P' \vdash P \qquad (\forall v.\, Q(v) \vdash Q'(v)) \qquad \{P\}\, e\, \{v.\, Q(v)\}}{\{P'\}\, e\, \{v.\, Q'(v)\}}$$

$$\textsc{Bind}\ \frac{\{P\}\, e\, \{v.\, Q(v)\} \qquad \forall v.\, \{Q(v)\}\, K[v]\, \{w.\, R(w)\}}{\{P\}\, K[e]\, \{w.\, R(w)\}} \qquad \textsc{Pure}\ \frac{P \vdash \phi \qquad \phi \Rightarrow \{P\}\, e\, \{v.\, Q(v)\}}{\{P\}\, e\, \{v.\, Q(v)\}}$$

$$\textsc{Exists}\ \frac{\forall x : X.\, \{P(x)\}\, e\, \{v.\, Q(v)\}}{\{\exists x : X.\, P(x)\}\, e\, \{v.\, Q(v)\}} \qquad \textsc{PureStep}\ \frac{e_1 \rightarrow_{\mathsf{pure}} e_2 \qquad \{P\}\, e_2\, \{v.\, R(v)\}}{\{P\}\, e_1\, \{v.\, R(v)\}}$$

Pure Steps

$$\boxed{e_1 \rightarrow_{\mathsf{pure}} e_2}$$

$$\frac{e_1 \rightarrow_{\mathsf{pure}} e_2}{K[e_1] \rightarrow_{\mathsf{pure}} K[e_2]} \qquad \overline{n} + \overline{m} \rightarrow_{\mathsf{pure}} \overline{n+m} \qquad \overline{n} - \overline{m} \rightarrow_{\mathsf{pure}} \overline{n-m} \qquad \frac{m \neq 0}{\overline{n}\,\mathsf{div}\,\overline{m} \rightarrow_{\mathsf{pure}} \overline{n/m}}$$

$$\frac{n = m}{\overline{n} == \overline{m} \rightarrow_{\mathsf{pure}} \mathsf{true}} \qquad \frac{n \neq m}{\overline{n} == \overline{m} \rightarrow_{\mathsf{pure}} \mathsf{false}} \qquad (\mathsf{fix}\, f\, x.\, e)\, v \rightarrow_{\mathsf{pure}} e[\mathsf{fix}\, f\, x.\, e/f, v/x]$$

$$\overline{n} * \overline{m} \rightarrow_{\mathsf{pure}} \overline{n \cdot m} \qquad \mathsf{if\,true\,then}\; e_1\; \mathsf{else}\; e_2 \rightarrow_{\mathsf{pure}} e_1 \qquad \mathsf{if\,false\,then}\; e_1\; \mathsf{else}\; e_2 \rightarrow_{\mathsf{pure}} e_2$$

$$\mathsf{match\;inj}_1 v\;\mathsf{of\;inj}_1\, x \Rightarrow e_1 \mid \mathsf{inj}_2\, y \Rightarrow e_2\;\mathsf{end} \rightarrow_{\mathsf{pure}} e_1[v/x] \qquad \pi_1(v_1, v_2) \rightarrow_{\mathsf{pure}} v_1$$

$$\mathsf{match\;inj}_2 v\;\mathsf{of\;inj}_1\, x \Rightarrow e_1 \mid \mathsf{inj}_2\, y \Rightarrow e_2\;\mathsf{end} \rightarrow_{\mathsf{pure}} e_2[v/x] \qquad \pi_2(v_1, v_2) \rightarrow_{\mathsf{pure}} v_2$$

Exercise 62 Traditionally, Hoare logic is presented without the rule PureStep. Instead, following the *axiomatic style*, they are presented with structural rules for all language connectives. Derive the following structural rules:

Rec
$$\frac{\{P\}\; e[\mathsf{fix}\, f\, x.\, e/f, v/x]\; \{w.\, Q(w)\}}{\{P\}\; (\mathsf{fix}\, f\, x.\, e)\, v\; \{w.\, Q(w)\}}$$

IsNeq
$$\{n \neq m\}\; \overline{n} == \overline{m}\; \{v.\, v = \mathsf{false}\}$$

Sub
$$\{\mathsf{True}\}\; \overline{n} - \overline{m}\; \{v.\, v = \overline{n-m}\}$$

Add
$$\{\mathsf{True}\}\; \overline{n} + \overline{m}\; \{v.\, v = \overline{n+m}\}$$

IfFalse
$$\frac{\{P\}\; e_2\; \{v.\, Q(v)\}}{\{P\}\; \mathsf{if\,false\,then}\; e_1\; \mathsf{else}\; e_2\; \{v.\, Q(v)\}}$$

•

Exercise 63 Derive the following rule for Hoare triples with pure preconditions:

PurePre
$$\{\phi\}\; e\; \{v.\, Q(v)\} \iff (\phi \Rightarrow \{\mathsf{True}\}\; e\; \{v.\, Q(v)\})$$

•

Program Verification We use the rules for $\{P\}\; e\; \{v.\, Q\}$ to verify the following implementation of the Fibonacci function:

$$\mathsf{fix\,fib\,n.\,if\,n} == \overline{0}\;\mathsf{then}\;\overline{0}\;\mathsf{else\,if\,n} == \overline{1}\;\mathsf{then}\;\overline{1}\;\mathsf{else\,fib(n} - \overline{1}) + \mathsf{fib(n} - \overline{2})$$

Lemma 75. $\{n \geq 0\}\; \mathsf{fib}\; \overline{n}\; \{v.\, v = \overline{F_n}\}$ *where F_n is the n-th Fibonacci number.*

Proof. Let $n \geq 0$ by PurePre. We show $\{\mathsf{True}\}\; \mathsf{fib}\; \overline{n}\; \{v.\, v = \overline{F_n}\}$ by strong induction on n. The base cases follow with Lemma 76 and the recursive case with Lemma 77. □

Lemma 76. $\{\mathsf{True}\}\; \mathsf{fib}\; \overline{0}\; \{v.\, v = 0\}$ *and* $\{\mathsf{True}\}\; \mathsf{fib}\; \overline{1}\; \{v.\, v = 1\}$.

Exercise 64 Prove Lemma 76. •

Lemma 77.

$$\frac{\{\mathsf{True}\}\,\mathsf{fib}(\overline{n-1})\,\{v.\,v = \overline{F_{n-1}}\} \qquad \{\mathsf{True}\}\,\mathsf{fib}(\overline{n-2})\,\{v.\,v = \overline{F_{n-2}}\}}{\{n > 1\}\,\mathsf{fib}\ \overline{n}\,\{v.\,v = \overline{F_n}\}}$$

Proof. Let $n > 1$ by PurePre. By executing several pure steps (*e.g.*, the conditions of the if-expressions), it remains to show $\{\mathsf{True}\}\,\mathsf{fib}(\overline{n-1}) + \mathsf{fib}(\overline{n-2})\,\{v.\,v = \overline{F_n}\}$. We bind $\mathsf{fib}(\overline{n-2})$ with Bind and use our assumption for the recursive call. It remains to show $\{v = \overline{F_{n-2}}\}\,\mathsf{fib}(\overline{n-1}) + v\,\{w.\,w = \overline{F_n}\}$, so $\{\mathsf{True}\}\,\mathsf{fib}(\overline{n-1}) + \overline{F_{n-2}}\,\{w.\,w = \overline{F_n}\}$ by PurePre. Similarly, we evaluate the recursive call with $\overline{n-1}$, leaving us to prove $\{\mathsf{True}\}\,\overline{F_{n-1}} + \overline{F_{n-2}}\,\{w.\,w = \overline{F_n}\}$, which follows from a pure step and rule Value. ☐

Exercise 65 (Old School) Prove

$$\frac{\{\mathsf{True}\}\,\mathsf{fib}\ \overline{n-1}\,\{v.\,v = \overline{F_{n-1}}\} \qquad \{\mathsf{True}\}\,\mathsf{fib}\ \overline{n-2}\,\{v.\,\overline{F_{n-2}}\}}{\{n > 1\}\,\mathsf{fib}\ \overline{n}\,\{v.\,v = \overline{F_n}\}}$$

without using the rule PureStep and, instead, use *only* the structural rules you derived previously. •

Example 78 (Euclid). *We can also verify the specification for the* euclid *function given above:*

$$\{a \geq 0 \wedge b > 0\}\,\mathrm{mod}(a, b)\,\{c.\,\exists k \geq 0.\, a = b \cdot k + c \wedge 0 \leq c < b\}$$

$$\{a \geq 0 \wedge b \geq 0\}\,\mathrm{euclid}(a, b)\,\{c.\,\mathrm{gcd}(a, b, c)\}$$

The specification for mod *can be straightforwardly verified using the rules for Hoare triples and standard mathematical reasoning. The specification for* euclid *is proved by strong induction on $b \geq 0$. We do a case analysis on b and end up with a base case and the inductive step:*

$$\{\mathsf{True}\}\,\mathrm{euclid}\ \overline{a}\ \overline{0}\,\{v.\,v = a\}$$

$$\frac{\forall c.\,\{0 \leq c < b\}\,\mathrm{euclid}\ \overline{b}\ \overline{c}\,\{d.\,\mathrm{gcd}(b, c, d)\}}{\{b > 0 \wedge a \geq 0\}\,\mathrm{euclid}\ \overline{a}\ \overline{b}\,\{c.\,\mathrm{gcd}(a, b, c)\}}$$

For the inductive step, we use the inductive hypothesis for the result of mod *and then employ the property* $\mathrm{gcd}(b, c, d) \leftrightarrow \mathrm{gcd}(b \cdot k + c, b, d)$.

Exercise 66 Let fix fac n := if n == $\overline{0}$ then $\overline{1}$ else $n * \mathrm{fac}(\mathrm{n} - \overline{1})$. Prove that fac computes the factorial function: $\{n \geq 0\}\,\mathrm{fac}\ \overline{n}\,\{v.\,v = \overline{n!}\}$. •

5.2 Separation Logic

The Hoare logic we have seen above is quite limited in that it can only be used to reason about pure, functional programs. To support more expressive reasoning about programs with a heap, we now turn to a successor of Hoare logic[2]: *separation logic* [8].

[2]Traditionally, Hoare logics are designed for imperative languages. Fittingly, they have built in support for reasoning about the program state in the form of variable assignments. However, they do not make

We extend our propositions, terms, and values:

$$\begin{array}{rrcl} \text{Propositions} & P, Q, R & ::= & \cdots \mid \ell \mapsto v \mid P * Q \\ \text{Terms} & e & ::= & \cdots \mid \mathsf{new}(e) \mid e_1 \leftarrow e_2 \mid\ !\,e \\ \text{Values} & v & ::= & \cdots \mid \ell \end{array}$$

The proposition $\ell \mapsto v$ (read "ℓ points to v") asserts that the location ℓ currently stores the value v. The proposition $P * Q$ is called the *separating conjunction.* Like conjunction, it asserts that both P and Q are true. What makes it special—and the distinguishing feature of separation logic—is that it ensures P and Q are satisfied by *disjoint* parts of the heap. That is, the proposition $\ell \mapsto v * \ell \mapsto w$ is false, because both conjuncts do not refer to separate parts of the heap. Correspondingly, the proposition $\ell \mapsto v * \ell' \mapsto w$ ensures that $\ell \neq \ell'$ and $\ell \mapsto v$ and $\ell' \mapsto w$.

Proposition Proof Rules $\boxed{P \vdash Q}$

$$\textsc{SepWeaken}\quad P * Q \vdash P \qquad \textsc{SepTrue}\quad P \vdash P * \mathsf{True} \qquad \textsc{SepComm}\quad P * Q \vdash Q * P \qquad \textsc{SepSplit}\quad \frac{P \vdash P' \qquad Q \vdash Q'}{P * Q \vdash P' * Q'}$$

$$\textsc{SepAssoc}\quad P * (Q * R) \dashv\vdash (P * Q) * R \qquad \textsc{ExistsSep}\quad (\exists x : X. P(x)) * Q \vdash (\exists x : X. P(x) * Q)$$

$$\textsc{PointstoSep}\quad \ell \mapsto v * \ell \mapsto w \vdash \mathsf{False} \qquad \textsc{PointstoAnd}\quad \ell \mapsto v \wedge \ell \mapsto w \vdash v = w$$

We write $P \dashv\vdash Q$ as a shorthand for $P \vdash Q$ and $Q \vdash P$.

Exercise 67 Derive the following proof rules:

$$\textsc{PointstoDisj}\quad \ell \mapsto v * \ell' \mapsto w \vdash \ell \neq \ell' \qquad \textsc{SepExists}\quad (\exists x : X. P(x)) * Q \dashv\vdash (\exists x : X. P(x) * Q)$$

●

Example 79 (Chains). *Using the separating conjunction, we can concisely express the notion of a chain of references:*

$$\begin{aligned} \text{chain}(\ell, r) &:= \exists n > 0.\, \text{chain}_n(\ell, r) \\ \text{chain}_0(\ell, r) &:= \ell = r \\ \text{chain}_{n+1}(\ell, r) &:= \exists t.\, \ell \mapsto t * \text{chain}_n(t, r) \end{aligned}$$

Let us look at a simple chain:

$$\ell_2 \mapsto \ell_3 * \ell_1 \mapsto \ell_2 * \ell_3 \mapsto \ell_4 \vdash \text{chain}(\ell_1, \ell_4)$$

memory locations first class values and, thus, fall short of the expressive power of separation logic, which supports nested references and linked lists.

We can also establish some simple properties of chains:

$$\begin{aligned}
\ell \mapsto r &\vdash \mathrm{chain}(\ell, r) \\
\ell \mapsto r * \mathrm{chain}(r, t) &\vdash \mathrm{chain}(\ell, t) \\
\mathrm{chain}(\ell, r) * \mathrm{chain}(r, t) &\vdash \mathrm{chain}(\ell, t) \\
\mathrm{chain}(\ell, r) * \mathrm{chain}(\ell, t) &\vdash \mathsf{False} \\
\mathrm{chain}(\ell, r) * \mathrm{chain}(r, \ell) &\vdash \mathrm{cycle}(\ell)
\end{aligned}$$

where $\mathrm{cycle}(\ell) := \mathrm{chain}(\ell, \ell)$.

Exercise 68 Prove the chain properties. •

Exercise 69 (Awkward Cycles) Can you prove $\mathrm{cycle}(\ell) \vdash \exists r.\, \mathrm{chain}(\ell, r) * \mathrm{chain}(r, \ell)$? •

Separation logic rules $\boxed{\{P\}\, e\, \{v.\, Q(v)\}}$

We extend our Hoare proof rules by the following separation logic proof rules.

FRAME
$$\frac{\{P\}\, e\, \{v.\, Q\}}{\{P * R\}\, e\, \{v.\, Q * R\}}$$

NEW
$$\{\mathsf{True}\}\ \mathsf{new}\ v\ \{w.\, \exists \ell.\, w = \ell * \ell \mapsto v\}$$

STORE
$$\{\ell \mapsto v\}\ \ell \leftarrow w\ \{\_.\, \ell \mapsto w\}$$

LOAD
$$\{\ell \mapsto v\}\ !\,\ell\ \{w.\, w = v * \ell \mapsto v\}$$

Note that FRAME is completely agnostic about the expression e it is applied to while all the other rules (*i.e.*, NEW, STORE, and LOAD) are structural: they apply only to specific forms of expressions. Out of these four rules, the FRAME rule deserves the most attention: it is the heart of separation logic. Since the separating conjunction $P * R$ expresses that P and R assert facts about *disjoint* parts of the heap, we know that if e only needs the P part of the heap, then R remains unchanged and holds again after the execution of e.

Ownership reasoning Let us dwell on the frame rule for a bit longer. What it concisely encapsulates is the so-called "ownership reasoning" of separation logic. The idea is that the assertion $\ell \mapsto v$ expresses "ownership" of the location ℓ. Owning a location ℓ means no other program part can manipulate it. For example, in the triple $\{\ell \mapsto \overline{0}\}\ f\ (\ell, \ell')\ \{\_.\, \ell \mapsto \overline{42}\}$ the function f "owns" ℓ for the duration of the call and it can be sure that no other program part (even in a concurrent setting) will interfere with ℓ. Moreover, from the triple $\{\ell \mapsto \overline{0}\}\ f\ (\ell, \ell')\ \{\_.\, \ell \mapsto \overline{42}\}$ we know f *only* needs the location ℓ from the current heap—ownership of all other locations can be framed around the function call. For example, we can verify the following program:

$$e_{\mathsf{own}} := \mathsf{let}\ x := \mathsf{new}(\overline{0})\ \mathsf{in}\ \mathsf{let}\ y := \mathsf{new}(\overline{42})\ \mathsf{in}\ f(x, y)\ ;\ \mathsf{assert}\ (!\,x == !\,y)$$

Exercise 70 Prove the following rule for $\mathsf{assert}\ (e) := \mathsf{if}\ e\ \mathsf{then}\ ()\ \mathsf{else}\ \overline{0}\ \overline{0}$: If $\{P\}\, e\, \{v.\, v = \mathsf{true}\}$, then $\{P\}\ \mathsf{assert}\ (e)\ \{v.\, v = ()\}$. •

Lemma 80.

$$\{\mathsf{True}\}\ e_{\mathsf{own}}\ \{\_.\ \mathsf{True}\}$$

Proof Sketch. For the proof of this lemma on paper, we use a typical proof style for Hoare triples. We write the assertions that hold before/after each line in curly braces (and we use the same name for locations that we use for the variables):

$\{\mathsf{True}\}$
$\mathsf{let}\ x := \mathsf{new}(\overline{0})\ \mathsf{in}$
$\{x \mapsto \overline{0}\}$
$\mathsf{let}\ y := \mathsf{new}(\overline{42})\ \mathsf{in}$
$\{x \mapsto \overline{0} * y \mapsto \overline{42}\}$
$f(x, y);$
$\{x \mapsto \overline{42} * y \mapsto \overline{42}\}$
$\mathsf{assert}\ (!\,x == !\,y)$
$\{x \mapsto \overline{42} * y \mapsto \overline{42}\}$
$\{\mathsf{True}\}$

Note that we use the FRAME rule here multiple times: The first time that we use it, we frame the ownership of $x \mapsto \overline{0}$ around the allocation of y. While this use of framing is perhaps not very interesting, the second one is. The second time we use framing, we frame the ownership of $y \mapsto \overline{42}$ around the call to f. How do we know that y is not altered by f? The answer is that f only demands ownership of $x \mapsto \overline{0}$ in its precondition (even though it also takes y as an argument) and, hence, e_{own} can keep the ownership of $y \mapsto \overline{42}$: we can frame it around the function call. Thus, it is not very surprising that the assert afterwards succeeds. □

Exercise 71 Verify the function $\mathrm{swap}(x, y) := \mathsf{let}\ t := !\,x\ \mathsf{in}\ x \leftarrow !\,y\,;\, y \leftarrow t$ by proving

$$\{\ell \mapsto v * r \mapsto w\}\ \mathrm{swap}(\ell, r)\ \{\_.\ \ell \mapsto w * r \mapsto v\}$$

●

Adequacy If we verify functions such as swap or fib, *what* do we actually prove? The following adequacy statement can shed some light on this question:

Statement 81 (Pure Adequacy)**.** *If* $\{\mathsf{True}\}\ e\ \{v.\ \phi(v)\}$ *and* $(e, h) \rightsquigarrow^* (e', h')$*, then* (e', h') *can take a step or* $e' = v$ *for some* v *such that* $\phi(v)$*.*

Intuitively, this statement says that e never gets stuck and if it eventually terminates, then the value satisfies the postcondition. To reason about programs which manipulate the heap, we can use the following strengthened version:

Statement 82 (Adequacy)**.** *If* $\left\{\bigstar_{\ell \mapsto v \in h_{in}} \ell \mapsto v\right\}\ e\ \left\{v.\ \exists h_{out}.\ (\bigstar_{\ell \mapsto w \in h_{out}} \ell \mapsto w) * \phi(v, h_{out})\right\}$ *and* $h \supseteq h_{in}$ *and* $(e, h) \rightsquigarrow^* (e', h')$*, then* (e', h') *can take a step or* $e' = v$ *for some* v *and* $h' \supseteq h_{out}$ *for some* h_{out} *such that* $\phi(v, h_{out})$*.*

Linked List Example Let us turn to our first larger example: verifying a linked list implementation. We define recursively what it means for a value v to represent the list xs:

$$\begin{aligned}
\mathrm{list}(v, \mathsf{nil}) &:= v = \mathsf{None} \\
\mathrm{list}(v, x :: xr) &:= \exists \ell, w.\, v = \mathsf{Some}(\ell) * \ell \mapsto (x, w) * \mathrm{list}(w, xr)
\end{aligned}$$

Here, we use None as notation for $\mathsf{inj}_1\,()$ and Some as notation for inj_2. For this representation of lists, we can define a range of standard list functions:

```
$\mathrm{new}() := \mathsf{None}$
$\mathrm{cons}(a, x) := \mathsf{let}\, r := \mathsf{new}(a, x)\, \mathsf{in}\, \mathsf{Some}(r)$
$\mathrm{head}(x) := \mathsf{match}\ x\ \mathsf{with}$
    $\mid \mathsf{None} \Rightarrow \mathsf{skip}$
    $\mid \mathsf{Some}\ r \Rightarrow \mathsf{let}\, (a, x) := {!\,r}\ \mathsf{in}\ a$
    $\mathsf{end}$
$\mathrm{tail}(x) := \mathsf{match}\ x\ \mathsf{with}$
    $\mid \mathsf{None} \Rightarrow \mathsf{skip}$
    $\mid \mathsf{Some}\ r \Rightarrow \mathsf{let}\, (a, x) := {!\,r}\ \mathsf{in}\ x$
    $\mathsf{end}$
$\mathrm{len}(x) := \mathsf{match}\ x\ \mathsf{with}$
    $\mid \mathsf{None} \Rightarrow \overline{0}$
    $\mid \mathsf{Some}\ r \Rightarrow \mathsf{let}\, (a, x) := {!\,r}\ \mathsf{in}\ \mathrm{len}(x) + \overline{1}$
    $\mathsf{end}$
$\mathrm{app}(x, y) := \mathsf{match}\ x\ \mathsf{with}$
    $\mid \mathsf{None} \Rightarrow y$
    $\mid \mathsf{Some}\ r \Rightarrow \mathsf{let}\, (a, x) := {!\,r}\ \mathsf{in}\ r \leftarrow (a, \mathrm{app}(x, y))\, ;\, \mathsf{Some}\ r$
    $\mathsf{end}$
```

We verify the append function:

Lemma 83. $\{\mathrm{list}(v, xs) * \mathrm{list}(w, ys)\}\ \mathrm{app}(v, w)\ \{u.\, \mathrm{list}(u, xs \mathbin{+\!\!+} ys)\}$

Proof. By induction on xs. In the case $xs = \mathsf{nil}$, we have to show

$$\{v = \mathsf{None} * \mathrm{list}(w, ys)\}\ \mathrm{app}(v, w)\ \{u.\, \mathrm{list}(u, ys)\}$$

In the case $xs = x :: xr$, we have to show

$$\{(\exists \ell, u.\, v = \mathsf{Some}(\ell) * \ell \mapsto (x, u) * \mathrm{list}(u, xr)) * \mathrm{list}(w, ys)\}\ \mathrm{app}(v, w)\ \{u.\, \mathrm{list}(u, x :: (xr \mathbin{+\!\!+} ys))\}$$

given the inductive hypothesis $\forall v.\, \{\mathrm{list}(v, xr) * \mathrm{list}(w, ys)\}\ \mathrm{app}(v, w)\ \{u.\, \mathrm{list}(u, xr \mathbin{+\!\!+} ys)\}$. Both cases follow by prudent application of the rules. □

Exercise 72 Verify the remaining linked list functions:

$$\{\mathsf{True}\}\ \text{new}()\ \{v.\ \text{list}(v, \mathsf{nil})\} \qquad \{\text{list}(v, xs)\}\ \text{cons}(x, v)\ \{u.\ \text{list}(u, x :: xs)\}$$

$$\{\text{list}(v, x :: xs)\}\ \text{head}(v)\ \{w.\ w = x\} \qquad \{\text{list}(v, x :: xs)\}\ \text{tail}(v)\ \{w.\ \text{list}(w, xs)\}$$

$$\{\text{list}(v, xs)\}\ \text{len}(v)\ \left\{w.\ w = \overline{|xs|} * \text{list}(v, xs)\right\}$$

●

Exercise 73 The specification of the tail function can be strengthened. One specification you might come up with is the following:

$$\{\text{list}(v, x :: xs)\}\ \text{tail}(v)\ \{w.\ \text{list}(v, x :: xs) * \text{list}(w, xs)\}$$

Is this specification true? If yes, prove this specification. If not, explain why, try to find another valid specification, and prove it. ●

6 Iris

We have seen how one can verify simple programs with a program logic. Unfortunately, the separation logic from the previous section is quite limited: (1) we have no support for (possibly) infinite loops, (2) we have no support for sharing ownership, (3) we have to do every single proof step manually, (4) and we have no support for concurrency. To verify larger and more complicated programs, we now turn to a much more powerful and usable separation logic: Iris [5].

Iris extends the separation logic from Section 5.2 with a number of connectives:

$$\text{Propositions}\quad P, Q, R ::= \cdots \mid P \mathrel{-\!\!*} Q \mid \mathsf{wp}\, e\, \{v.\, P(v)\} \mid \Box P \mid \boxed{P}^{\mathcal{N}} \mid \triangleright P \mid \mu x.\, P$$

We will now step-by-step introduce these connectives, Iris's automation, and how the connectives help us to verify increasingly complicated programs.

6.1 The Magic Wand

We start with the magic wand $P \mathrel{-\!\!*} Q$. From separation logic, we already know separating conjunction $P_1 * P_2$, which acts like ordinary conjunction $P_1 \wedge Q_2$ in many ways. For example, separating conjunction, like ordinary conjunction, is commutative and associative and one can show that $P_1 * P_2 \vdash P_1 \wedge P_2$. But there is an additional aspect in which they are similar: just like ordinary conjunction $P_1 \wedge P_2$ has an associated notion of implication $P \Rightarrow Q$ in propositional logic, separating conjunction $P_1 * P_2$ has an associated notion of implication in separation logic—the *magic wand* $P \mathrel{-\!\!*} Q$.

Intuitively, the magic wand $P \mathrel{-\!\!*} Q$ expresses that Q holds under the assumption of P. That is, Q holds under the assumption of *ownership* of the heap fragment described by P. There are only two proof rules for $P \mathrel{-\!\!*} Q$ and they are quite simple:

$$\textsc{WandIntro}\quad \frac{P * Q \vdash R}{P \vdash Q \mathrel{-\!\!*} R} \qquad\qquad \textsc{WandElim}\quad \frac{P \vdash Q \mathrel{-\!\!*} R}{P * Q \vdash R}$$

With the addition of the magic wand, we can prove one of the rules of separating conjunction that we introduced as an axiom in Section 5, the rule ExistsSep.

Lemma 84.

ExistsSep
$(\exists x : X.\, P(x)) * Q \vdash \exists x : X.\, P(x) * Q$

Proof. Using WandElim, we can revert the assumption Q from our context, leaving us to prove $(\exists x : X.\, P(x)) \vdash Q \mathrel{-\!\!*} \exists x : X.\, P(x) * Q$. We use ExistElim to eliminate the existential quantifier in our assumptions. We have to show $P(x) \vdash Q \mathrel{-\!\!*} \exists x : X.\, P(x) * Q$ for arbitrary $x : X$. By WandIntro, we have to show $P(x) * Q \vdash \exists x : X.\, P(x) * Q$, which follows by ExistIntro. □

Exercise 74 Prove the following derived rules:

CarryRes
$P \vdash Q \mathrel{-\!\!*} P * Q$

CommPremise
$Q \mathrel{-\!\!*} P \mathrel{-\!\!*} R \vdash P \mathrel{-\!\!*} Q \mathrel{-\!\!*} R$

SepOrDisj2
$(P \vee R) * (Q \vee R) \vdash (P * Q) \vee R$

•

6.2 Iris Proof Mode

To reason about the entailment $P \vdash Q$, we currently use an ever-growing collection of rules about $P \vdash Q$. In particular, we have to use rules to destruct assumptions, instantiate existential quantifiers, use associativity (and commutativity) of the separating conjunction, etc. This is in stark contrast to what Coq and other interactive theorem provers offer their users: In Coq, we have a proof context (not just a single proposition) and we can use tactics to manipulate our proof context and our goal.

With the Iris Proof Mode (IPM) [6], we now turn to similar machinery for Iris that simplifies proving $P \vdash Q$. The IPM introduces an additional context for separation logic propositions and tactic support to manipulate the assumptions therein. To understand what that looks like, let us turn to an example, proving $(P \vee Q) * R \vdash (P * R) \vee (Q * R)$. In the IPM, we start the proof as follows:

```
Lemma or_sep P Q R: (P ∨ Q) ∗ R ⊢ (P ∗ R) ∨ (Q ∗ R).
Proof.
    iIntros "[HPQ HR]".
```

Afterwards, the proof state looks as follows:

```
    P, Q, R : iProp
    -------------------------
    "HPQ": P ∨ Q
    "HR": R
    -------------------------∗
    (P ∗ R) ∨ (Q ∗ R)
```

Above the first line is the normal Coq context containing the propositions P and Q. Below the first line and above the second line is the *spatial* context. It contains two assumptions, `HPQ` and `HR`, which express ownership of $P \vee Q$ and R. Below the second line is our remaining goal $(P * R) \vee (Q * R)$. Similar to ordinary Coq proofs, we can destruct $P \vee Q$:

```
Lemma or_sep P Q R: (P ∨ Q) ∗ R ⊢ (P ∗ R) ∨ (Q ∗ R).
Proof.
    iIntros "[HPQ HR]". iDestruct "HPQ" as "[HP|HQ]".
```

As one might expect, we now have two subgoals to prove:

```
      P, Q, R : iProp                      P, Q, R : iProp
      -------------------------            -------------------------
      "HP": P                              "HQ": Q
      "HR": R                              "HR": R
      -------------------------∗           -------------------------∗
      (P ∗ R) ∨ (Q ∗ R)                    (P ∗ R) ∨ (Q ∗ R)
```

In the first case, we want to pick the left side of the disjunct and in the second case the right side. We do so with the tactics `iLeft` and `iRight`, leaving us to prove:

```
P, Q, R : iProp
-------------------------
"HP": P
"HR": R
-------------------------*
P ∗ R
```

```
P, Q, R : iProp
-------------------------
"HQ": Q
"HR": R
-------------------------*
Q ∗ R
```

In both cases, we can finish the proof with the tactic `iFrame`. `iFrame` will go through the spatial context and search for any propositions that appear in the goal. It will then try to use those to discharge proof obligations in the goal.

The spatial context. One can think of the linear context of the IPM as a large separating conjunction of all the assumptions. That is, since separating conjunction is commutative and associative, we can display the assumptions as an unordered list:

$$\Delta \vdash Q := P_1 * \cdots * P_n \vdash Q \qquad \text{where } \Delta = [P_1, \ldots, P_n]$$

An important distinction between the Coq proof context and the spatial context is that assumptions in the spatial context are *linear*, meaning we can only use them once. That is, if we have the assumption $n > 5$ in our Coq context, then it is *persistent*—it does not change and remains true as we proceed in the proof. In contrast, if we have the assumption $\ell \mapsto \overline{42}$ in our spatial context, then it is only true now—we have to give it up to mutate ℓ and get back a new points to assumption. The reason the spatial context is linear is that $\ell \mapsto \overline{42}$ would be false if we assign $\overline{0}$ to location ℓ. Put differently, the ownership reasoning of separation logic only works if we are required to give up ownership in certain places. Logics like separation logic, where we have to give up propositions when we use them, are called *linear logics*.

Using entailments. Of course, we can also use entailments that we have already proven as part of other proofs. For example, let us return to the proof of $(P \vee Q) * R \vdash (P * R) \vee (Q * R)$. Suppose, for the sake of argument, we want to use the rule OrIntroL directly instead of the tactic `iLeft`. To use OrIntroL, we use the tactic `iPoseProof`. Concretely, in the proof state:

```
P, Q, R : iProp
-------------------------
"HP": P
"HR": R
-------------------------*
(P ∗ R) ∨ (Q ∗ R)
```

the tactic `iPoseProof (or_intro_l (P ∗ R) (Q ∗ R)) as "-#Hor"` brings us to the proof state:

```
P, Q, R : iProp
-------------------------
"HP": P
"HR": R
"Hor": P ∗ R -∗ (P ∗ R) ∨ (Q ∗ R)
-------------------------*
(P ∗ R) ∨ (Q ∗ R)
```

The entailment becomes a magic wand in our proof context. (The reader can ignore the `-#` after the `as`.) Since the magic wand is similar to an implication, we can then apply it to our goal with `iApply "Hor"`.

Coq propositions. Recall that we embed arbitrary Coq propositions into our goals as ϕ. If we want to prove a pure proposition with the IPM, we can use the tactic `iPureIntro`, which will change the goal to allow us to prove the pure proposition (inspired by PURE). To get access to pure propositions in our assumptions (if they are in the linear context and not the Coq context), we can lift them out of the linear context (inspired by FROMPURE). To do so, we destruct the assumption, which we will look at next.

Destructing assumptions. Recall that to destruct assumptions, the IPM offers the tactic `iDestruct`. With `iDestruct`, we can destruct an assumption `H`, say $(P \vee Q) * R$, into its components. Specifically, here we would use `iDestruct "H" as "[[HP|HQ] HR]"` inspired by Coq's destruction patterns. If we want to destruct an existential quantifier $\exists x : X. P(x)$, then the pattern becomes `"[%w HP]"` where `w` is the Coq name the witness will get and `HP` the name the identifier for the assumption $P(w)$ that will be added to the context. If we want to access the contents of an assumption that is a pure proposition (*i.e.*, $n < 5$), then the destruct pattern becomes `"%Hn"` where `Hn` is the Coq name the assumption $n < 5$ will get. The tactic `iIntros` can also be given a destruction pattern.

Specializing assumptions. Recall that we can add entailments as assumptions using the tactic `iPoseProof`. If we have used it to add `"Hor": P * R -* (P * R) \/ (Q * R)` to our spatial context, then we can specialize the assumption of the magic wand. With `iSpecialize ("Hor" with "[HR HP]")`, we create a new proof obligation:

```
P, Q, R : iProp
-------------------------
"HP": P
"HR": R
-------------------------*
P ∗ R
```

We have to show the premise `P ∗ R` from the assumptions we carry over `HR` and `HP`. We can use the machinery behind `iFrame` to directly frame `HR` and `HP` when we specialize the wand. To do so, we execute `iSpecialize ("Hor" with "[$HR $HP]")`.

A more exhaustive overview of the tactics of the IPM can be found at:

https://gitlab.mpi-sws.org/iris/iris/-/blob/master/docs/proof_mode.md

Exercise 75 Prove $\vdash P \twoheadrightarrow Q \twoheadrightarrow P{*}Q$, $P{*}(P \twoheadrightarrow Q) \vdash Q$, and $P{\vee}Q \vdash R \twoheadrightarrow (P{*}R){\vee}(Q{*}R)$ with and without the IPM. •

6.3 Weakest Preconditions

The IPM is very convenient for reasoning about the entailments $P \vdash Q$. But to prove Hoare triples, we still have to leave the IPM and use our collection of lemmas manually. With the *weakest precondition*, we will now see how we can reuse the IPM to prove Hoare triples. In Iris, we define Hoare triples $\{P\}\, e\, \{v.\, Q(v)\}$ as an entailment:

$$P \vdash \mathsf{wp}\; e\; \{v.\, Q(v)\}$$

where $\mathsf{wp}\ e\ \{v.\, Q(v)\}$ is the weakest precondition that we have to impose such that e never gets stuck and (if it terminates) results in a value satisfying the postcondition $Q(v)$.

Proving a Hoare triple $\{P\}\, e\, \{v.\, Q(v)\}$ by showing that P implies the weakest precondition of e is not specific to Iris. However, in the case of Iris, it is important to note that we prove an entailment between two *separation logic propositions*—the weakest precondition is a separation logic assertion! Thus, we can reuse the Iris proof mode.

Weakest Precondition Rules $\boxed{\mathsf{wp}\ e\ \{v.\, Q(v)\}}$

Structural Rules

Value
$$Q(v) \vdash \mathsf{wp}\ v\ \{w.\, Q(w)\}$$

Wand
$$(\forall v.\, Q(v) \mathrel{-\!\!*} Q'(v)) * \mathsf{wp}\ e\ \{w.\, Q(w)\} \vdash \mathsf{wp}\ e\ \{w.\, Q'(w)\}$$

WpBind
$$\mathsf{wp}\ e\ \{v.\, \mathsf{wp}\ K[v]\ \{w.\, Q(w)\}\} \vdash \mathsf{wp}\ K[e]\ \{w.\, Q(w)\}$$

PureStep
$$\frac{e \rightarrow_{\mathsf{pure}} e'}{\mathsf{wp}\ e'\ \{v.\, Q(v)\} \vdash \mathsf{wp}\ e\ \{v.\, Q(v)\}}$$

Heap Command Rules

New
$$(\forall \ell.\, \ell \mapsto v \mathrel{-\!\!*} Q(\ell)) \vdash \mathsf{wp}\ \mathsf{new}(v)\ \{w.\, Q(w)\}$$

Load
$$\ell \mapsto v * (\ell \mapsto v \mathrel{-\!\!*} Q(v)) \vdash \mathsf{wp}\ !\,\ell\ \{w.\, Q(w)\}$$

Store
$$\ell \mapsto v * (\ell \mapsto w \mathrel{-\!\!*} Q()) \vdash \mathsf{wp}\ \ell \leftarrow w\ \{u.\, Q(u)\}$$

The structural rules should not come as a big surprise. Let us take a look at the general rules. The rule Wand corresponds to Consequence and the rule WpBind to the rule Bind. But what about all the other rules for Hoare triples (*e.g.*, Exists,Pure, and Frame)? They can be derived from the rules above.

Exercise 76 (The Frame Rule) Using the rules for the weakest precondition, prove the Frame rule.

Frame
$$\frac{\{P\}\, e\, \{v.\, Q\}}{\{P * R\}\, e\, \{v.\, Q * R\}}$$

Hint: You may find $R \vdash Q \mathrel{-\!\!*} Q * R$ useful. •

Exercise 77 (The Heap Rules) Using the rules for the weakest precondition, reprove the structural rules for state manipulating expressions (*i.e.*, New, Load, and Store). •

Iris proof mode. In the IPM, we have several specialized tactics to use the weakest precondition rules: The first one is `wp_bind e` where `e` is the expression in the evaluation context that we want to use the rule WpBind on. The second one is `wp_pure e` where `e` is the expression (potentially inside an evaluation context) that we want to execute a pure step of. We can omit `e` and just write `_` instead, or just use the tactic: `wp_pures`, which will execute as many pure steps as possible. Finally, for non-pure reduction rules (*i.e.*, New, Load, and Store), we have the tactics `wp_alloc l as "H"` to allocate a fresh reference `l` with the new assertion `H`, `wp_load` to execute a load, and `wp_store` to execute a store.

Case Study: Linked Lists We return to the linked list example from Section 5. Recall the definition of the append function:

$$\begin{aligned}
\mathrm{app}(x, y) := {} & \mathsf{match}\ x\ \mathsf{with} \\
& \mid \mathsf{None} \Rightarrow y \\
& \mid \mathsf{Some}\ r \Rightarrow \mathsf{let}\ (a, x) := {!}\,r\ \mathsf{in}\ r \leftarrow (a, \mathrm{app}(x, y))\ ;\ \mathsf{Some}\ r \\
& \mathsf{end}
\end{aligned}$$

Once again, we prove its correctness—this time in Iris!

Lemma 85. $\{\mathrm{list}(v, xs) * \mathrm{list}(w, ys)\}\ \mathsf{app}(v, w)\ \{u.\, \mathrm{list}(u, xs +\!\!+ ys)\}$

Proof. By induction on xs.

a) Let $xs = \mathsf{nil}$. We show $\{\mathrm{list}(v, \mathsf{nil}) * \mathrm{list}(w, ys)\}\ \mathsf{app}(v, w)\ \{u.\, \mathrm{list}(u, \mathsf{nil} +\!\!+ ys)\}$. Step-by-step, we prove the triple in Iris (if the postcondition of a weakest pre does not change, just the code, we omit it):

Context:	Goal:
$\mathrm{list}(v, \mathsf{nil}) * \mathrm{list}(w, ys)$	$\mathsf{wp}\ \mathrm{app}(v, w)\ \{u.\, \mathrm{list}(u, \mathsf{nil} +\!\!+ ys)\}$
$v = \mathsf{None} * \mathrm{list}(w, ys)$	$\mathrm{app}(v, w)$
$\mathrm{list}(w, ys)$	$\mathrm{app}(\mathsf{None}, w)$
$\mathrm{list}(w, ys)$	$\mathsf{match}\ \mathsf{None}\ \mathsf{with}$ $\mid \mathsf{None} \Rightarrow w$ $\mid \mathsf{Some}\ r \Rightarrow \mathsf{let}\ (a, x) := {!}\,r\ \mathsf{in}\ r \leftarrow (a, \mathrm{app}(x, y))\ ;\ \mathsf{Some}\ r$ end
$\mathrm{list}(w, ys)$	w

From $\mathrm{list}(w, ys)$, we can deduce our postcondition $\mathrm{list}(w, \mathsf{nil} +\!\!+ ys)$.

b) Let $xs = x :: xr$. We show $\{\mathrm{list}(v, x :: xr) * \mathrm{list}(w, ys)\}\ \mathrm{app}(v, w)\ \{u.\, \mathrm{list}(u, (x :: xr) +\!\!+ ys)\}$. Step-by-step, we prove the triple in Iris:

Context:	Goal:
$\mathrm{list}(v, x :: xr) * \mathrm{list}(w, ys)$	$\mathsf{wp}\ \mathrm{app}(v, w)\ \{u.\, \mathrm{list}(u, (x :: xr) +\!\!+ ys)\}$
$(\exists r, n.\, v = \mathsf{Some}(r) * r \mapsto (x, n) * \mathrm{list}(n, xr))$ $* \mathrm{list}(w, ys)$	$\mathsf{app}(v, w)$
$r \mapsto (x, n) * \mathrm{list}(n, xr) * \mathrm{list}(w, ys)$	$\mathsf{app}(\mathsf{Some}(r), w)$
$r \mapsto (x, n) * \mathrm{list}(n, xr) * \mathrm{list}(w, ys)$	$\mathsf{let}\ (x, n) := {!}\,r\ \mathsf{in}\ r \leftarrow (x, \mathrm{app}(n, w))\ ;\ \mathsf{Some}\ r$
$r \mapsto (x, n) * \mathrm{list}(n, xr) * \mathrm{list}(w, ys)$	$r \leftarrow (x, \mathrm{app}(n, w))\ ;\ \mathsf{Some}\ r$
By induction for xs, binding $\mathrm{app}(n, w)$, and framing $r \mapsto (x, n)$:	
$r \mapsto (x, n) * \mathrm{list}(u, xr +\!\!+ ys)$	$r \leftarrow (x, u)\ ;\ \mathsf{Some}\ r$
$r \mapsto (x, u) * \mathrm{list}(u, xr +\!\!+ ys)$	$\mathsf{Some}\ r$
$\mathrm{list}(\mathsf{Some}\ r, x :: (xr +\!\!+ ys))$	$\mathsf{Some}\ r$

From $\mathrm{list}(\mathsf{Some}\ r, x :: (xr +\!\!+ ys))$, we can deduce $\mathrm{list}(\mathsf{Some}\ r, (x :: xr) +\!\!+ ys)$.

$\square$

Exercise 78 Verify the remaining linked list functions in Iris:

$$\{\mathsf{True}\}\ \mathrm{new}()\ \{v.\ \mathrm{list}(v, \mathsf{nil})\} \qquad \{\mathrm{list}(v, xs)\}\ \mathrm{cons}(x, v)\ \{u.\ \mathrm{list}(u, x :: xs)\}$$

$$\{\mathrm{list}(v, x :: xs)\}\ \mathrm{head}(v)\ \{w.\ w = x\} \qquad \{\mathrm{list}(v, x :: xs)\}\ \mathrm{tail}(v)\ \{w.\ \mathrm{list}(w, xs)\}$$

$$\{\mathrm{list}(v, xs)\}\ \mathrm{len}(v)\ \left\{w.\ w = \overline{|xs|} * \mathrm{list}(v, xs)\right\}$$

●

Exercise 79 We can write down a function lookup for linked lists that returns a reference to a link of the list given its index:

```
lookup(l, i) := match l with
                | None ⇒ None
                | Some l ⇒ if i == 0̄ then Some l else lookup(π₂ ! l, i − 1̄)
                end
```

We can give a specification (for the case that the index is in range) that allows mutation of that link of the linked list using a specification pattern commonly known as the *magic wand encoding* (which is often used to give access to parts of a data structure).

$$\begin{array}{l}\{\mathrm{list}(v, xs) * i < |xs|\}\\ \quad \mathrm{lookup}(v, \bar{i})\\ \{w.\ \exists \ell, n.\ w = \mathsf{Some}\ \ell * \ell \mapsto (xs[i], n) * (\forall u.\ \ell \mapsto (u, n) \twoheadrightarrow \mathrm{list}(v, xs[i \mapsto u]))\}\end{array}$$

The postcondition gives ownership of the location ℓ returned by the function, and logically ownership of the whole linked list can be restored once ownership of ℓ is given up again.

Verify this specification. ●

6.4 Invariants and Persistency

Recall the MUTBIT example:

$$\begin{array}{rl}\mathrm{MUTBIT} := & \{\mathrm{flip} : \mathbf{1} \to \mathbf{1},\ \mathrm{get} : \mathbf{1} \to \mathsf{bool}\}\\ \mathrm{MyMutBit} := & \mathsf{let}\ x = \mathsf{new}\ \bar{0}\\ & \mathsf{in}\ \{\mathrm{flip} := \lambda y.\ x \leftarrow \bar{1} - !\,x,\\ & \quad \mathrm{get} := \lambda y.\ !\,x > 0\}\end{array}$$

In Section 4, we proved that MyMutBit is semantically safe (*i.e.*, $\vDash$ MyMutBit : MUTBIT). Let us try to verify MyMutBit in Iris (records are essentially just pairs with named projections). Instead of using types to specify the function, in Iris, we verify MyMutBit by proving a Hoare triple:

$$\{\mathsf{True}\}\ \mathrm{MyMutBit}\ \{v.\ \mathrm{MutBit}(v)\}$$

We just have to settle on a predicate MutBit. Intuitively, to match the type specification, we want to say something like:

$$\text{MutBit}(v) := \{\mathsf{True}\}\, v.\text{flip}()\, \{w.\, w = ()\} * \{\mathsf{True}\}\, v.\text{get}()\, \{w.\, w \in \mathbb{B}\}$$

For the flip component, we want the result to be unit, and for get we want to obtain some boolean. The types do not specify any input, so we pick the precondition True for both.

The specification MutBit poses two problems: First, Hoare triples are defined as entailments $P \vdash \mathsf{wp}\, e\, \{v.\, Q(v)\}$, which is not an Iris proposition itself. So if we embed them as a meta level proposition into our specification, we lose all information about the current program state (*e.g.*, the location x). Second, the preconditions do not make the location x available to flip and get. How can we prove the two triples without ownership of x?

Internalizing Hoare triples. Let us focus on the first problem—Hoare triples are not Iris propositions. Since they are defined using the entailment and is something like an implication, we could try do define $\{P\}\, e\, \{v.\, Q(v)\} := P \twoheadrightarrow \mathsf{wp}\, e\, \{v.\, Q(v)\}$ as an Iris proposition. Then, when we prove $\{P\}\, e\, \{v.\, Q(v)\}$, we can keep all of the ownership we currently have (*e.g.*, ownership of x after allocation). Unfortunately, with this definition, ownership of a triple $\{P\}\, e\, \{v.\, Q(v)\}$ would mean we could only use it once! So in the case of flip, we could only do a single flip and for any subsequent calls to flip, we would have a problem.

What we really want is that Hoare triples are duplicable:

$$\{P\}\, e\, \{v.\, Q(v)\} \vdash \{P\}\, e\, \{v.\, Q(v)\} * \{P\}\, e\, \{v.\, Q(v)\}$$

Then, just like with Coq propositions, we can always keep a copy whenever we want to use the Hoare triple. In Iris, we call propositions which can be duplicated (*i.e.*, $P \vdash P * P$) *persistent*, and we have a modality $\Box\, P$, which "makes" propositions persistent:

PersDup
$$\Box\, P \vdash (\Box\, P) * (\Box\, P)$$

PersElim
$$\Box\, P \vdash P$$

Thus, we define Hoare triples as $\{P\}\, e\, \{v.\, Q(v)\} := \Box(P \twoheadrightarrow \mathsf{wp}\, e\, \{v.\, Q(v)\})$.

Proving persistent propositions. Given the duplicable definition of Hoare triples, a new question arises: how do we *prove* a persistent proposition $\Box\, P$? The two rules we have seen so far allow us to eliminate $\Box\, P$, but no rule allows us to introduce $\Box\, P$ yet. To do so, we use the following rules for interacting with persistent propositions:

PersMono
$$\frac{P \vdash Q}{\Box\, P \vdash \Box\, Q}$$

PersPure
$$\phi \vdash \Box\, \phi$$

PersAndSep
$$(\Box\, P) \land Q \vdash (\Box\, P) * Q$$

PersIdemp
$$\Box\, P \vdash \Box\, \Box\, P$$

PersAll
$$\forall x : X.\, \Box\, P(x) \vdash \Box\, \forall x : X.\, P(x)$$

PersExists
$$\Box\, \exists x : X.\, P(x) \vdash \exists x : X.\, \Box\, P(x)$$

In the IPM, persistent propositions play a special role: they have their own context. That is, since persistent propositions are duplicable, they do not have to be given up when we use them. Thus, they reside in their own context. For example, if we start a proof with:

```
Lemma double_int f :
    {True} f () {v, ∃ z: Z, v = #z } ⊢ {True} f () + f () {v, ∃ z: Z, v = #z }
Proof.
    iIntros "#Hf".
```

then the resulting proof state is:

```
    f: val
    -------------------------
    "Hf": {True} f () {v, ∃ z: Z, v = #z }
    -------------------------□
    -------------------------*
    {True} f () + f () {v, ∃ z: Z, v = #z }
```

The `#` moves the assumption into the *persistent context*, which is right above the spatial context. All the propositions in the persistent context must be underneath a box. We can then get rid of the $\square$ in our goal (behind the definition of Hoare triples) with the tactic `iModIntro`, leaving us to prove

```
    True -* WP f () {v, ∃ z: Z, v = #z }
```

Note that when we get rid of the $\square$ in the goal, the proof mode implicitly applies PersMono. PersMono requires that all the assumptions we want to take with us are underneath a $\square$, so we can keep the persistent context but lose the spatial context.

Exercise 80 The rule PersDup can be derived from the more general (and less intuitive) rule PersAndSep. Derive PersDup from PersAndSep without the IPM. ●

Exercise 81 Given Hoare triples $\{P\}\, e\, \{v.\, Q(v)\} := \square(P \twoheadrightarrow \mathsf{wp}\, e\, \{v.\, Q(v)\})$ defined as Iris propositions, suppose we define $\{P\}\, e\, \{v.\, Q(v)\}_{\mathsf{ext}} := \vdash \{P\}\, e\, \{v.\, Q(v)\}$. Reprove all the Hoare rules we have discussed so far for $\{P\}\, e\, \{v.\, Q(v)\}_{\mathsf{ext}}$. ●

Invariants. Now that we have internalized Hoare triples, let us return to the verification of MyMutBit. After allocating the reference x, we have to prove:

$$x \mapsto \overline{0} \vdash \{\mathsf{True}\}\, \mathit{bit}.\mathrm{flip}()\, \{w.\, w = ()\} * \{\mathsf{True}\}\, \mathit{bit}.\mathrm{get}()\, \{w.\, w \in \mathbb{B}\}$$
$$\text{where } \mathit{bit} = \{\mathrm{flip} := \lambda y.\, x \leftarrow \overline{1} - !\, x,\ \ \mathrm{get} := \lambda y.\ !\, x > 0\}$$

How are we supposed to decide whether to give ownership of $x \mapsto \overline{0}$ to flip or get? Moreover, if we pick one side, then we subsequently have to prove a persistent proposition, but the proposition $x \mapsto \overline{0}$ is not persistent, so we cannot keep it!

Here, invariants $\boxed{P}^{\mathcal{N}}$ come in. With invariants, we can make the ownership of x duplicable. The price we have to pay (to retain a sound logic) is that we have to agree on a "protocol" how x will be used. Concretely, in the case of MyMutBit, we know that x will always be either 0 or 1. That is what we choose as the invariant:

$$I_{\mathrm{MyMutBit}} := \boxed{\exists n \in \{0, 1\}.\, x \mapsto \overline{n}}^{\mathcal{N}}$$

To understand how invariants work and why they help us here, we consider their rules:

InvPers
$$\boxed{F}^{\mathcal{N}} \vdash \Box\,\boxed{F}^{\mathcal{N}}$$

InvAlloc
$$\frac{P * \boxed{F}^{\mathcal{N}} \vdash \mathsf{wp}^{\mathcal{E}}\; e\;\{v.\, Q(v)\}}{P * F \vdash \mathsf{wp}^{\mathcal{E}}\; e\;\{v.\, Q(v)\}}$$

InvOpen
$$\frac{P * F \vdash \mathsf{wp}^{\mathcal{E}\setminus\mathcal{N}}\; e\;\{v.\, F * Q(v)\} \qquad \mathcal{N} \subseteq \mathcal{E}}{P * \boxed{F}^{\mathcal{N}} \vdash \mathsf{wp}^{\mathcal{E}}\; e\;\{v.\, Q(v)\}}$$

The rule InvPers says that invariants are persistent and, hence, we can duplicate them and share them with separate parts of our proof. The propositions we can put into invariants are from the restricted fragment of *state propositions*:

$$\text{State Proposition}\;\; F, G ::= \phi \;\mid\; \ell \mapsto v \;\mid\; \exists x : X.\, F(x) \;\mid\; \forall x : X.\, F(x) \;\mid\; F \vee G \;\mid\; F * G$$

Most importantly, the state propositions can neither contain invariants themselves nor weakest preconditions. (We will see a strengthened version, called impredicative invariants, in Section 6.5.)

The rule InvAlloc says that we can allocate the invariant F if we are willing to give up ownership of F. Thereafter, we can freely share ownership of F by sharing the invariant $\boxed{F}^{\mathcal{N}}$.

The rule InvOpen is the most interesting rule. It says that if we own the invariant $\boxed{F}^{\mathcal{N}}$, then we can get access to F. In exchange, we have to prove F again in our postcondition. The reason is that other program parts could rely on the invariant being true when they are executed. For example, both flip and get will rely on the invariant I_{MutBit} and, hence, they have to ensure that it holds again after their execution.

The rule InvOpen shows an additional bit of bookkeeping: we may not open the same invariant multiple times. For example, if we have the invariant MutBit, then opening it twice would be fatal: we could use PointstoSep to prove anything. To ensure we do not open the same invariant multiple times, invariants $\boxed{F}^{\mathcal{N}}$ have a so-called namespace $\mathcal{N}$ associated with them and weakest preconditions have masks $\mathcal{E}$. Whenever we open an invariant, we must make sure that it is not already opened by proving that the namespace $\mathcal{N}$ is still contained in the mask. Subsequently, the namespace is removed from the mask until the invariant is closed again (*i.e.*, in the postcondition). If we omit the mask from a weakest precondition (as we did above), we mean the full mask $\top$. All the rules we have seen so far for the weakest precondition are true for arbitrary masks.

So let us return to the proof of MyMutBit:

Lemma 86.
$$\{\mathsf{True}\}\ \text{MyMutBit}\ \{v.\, \mathit{MutBit}(v)\}$$

Proof.

Context:	Goal:
	$\mathsf{wp}\ \mathrm{MyMutBit}\ \{v.\, \mathrm{MutBit}(v)\}$
	$\mathsf{let}\ x = \mathsf{new}\ \overline{0}\ \mathsf{in}$ $\{\mathrm{flip} := \lambda y.\, x \leftarrow \overline{1} - !\,x,\ \ \mathrm{get} := \lambda y.\ !\,x > 0\}$
$\ell \mapsto \overline{0}$	$\{\mathrm{flip} := \lambda y.\, \ell \leftarrow \overline{1} - !\,\ell,\ \ \mathrm{get} := \lambda y.\ !\,\ell > 0\}$
$\ell \mapsto \overline{0} \vee \ell \mapsto \overline{1}$	$\{\mathrm{flip} := \lambda y.\, \ell \leftarrow \overline{1} - !\,\ell,\ \ \mathrm{get} := \lambda y.\ !\,\ell > 0\}$
We allocate $\ell \mapsto \overline{0} \vee \ell \mapsto \overline{1}$ as an invariant.	
I_{MutBit}	$\{\mathrm{flip} := \lambda y.\, \ell \leftarrow \overline{1} - !\,\ell,\ \ \mathrm{get} := \lambda y.\ !\,\ell > 0\}$
Let $bit := \{\mathrm{flip} := \lambda y.\, \ell \leftarrow \overline{1} - !\,\ell,\ \ \mathrm{get} := \lambda y.\ !\,\ell > 0\}$.	
I_{MutBit}	$\mathrm{MutBit}(bit)$

Case flip.

I_{MutBit}	$\mathsf{wp}\ bit.\mathrm{flip}()\ \{w.\, w = ()\}$
I_{MutBit}	$\ell \leftarrow \overline{1} - !\,\ell$
We open the invariant I_{MutBit}.	
$n \in \{0,1\} * \ell \mapsto \overline{n}$	$\mathsf{wp}^{\top \setminus \mathcal{N}}\ \ell \leftarrow \overline{1} - !\,\ell\ \{w.\, (\exists m \in \{0,1\}\,.\, \ell \mapsto \overline{m}) * w = ()\}$
$n \in \{0,1\} * \ell \mapsto \overline{n}$	$\ell \leftarrow \overline{1} - \overline{n}$
$n \in \{0,1\} * \ell \mapsto \overline{n}$	$\ell \leftarrow \overline{1 - n}$
$n \in \{0,1\} * \ell \mapsto \overline{1 - n}$	$()$
Since $n = 0 \vee n = 1$, we have $1 - n = 0 \vee 1 - n = 1$. Thus, $\exists m \in \{0,1\}\,.\, \ell \mapsto \overline{m}$.	

Case get.

I_{MutBit}	$\mathsf{wp}\ bit.\mathrm{get}()\ \{w.\, w \in \mathbb{B}\}$
I_{MutBit}	$!\,\ell > 0$
We open the invariant I_{MutBit}.	
$n \in \{0,1\} * \ell \mapsto \overline{n}$	$\mathsf{wp}^{\top \setminus \mathcal{N}}\ !\,\ell > 0\ \{w.\, (\exists m \in \{0,1\}\,.\, \ell \mapsto \overline{m}) * w \in \mathbb{B}\}$
$n \in \{0,1\} * \ell \mapsto \overline{n}$	$\overline{n} > 0$
$n \in \{0,1\} * \ell \mapsto \overline{n} * b \in \mathbb{B}$	b
The postcondition holds for b.	

□

Exercise 82 (Abstract integers) We define an ADT which represents an integer as two non-negative numbers:

$$\begin{aligned}\mathrm{MyInt}(z) := \ & \mathsf{let}\ x = \mathsf{new}\ \mathsf{if}\ \overline{0} < z\ \mathsf{then}\ (\overline{0}, z)\ \mathsf{else}\ (-z, \overline{0})\\ & \mathsf{in}\ \{\mathrm{get} := \lambda y.\, \mathsf{let}\ z = !\,x\ \mathsf{in}\ \mathsf{assert}\ (\overline{0} \le \pi_1 z)\ ;\ \mathsf{assert}\ (\overline{0} \le \pi_2 z)\ ;\ \pi_2 z - \pi_1 z,\\ & \quad \mathrm{flip} := \lambda y.\, \mathsf{let}\ z = !\,x\ \mathsf{in}\ (\pi_2 z, \pi_1 z)\}\end{aligned}$$

Prove the following specification:

$$\{\mathsf{True}\}\ \mathrm{MyInt}(\overline{z})\ \{v.\, \mathrm{FlipInt}(v)\}$$

where $\mathrm{FlipInt}(v) := \{\mathsf{True}\}\ v.\mathrm{get}()\ \{w.\, \exists z.\, w = \overline{z}\} * \{\mathsf{True}\}\ v.\mathrm{flip}()\ \{w.\, w = ()\}$. •

6.5 Step-indexing

Up to now, all examples that we have discussed are terminating. But what if we want to verify a potentially non-terminating program (*e.g.*, a parameterized function, a potentially diverging loop, or the Z-combinator from Section 3)? We are now going to discuss how *step-indexing* can help us—just like with the Z-combinator—to prove properties of potentially diverging programs.

To keep matters concrete, we will focus on a specific example: deriving a recursion rule for our built-in recursive functions $\mathsf{fix}\, f\, x.\, e$. We already have a rule to execute a single step of our recursive functions with PureStep. However, if we use that rule for $(\mathsf{fix}\, f\, x.\, e)\, v$, then we do not obtain any assumptions about the recursive occurrences of f in e. However, in a *partial correctness logic* like Iris, where we do not prove termination[3], we can assume for any recursive call the specification is already true:

Rec
$$\frac{P(v) * \forall u.\, \{P(u)\}\ (\mathsf{fix}\, f\, x.\, e)\, u\ \{w.\, Q(u,w)\} \vdash \mathsf{wp}\ e[(\mathsf{fix}\, f\, x.\, e)/f, v/x]\ \{w.\, Q(v,w)\}}{P(v) \vdash \mathsf{wp}\ (\mathsf{fix}\, f\, x.\, e)\, v\ \{w.\, Q(v,w)\}}$$

Logical step-indexing In Section 3, we have encountered step-indexing as a technique to define logical relations and a way to give cyclic proofs for programs in the relation. We will now discuss the *logical* version of step-indexing where all the step-index manipulation is hidden behind a modality, the *later modality* $\triangleright P$. As we will see in subsequent sections, the model of our propositions is step-indexed, meaning they are modelled as predicates over natural numbers, and the later modality makes sure that the step-index decreases.

LaterIntro
$$P \vdash \triangleright P$$

LaterMono
$$\frac{P \vdash Q}{\triangleright P \vdash \triangleright Q}$$

Loeb
$$\frac{\triangleright P \vdash P}{\vdash P}$$

LaterSep
$$\triangleright(P * Q) \dashv\vdash \triangleright P * \triangleright Q$$

LaterExists
$$\frac{X \text{ non-empty}}{\triangleright(\exists x : X.\, P(x)) \dashv\vdash \exists x : X.\, \triangleright P(x)}$$

LaterAll
$$\triangleright(\forall x : X.\, P(x)) \dashv\vdash \forall x : X.\, \triangleright P(x)$$

LaterPers
$$\triangleright \Box P \dashv\vdash \Box \triangleright P$$

LaterPureStep
$$\frac{e \to_{\mathsf{pure}} e'}{\triangleright \mathsf{wp}\ e'\ \{v.\, P(v)\} \vdash \mathsf{wp}\ e\ \{v.\, P(v)\}}$$

LaterNew
$$\triangleright(\forall \ell.\, \ell \mapsto v \mathrel{-\!\!*} Q(\ell)) \vdash \mathsf{wp}\ \mathsf{new}(v)\ \{w.\, Q(w)\}$$

LaterLoad
$$\ell \mapsto v * \triangleright(\ell \mapsto v \mathrel{-\!\!*} Q(v)) \vdash \mathsf{wp}\ !\,\ell\ \{w.\, Q(w)\}$$

LaterStore
$$\ell \mapsto v * \triangleright(\ell \mapsto w \mathrel{-\!\!*} Q()) \vdash \mathsf{wp}\ \ell \leftarrow w\ \{w.\, Q(w)\}$$

The introduction rule LaterIntro corresponds to *downward closure*—we can always

[3]Instead, we only prove that the program does not get stuck and *if it terminates*, that then the postcondition holds.

decrease the step-index of our assumptions and nothing goes wrong. The monotonicity rule LaterMono forces us to always decrease the step-index in the goal when we want to decrease it in our assumptions—it ensures the step-index in our assumptions matches the one in the goal. The induction rule Loeb corresponds to Löb induction. It says that if we want to prove P, then we get to assume that P already holds *later* (*i.e.*, for smaller step-indices, so after we have taken steps).

The rules LaterPureStep, LaterNew, LaterLoad, and LaterStore all execute a step of the program. For step-indexed logical relations, executing a step decreases the step-index. For logical step-indexing, executing a step allows us to add a later in front of our goal (if we think about the rules being applied transitively). Thus, we can eliminate laters from our assumptions. Essentially, we decrease the step-index everywhere.

Exercise 83 The existing rules that we have for program execution are strictly weaker than the new rules involving later. Derive PureStep, New, Load, and Store from the new rules. •

Exercise 84 Prove the following commuting rules:

LaterAnd
$$\triangleright(P \wedge Q) \dashv\vdash \triangleright P \wedge \triangleright Q$$

LaterOr
$$\triangleright(P \vee Q) \dashv\vdash \triangleright P \vee \triangleright Q$$

Hint: Disjunction can be represented as existential quantification, and conjunction as universal quantification. •

Recursive Functions and Definitions Let us return to the recursion rule that we set out to prove in the beginning. We now have everything in place to prove the rule.

Lemma 87.

Rec
$$\frac{\forall v.\, P(v) * \forall u.\, \{P(u)\}\, (\mathsf{fix}\, f\, x.\, e)\, u\, \{w.\, Q(u,w)\} \vdash \mathsf{wp}\; e[(\mathsf{fix}\, f\, x.\, e)/f, v/x]\; \{w.\, Q(v,w)\}}{P(v) \vdash \mathsf{wp}\; (\mathsf{fix}\, f\, x.\, e)\, v\; \{w.\, Q(v,w)\}}$$

Proof. It suffices to prove $\vdash \forall v.\, \{P(v)\}\, (\mathsf{fix}\, f\, x.\, e)\, v\, \{w.\, Q(v,w)\}$.

Context:	Goal:
	$\forall v.\, \{P(v)\}\, (\mathsf{fix}\, f\, x.\, e)\, v\, \{w.\, Q(v,w)\}$
By Loeb induction:	
$\triangleright \forall u.\, \{P(u)\}\, (\mathsf{fix}\, f\, x.\, e)\, u\, \{w.\, Q(u,w)\}$	$\forall v.\, \{P(v)\}\, (\mathsf{fix}\, f\, x.\, e)\, v\, \{w.\, Q(v,w)\}$
We introduce the $\square$ and the precondition.	
$P(v) * \triangleright \forall u.\, \{P(u)\}\, (\mathsf{fix}\, f\, x.\, e)\, u\, \{w.\, Q(u,w)\}$	$\mathsf{wp}\; (\mathsf{fix}\, f\, x.\, e)\, v\; \{w.\, Q(v,w)\}$
By LaterIntro and LaterSep.	
$\triangleright(P(v) * \forall u.\, \{P(u)\}\, (\mathsf{fix}\, f\, x.\, e)\, u\, \{w.\, Q(u,w)\})$	$\mathsf{wp}\; (\mathsf{fix}\, f\, x.\, e)\, v\; \{w.\, Q(v,w)\}$
With LaterPureStep.	
$\triangleright(P(v) * \forall u.\, \{P(u)\}\, (\mathsf{fix}\, f\, x.\, e)\, u\, \{w.\, Q(u,w)\})$	$\triangleright\, \mathsf{wp}\; e[(\mathsf{fix}\, f\, x.\, e)/f, v/x]\; \{w.\, Q(v,w)\}$
By LaterMono.	
$P(v) * \forall u.\, \{P(u)\}\, (\mathsf{fix}\, f\, x.\, e)\, u\, \{w.\, Q(u,w)\}$	$\mathsf{wp}\; e[(\mathsf{fix}\, f\, x.\, e)/f, v/x]\; \{w.\, Q(v,w)\}$
Follows from our assumption.	

□

Löb induction in logical step-indexing—similar to the explicitly step-indexed model—corresponds to strong induction on the step-index. And just like we can (additionally) introduce definitions recursively on the step-index in an explicitly step-indexed model, we can introduce recursive definitions in logical step-indexing. To do so, we use Iris's fixpoint combinator $\mu x. P$. For example, we can define that an expression has an infinite, pure execution as the predicate:

$$\text{inf} := \mu \text{inf}. \lambda e. \exists e'. e \rightarrow_{\mathsf{pure}} e' * \triangleright \text{inf } e'$$

A recursive definition $\mu x. P$ is well-formed if all occurrences of x are guarded by a later modality (just like in the case of inf).

We can convince ourselves that the definition of inf is sensible by verifying that $\text{inf}(\Omega)$: the proof is by Löb induction.

Exercise 85 Prove that $\text{inf}(e) \vdash \mathsf{wp}\ e\ \{\_. \mathsf{False}\}$. •

Step-Indexing in the IPM In the Iris proof mode, we do not have to do all of the above proof steps manually. Instead, we can interact with step-indexing in the following way:

a) If some of our assumptions are guarded by a later, then the `iDestruct` tactic will automatically apply the commuting properties of the later modality (*e.g.*, LaterSep, LaterOr, and LaterExists) when we destruct the assumption.

b) If our goal is underneath a later, the tactic `iNext` will introduce the later and strip a later from all of the assumptions in the Iris context. Technically, it will do so in an unconventional way: Similar to the proof of the fixpoint combinator above, it will add a later in front of all the assumptions that are currently not guarded by a later (using LaterIntro). Subsequently, it (implicitly) moves the later to the very outside of the separating conjunction that is the spatial context. Finally, it uses monotonicity (*i.e.*, LaterMono) to get rid of the later around the spatial context and the later in the goal.

c) Löb induction can be used with the tactic `iLöb as "IH"`, which will make the current goal available as `IH` guarded by a later. The IPM will automatically revert all the propositions in the spatial context such that (a variant of) the Loeb rule becomes applicable.

Exercise 86 Prove the same specification for the Z-combinator.Recall: for a fixed expression e, the Z-combinator is defined as

$$Z := \lambda x.\, g\, g\, x$$
$$g := \lambda r.\ \mathsf{let}\ f = \lambda x.\, r\, r\, x\ \mathsf{in}\ \lambda x.\, e$$

Formally, prove:

$$\frac{\forall v.\, P(v) * \forall u.\, \{P(u)\}\, Z\, u\, \{w.\, Q(u,w)\} \vdash \mathsf{wp}\ e[Z/f, v/x]\ \{w.\, Q(v,w)\}}{P(v) \vdash \mathsf{wp}\ Z\, v\ \{w.\, Q(v,w)\}}$$

•

Higher-order state Let us return the challenging example from Section 4, *Landin's knot* [?], which represents a sound way to encode recursion using state, but which is not in our step-indexed logical relation due to the use of higher-order state:

$$\begin{aligned}\text{knot} := \lambda f.\, &\mathsf{let}\ x = \mathsf{new}\ \lambda x.\, \bar{0}\ \mathsf{in} \\ &\mathsf{let}\ g = \lambda z.\, f\ (\lambda y.\, (!\, x)\ y)\ z\ \mathsf{in} \\ &x \leftarrow g\ ;\ g\end{aligned}$$

Using Iris, we can verify Landin's knot. We leave the full verification of knot to the reader (see Exercise 87) and focus on a simplified variant in the following, a program of similar structure that uses higher-order state to diverge:

$$\begin{aligned}\text{diverge} := &\mathsf{let}\ d = \mathsf{new}\ (\lambda x.\, x)\ \mathsf{in} \\ &d \leftarrow \lambda x.\, (!\, d)\ x; \\ &(!\, d)()\end{aligned}$$

Lemma 88.

$$\{\mathsf{True}\}\ \text{diverge}\ \{\_.\ \mathsf{False}\}$$

Proof.

Context:	Goal:
	wp diverge $\{\_.\ \mathsf{False}\}$
We allocate d as some location ℓ.	
$\ell \mapsto \lambda x.\, x$	$\mathsf{wp}\ \ell \leftarrow (\lambda x.\, (!\, \ell)\ x)\ ;\ (!\, \ell)()\ \{\_.\ \mathsf{False}\}$
We execute the store. Let $g := \lambda x.\, (!\, \ell)\ x$.	
$\ell \mapsto g$	$\mathsf{wp}\ (!\, \ell)()\ \{\_.\ \mathsf{False}\}$
By Löb.	
$\triangleright(\ell \mapsto g \twoheadrightarrow \mathsf{wp}\ (!\, \ell)()\ \{\_.\ \mathsf{False}\}) * \ell \mapsto g$	$\mathsf{wp}\ (!\, \ell)()\ \{\_.\ \mathsf{False}\}$
After dereferencing ℓ.	
$(\ell \mapsto g \twoheadrightarrow \mathsf{wp}\ (!\, \ell)()\ \{\_.\ \mathsf{False}\}) * \ell \mapsto g$	$\mathsf{wp}\ g\ ()\ \{\_.\ \mathsf{False}\}$
Executing g for one step.	
$(\ell \mapsto g \twoheadrightarrow \mathsf{wp}\ (!\, \ell)()\ \{\_.\ \mathsf{False}\}) * \ell \mapsto g$	$\mathsf{wp}\ (!\, \ell)()\ \{\_.\ \mathsf{False}\}$

□

Exercise 87 We write $f : P \to Q$ for $\forall v.\, \{P(v)\}\, f\, v\, \{w.\, Q(w)\}$. Verify Landin's knot in Iris by proving:

$$\frac{\forall f.\, \{f : P \to Q\}\ t\ f\ \{g.\, g : P \to Q\}}{\{\mathsf{True}\}\ \text{knot}\ t\ \{g.\, g : P \to Q\}}$$

Note that the premise can be rewritten as $t : (\lambda f.\, f : P \to Q) \to (\lambda g.\, g : P \to Q)$ •

Impredicative Invariants Let us now turn to another strength of step-indexing: *impredicative invariants.* To motivate impredicative invariants, we consider an example, caching lazy integers.

Example 89 (Lazy Integers)**.** *A lazy integer is a delayed computation (i.e., a function $f : \mathbf{1} \to \mathsf{int}$), which computes the actual integer only if it is executed. For example, here is an addition function on lazy integers:*

$$\mathrm{add}(f, g) := \lambda\_.\, f() + g()$$

The idea of lazy integers is that the computation that produces the integer may be time intensive or redundant (i.e., the result is not needed) and, hence, we do not evaluate it eagerly. Instead, we shield the computation by functions. Fittingly, we define:

$$\mathrm{LazyInt}(f, n) := \mathsf{wp}\ f()\ \{w.\ w = \overline{n}\}$$

In this definition, we use a weakest precondition and not a Hoare triple, because our lazy integers should not be duplicable. That is, the idea of more efficiency by delaying the execution goes out the window if we recompute the same result over and over by executing f multiple times. Of course, there may be situations where we want to use the same lazy integer multiple times.

For these situations, we introduce an additional function to cache lazy integers. Since we want the resulting lazy integer to be duplicable, the specification of the cache *function should be:*

$$\{\mathrm{LazyInt}(f, n)\}\ \mathrm{cache}(f)\ \{h.\ \Box\, \mathrm{LazyInt}(h, n)\}$$

To define cache*, we use a reference to a data type with three elements: (1) initially, we are in the state* $\mathsf{unused}(f)$ *indicating that we have not executed the function f yet, (2) while we are executing f, we are in the state* $\mathsf{pending}$*, and, (3) finally, after the execution, we store the cached result as* $\mathsf{result}(y)$*. (The datatype constructors* $\mathsf{unused}(f)$, $\mathsf{pending}$*, and* $\mathsf{result}(y)$ *can be encoded using the primitive injections* inj_i*.)*

```
         cache(f) := let c = new(unused(f)) in cachebody(f, c)
cachebody(f, c) := λ_. let r = retrieve(c) in
                          match r with
                          | inj₁(f) ⇒ let y = f() in c ← result(y) ; y
                          | inj₂(y) ⇒ y
                          end
      retrieve(c) := match !c with
                       | unused(f) ⇒ c ← pending ; inj₁(f)
                       | result(y) ⇒ inj₂(y)
                       | pending ⇒ diverge()
                       end
```

During the execution of f, the state of the cache reference c is $\mathsf{pending}$*. If* $\mathrm{cachebody}(f, c)$ *is called again during the* $\mathsf{pending}$ *state, the execution will diverge. It is illegal for the lazy integer to be called while caching is in progress.*

Let us try to verify cache (verifying add is straightfoward). After allocating the reference c, we end up in the following proof state:

$$c \mapsto \mathsf{unused}(f) * \mathrm{LazyInt}(f, n) \vdash \mathsf{wp}\ \mathrm{cachebody}(f, c)\ \{h.\ \Box\, \mathrm{LazyInt}(h, n)\}$$

Now we are stuck! We need to show that cachebody(f, c) is a persistent lazy integer, but we have non-persistent resources: $c \mapsto \mathsf{unused}(f)$ and LazyInt(f, n). We know that we can put $c \mapsto \mathsf{unused}(f)$ into an invariant, but what about LazyInt(f, n)? It does not correspond to a state proposition F. Here, *impredicative invariants* come to the rescue!

Impredicative invariants allow us to put *arbitrary* propositions R into invariants:

$$\textsc{InvPers}\quad \boxed{R}^{\mathcal{N}} \vdash \Box\, \boxed{R}^{\mathcal{N}} \qquad\qquad \textsc{InvAlloc}\quad \frac{P * \boxed{R}^{\mathcal{N}} \vdash \mathsf{wp}^{\mathcal{E}}\, e\, \{v.\, Q(v)\}}{P * R \vdash \mathsf{wp}^{\mathcal{E}}\, e\, \{v.\, Q(v)\}}$$

$$\textsc{InvOpen}\quad \frac{P * \triangleright R \vdash \mathsf{wp}^{\mathcal{E}\setminus\mathcal{N}}\, e\, \{v.\, \triangleright R * Q(v)\} \qquad \mathcal{N} \subseteq \mathcal{E}}{P * \boxed{R}^{\mathcal{N}} \vdash \mathsf{wp}^{\mathcal{E}}\, e\, \{v.\, Q(v)\}}$$

The price we have to pay for this additional expressive power is that the proposition R is guarded by a later modality—we do not get access to it directly.

The reason the later modality shows up is step-indexing: the intuitive model of invariants is cyclic and step-indexing can be used to stratify it. We will discuss this point in more detail later on. For now, it suffices to know that the model of Iris's propositions roughly looks as follows:

$$\mathsf{iProp} = \mathsf{Inv} \to \mathsf{Heap} \to \mathsf{Prop} \qquad\qquad \mathsf{Inv} = \mathbb{N} \xrightarrow{\mathsf{fin}} \mathsf{iProp}$$

Iris propositions are predicates over invariants and program heaps. Invariants, in turn, are finite maps of Iris propositions. Clearly, the definition is recursive: to define Iris propositions, we need Iris propositions in the definition of invariants. Unfortunately, this definition is not only recursive, but also has a negative occurrence. In essence, we define iProp as predicates over iProp, which has no solution in set or type theory. Thus, to make sense of this definition, behind the scenes, step-indexing is used to define iProp similar to how we used step-indexing before to stratify cyclic definitions.

In fact, the state invariants we saw before are just a special case of general impredicative invariants: state propositions belong to the class of so-called *timeless* Iris propositions. If a proposition P is timeless, we can just eliminate a later modality in front of it when proving a weakest precondition. Thus, for timeless propositions, the old invariant opening rule can be derived from the new one! Specifically, timeless propositions satisfy the following rules:

$$\textsc{TimelessPure}\quad \mathsf{timeless}(\phi) \qquad \textsc{TimelessPers}\quad \frac{\mathsf{timeless}(P)}{\mathsf{timeless}(\Box\, P)} \qquad \textsc{TimelessSep}\quad \frac{\mathsf{timeless}(P) \qquad \mathsf{timeless}(Q)}{\mathsf{timeless}(P * Q)} \qquad \textsc{TimelessWand}\quad \frac{\mathsf{timeless}(Q)}{\mathsf{timeless}(P \mathrel{-\!\!*} Q)}$$

$$\textsc{TimelessOr}\quad \frac{\mathsf{timeless}(P) \qquad \mathsf{timeless}(Q)}{\mathsf{timeless}(P \vee Q)} \qquad \textsc{TimelessAnd}\quad \frac{\mathsf{timeless}(P) \qquad \mathsf{timeless}(Q)}{\mathsf{timeless}(P \wedge Q)} \qquad \textsc{TimelessAll}\quad \frac{\forall x.\, \mathsf{timeless}(P(x))}{\mathsf{timeless}(\forall x.\, P)}$$

$$\textsc{TimelessExists}\quad \frac{\forall x.\, \mathsf{timeless}(P(x))}{\mathsf{timeless}(\exists x.\, P)} \qquad \textsc{TimelessPointsto}\quad \mathsf{timeless}(\ell \mapsto v) \qquad \textsc{TimelessStrip}\quad \frac{\mathsf{timeless}(P) \qquad P * Q \vdash \mathsf{wp}^{\mathcal{E}}\, e\, \{v.\, R(v)\}}{(\triangleright P) * Q \vdash \mathsf{wp}^{\mathcal{E}}\, e\, \{v.\, R(v)\}}$$

Note that $\triangleright P$ will in general not be timeless if P is timeless. Thus, TimelessStrip can not be used to strip laters when there are multiple laters above a timeless proposition.

Exercise 88 Derive the invariant opening rule without a later from the new one with a later, assuming that the invariant's contents are timeless. ●

Lemma 90.

$$\{\mathrm{LazyInt}(f, n)\}\ \mathrm{cache}(f)\ \{h.\ \square\, \mathrm{LazyInt}(h, n)\}$$

Proof. We factor the proof into two steps: (1) proving a specification for retrieve and (2) proving the specification for cache. During the proof, we will set up an invariant $\boxed{I_c}^{\mathcal{N}}$ and retrieve will work on the opened invariant. For (contents of) the invariant I_c, we encode the three different states that c can be in:

$$I_c := (c \mapsto \mathsf{unused}(f) * \mathrm{LazyInt}(f, n)) \lor c \mapsto \mathsf{pending} \lor c \mapsto \mathsf{result}(\overline{n})$$

Let us start with retrieve. For retrieve, we show for an arbitrary location c that if we own I_c before the execution (and the invariant $\mathcal{N}$ is currently open), then I_c holds again afterwards and we have either obtained f or the result n:

$$\triangleright I_c \vdash \mathsf{wp}^{\top \setminus \mathcal{N}}\ \mathrm{retrieve}(c)\ \{w.\ I_c * (w = \mathsf{inj}_1(f) * \mathrm{LazyInt}(f, n) \lor w = \mathsf{inj}_2(\overline{n}))\}$$

Context:	Goal:
$\triangleright I_c$	$\mathsf{wp}^{\top \setminus \mathcal{N}}\ \mathrm{retrieve}(c)\ \{w.\ I_c * (w = \mathsf{inj}_1(f) * \mathrm{LazyInt}(f, n) \lor w = \mathsf{inj}_2(\overline{n}))\}$
$c \mapsto \mathsf{unused}(f) * \mathrm{LazyInt}(f, n)$	$\mathsf{match}\ \ !c\ \mathsf{with}$
$\lor c \mapsto \mathsf{pending}$	$\mid \mathsf{unused}(f) \Rightarrow c \leftarrow \mathsf{pending}\ ;\ \mathsf{inj}_1(f)$
$\lor c \mapsto \mathsf{result}(\overline{n})$	$\mid \mathsf{result}(y) \Rightarrow \mathsf{inj}_2(y)$
	$\mid \mathsf{pending} \Rightarrow \mathit{diverge}()$
	end

Case $\mathsf{result}(\overline{n})$.

$c \mapsto \mathsf{result}(\overline{n})$	$\mathsf{inj}_2(\overline{n})$
$c \mapsto \mathsf{result}(\overline{n})$	$I_c * (\mathsf{inj}_2(\overline{n}) = \mathsf{inj}_1(f) * \mathrm{LazyInt}(f, n) \lor \mathsf{inj}_2(\overline{n}) = \mathsf{inj}_2(\overline{n}))$

Done by picking the $\mathsf{result}(n)$ case in I_c.

Case $\mathsf{unused}(f)$.

$c \mapsto \mathsf{unused}(f) * \mathrm{LazyInt}(f, n)$	$c \leftarrow \mathsf{pending}\ ;\ \mathsf{inj}_1(f)$
$c \mapsto \mathsf{pending} * \mathrm{LazyInt}(f, n)$	$I_c * (\mathsf{inj}_1(f) = \mathsf{inj}_1(f) * \mathrm{LazyInt}(f, n) \lor \mathsf{inj}_1(f) = \mathsf{inj}_2(\overline{n}))$

Done by picking the $\mathsf{pending}$ case.

Case $\mathsf{pending}$.

$c \mapsto \mathsf{pending}$	$\mathit{diverge}()$

Done by Löb induction.

Given the specification for retrieve, let us now turn to the more interesting part: the proof of the specification of cache itself.

Context:	Goal:
$\mathrm{LazyInt}(f, n)$	$\mathsf{wp}\ \mathrm{cache}(f)\ \{h.\ \Box\,\mathrm{LazyInt}(h, n)\}$
$\mathrm{LazyInt}(f, n) * c \mapsto \mathsf{unused}(f)$	$\mathsf{wp}\ \mathrm{cachebody}(f, c)\ \{h.\ \Box\,\mathrm{LazyInt}(h, n)\}$
We allocate the invariant I_c.	
$\boxed{I_c}^{\mathcal{N}}$	$\mathsf{wp}\ \mathrm{cachebody}(f, c)\ \{h.\ \Box\,\mathrm{LazyInt}(h, n)\}$

Let $t := \lambda\_.\ \mathsf{let}\ r = \mathrm{retrieve}(c)\ \mathsf{in}\ M_r$
and $M_r := \mathsf{match}\ r\ \mathsf{with}$

```
    | $\mathsf{inj}_1(f) \Rightarrow \mathsf{let}\ y = f()\ \mathsf{in}\ c \leftarrow \mathsf{result}(y)\ ;\ y$
    | $\mathsf{inj}_2(y) \Rightarrow y$
    end.
```

Context:	Goal:
$\boxed{I_c}^{\mathcal{N}}$	$\mathsf{wp}\ t\ \{v.\ \Box\,\mathrm{LazyInt}(v, n)\}$
$\boxed{I_c}^{\mathcal{N}}$	$\Box\,\mathrm{LazyInt}(t, n)$
$\boxed{I_c}^{\mathcal{N}}$	$\mathrm{LazyInt}(t, n)$
$\boxed{I_c}^{\mathcal{N}}$	$\mathsf{wp}\ \mathsf{let}\ r = \mathrm{retrieve}(c)\ \mathsf{in}\ M_r\ \{v.\ v = \overline{n}\}$
$\boxed{I_c}^{\mathcal{N}}$	$\mathsf{wp}\ \mathrm{retrieve}(c)\ \{w.\ \mathsf{wp}\ \mathsf{let}\ r = w\ \mathsf{in}\ M_r\ \{v.\ v = \overline{n}\}\}$
$\boxed{I_c}^{\mathcal{N}} * \triangleright I_c$	$\mathsf{wp}^{\top\setminus\mathcal{N}}\ \mathrm{retrieve}(c)\ \{w.\ \triangleright I_c * \mathsf{wp}\ \mathsf{let}\ r = w\ \mathsf{in}\ M_r\ \{v.\ v = \overline{n}\}\}$
By applying our specification for retrieve.	
$\boxed{I_c}^{\mathcal{N}} * I_c * (w = \mathsf{inj}_1(f) * \mathrm{LazyInt}(f, n) \vee w = \mathsf{inj}_2(\overline{n}))$	$\triangleright I_c * \mathsf{wp}\ \mathsf{let}\ r = w\ \mathsf{in}\ M_r\ \{v.\ v = \overline{n}\}$
By canceling I_c.	
$\boxed{I_c}^{\mathcal{N}} * (w = \mathsf{inj}_1(f) * \mathrm{LazyInt}(f, n) \vee w = \mathsf{inj}_2(\overline{n}))$	$\mathsf{wp}\ \mathsf{let}\ r = w\ \mathsf{in}\ M_r\ \{v.\ v = \overline{n}\}$

Case $w = \mathsf{inj}_2(\overline{n})$.

$\boxed{I_c}^{\mathcal{N}}$	$\mathsf{wp}\ \overline{n}\ \{v.\ v = \overline{n}\}$

Case $w = \mathsf{inj}_1(f)$.

$\boxed{I_c}^{\mathcal{N}} * \mathrm{LazyInt}(f, n)$	$\mathsf{let}\ y = f()\ \mathsf{in}\ c \leftarrow \mathsf{result}(y)\ ;\ y$
By binding $f()$ and using $\mathrm{LazyInt}(f, n)$.	
$\boxed{I_c}^{\mathcal{N}}$	$\mathsf{let}\ y = \overline{n}\ \mathsf{in}\ c \leftarrow \mathsf{result}(y)\ ;\ y$
$\boxed{I_c}^{\mathcal{N}}$	$c \leftarrow \mathsf{result}(\overline{n})\ ;\ \overline{n}$
After updating the invariant.	
	$\mathsf{wp}\ \overline{n}\ \{v.\ v = \overline{n}\}$

□

In the proof, we use a common pattern for dealing with laters we get when opening an invariant: in order to extract timeless components (*e.g.*, $\ell \mapsto v$) that we need directly after opening the invariant (*i.e.*, before taking another step to strip the later), we first apply the later commuting rules (*e.g.*, LaterSep, LaterOr, and LaterExists) to commute the later down to be directly in front of the timeless part we need access to. We then use timelessness to remove the later. The IPM applies the commuting rules automatically when destructing a hypothesis, while a later can be stripped by prefixing an intro pattern with `>`. Thus, to open the invariant in the proof above, we would use the intro pattern `"[[(>Hl & Hlazy) | >Hl] | >Hl]"`, where `Hl` is the points-to fact in the respective cases.

Exercise 89 We define the following function:

$$\text{lazyint\_two} := \lambda f_1\ f_2\ i.\, \mathsf{let}\, c = \text{cache}(i)\, \mathsf{in}\, f_1(c) + f_2(c)$$

Prove the following specification:

$$\frac{(\forall h, n.\, \{\text{LazyInt}(h, n)\}\, f_1(h)\, \{v.\, \exists m.\, v = \overline{m}\}) \qquad (\forall h, n.\, \{\text{LazyInt}(h, n)\}\, f_2(h)\, \{v.\, \exists m.\, v = \overline{m}\})}{\{\text{LazyInt}(i, n)\}\, \text{lazyint\_two}(f_1, f_2, i)\, \{v.\, \exists m.\, v = \overline{m}\}}$$

●

7 Logical Relations

Now that we have seen how to verify small examples in Iris, let us turn to our first larger case study: a logical relation for System F with recursive types and higher-order state. To define the logical relation, we will make some inessential simplifications. That is, we do not change our language and keep using the language from Section 6. This language is missing several constructs (*i.e.*, $\mathsf{pack}\, e$, $\mathsf{unpack}\, e\, \mathsf{as}\, x\, \mathsf{in}\, e_2$, $e\, \langle\rangle$, $\Lambda.\, e$, $\mathsf{roll}\, e$, and $\mathsf{unroll}\, e$). Since we only care about the operational behavior of these constructs, we replace them by operationally equivalent terms instead of primitives:

$$\begin{aligned} \Lambda.\, e &:= \lambda\_.\, e & e\,\langle\rangle &:= e() \\ \mathsf{pack}\, e &:= e & \mathsf{unpack}\, e\, \mathsf{as}\, x\, \mathsf{in}\, e_2 &:= (\lambda x.\, e_2)\, e \\ \mathsf{roll}\, e &:= e & \mathsf{unroll}\, e &:= \mathsf{let}\, x := e\, \mathsf{in}\, x \end{aligned}$$

Exercise 90 Convince yourself that

$$\begin{aligned} (\Lambda.\, e)\,\langle\rangle &\to_{\mathsf{pure}} e \\ \mathsf{unpack}\,(\mathsf{pack}\, v)\, \mathsf{as}\, x\, \mathsf{in}\, e &\to_{\mathsf{pure}} e[x/v] \\ \mathsf{unroll}(\mathsf{roll}\, v) &\to_{\mathsf{pure}} v \end{aligned}$$

●

Semantic Types $\boxed{\text{SemType}}$

A semantic type is a *persistent* predicate $\tau : \mathit{CVal} \to \mathit{iProp}$, meaning $\forall v.\, \tau v \vdash \square\, \tau v$. We make semantic types persistent, because in a function $\lambda x.\, e$ the variable $x : A$ can be used multiple times and, hence, we need to duplicate the argument in the appropriate places.

Note that we do not require τ to be downwards closed with respect to the step-index, which would mean $\forall v.\, \tau v \vdash \triangleright\, \tau v$ in logical step-indexing. The reason is quite simple: all Iris propositions are already downwards-closed by the rule LaterIntro.

Type Interpretations $\boxed{\mathcal{V}[\![A]\!]\delta,\ \mathcal{E}[\![A]\!]\delta,\ \text{and}\ \mathcal{G}[\![\Gamma]\!]\delta}$

In this version of the logical relation, the type interpretations are again predicates over values, expressions, and substitutions. However, this time they are *Iris predicates*, so predicates that return propositions of type *iProp*.

$$\begin{aligned} \mathcal{V}[\![\alpha]\!]\delta &:= \delta(\alpha) \\ \mathcal{V}[\![\mathbf{1}]\!]\delta &:= \{()\} \\ \mathcal{V}[\![\mathsf{int}]\!]\delta &:= \{\overline{n} \mid n \in \mathbb{Z}\} \\ \mathcal{V}[\![\mathsf{bool}]\!]\delta &:= \left\{\overline{b} \,\middle|\, b \in \mathbb{B}\right\} \\ \mathcal{V}[\![A \to B]\!]\delta &:= \{v \mid \square(\forall w.\, w \in \mathcal{V}[\![A]\!]\delta \twoheadrightarrow v\, w \in \mathcal{E}[\![B]\!]\delta)\} \\ \mathcal{V}[\![\forall \alpha.\, A]\!]\delta &:= \{v \mid \square(\forall \tau.\, v\,\langle\rangle \in \mathcal{E}[\![A]\!]\delta, \alpha \mapsto \tau)\} \\ \mathcal{V}[\![\exists \alpha.\, A]\!]\delta &:= \{\mathsf{pack}\, v \mid \exists \tau.\, v \in \mathcal{V}[\![A]\!]\delta, \alpha \mapsto \tau\} \\ \mathcal{V}[\![\mu\alpha.A]\!]\delta &:= \mu\tau.\, \{\mathsf{roll}\, v \mid \triangleright v \in \mathcal{V}[\![A]\!]\delta, \alpha \mapsto \tau\} \\ \mathcal{V}[\![\mathsf{ref}\, A]\!]\delta &:= \left\{\ell \,\middle|\, \boxed{\exists w.\, \ell \mapsto w * w \in \mathcal{V}[\![A]\!]\delta}^{\mathcal{N}}\right\} \\ \mathcal{E}[\![A]\!]\delta &:= \{e \mid \mathsf{wp}\, e\, \{v.\, v \in \mathcal{V}[\![A]\!]\delta\}\} \\ \mathcal{G}[\![\Gamma]\!]\delta &:= \left\{\gamma \,\middle|\, \mathop{\Huge *}_{x:A\in\Gamma} \gamma x \in \mathcal{V}[\![A]\!]\delta\right\} \end{aligned}$$

Let us discuss this definition case by case. For type variables, we once again carry a map that maps them to semantic types. The definitions of the interpretation of base types is straightforward. For function types, we want to say that if applied to values of type A the function evaluates to values of type B. We use the always modality to ensure that functions can be applied more than once.

For polymorphism, we use Iris's built-in quantification—we quantify over semantic types and extend the map with these semantic types. The fact that Iris's propositions can quantify over themselves (or in this case semantic types defined using Iris's propositions) is sometimes referred to as "impredicativity" or "being a higher-order" logic.

For recursive types, we make use of the step-indexed model underlying Iris. We can take recursive fixpoints as long as we make sure that the recursive occurrence is guarded by a later modality (which it is in this case). We will see in the compatibility lemmas how the later modality in this definition is handled.

For references, we make use of the impredicative invariants of Iris. Using them, we can define a protocol which values can be stored in the reference—values of type A. We have to use an invariant (and cannot just use a points-to assertion) to obtain an interpretation of the reference type that is duplicable. By using an impredicative invariant, we can easily handle higher-order reference types like $\mathsf{ref}(\mathbf{1} \to \mathbf{1})$.

For expressions, we want to say that the evaluation of the expression is *safe* and that the resulting value will be of type A. This notion in concisely encapsulated in Iris's weakest precondition, so we use it to lift the value relation to expressions.

Finally, for typing contexts, we just lift the value relations to finite maps. To do so, we use a big separating conjunction over all the type assignments in the context.

Note that, for references, we use a single, fixed invariant namespace $\mathcal{N}$. One can also use separate invariant namespace $\mathcal{N}.\ell$ (*i.e.*, one per location). For our purposes, it does not matter really matter which version we use (since we usually do not need to open two invariants at the same time).

Exercise 91 The case of universal quantification in the logical relation contains a $\Box$-modality, but the case of existential quantification does not. Can you explain why it is not needed? ●

Semantic Typing $\boxed{\Delta\,;\Gamma \vDash e : A}$

$$\Delta\,;\Gamma \vDash e : A := \forall \delta, \gamma.\, \gamma \in \mathcal{G}[\![\Gamma]\!]\delta \vdash \gamma e \in \mathcal{E}[\![A]\!]\delta$$

The semantic typing is then defined as an entailment between the typing assumptions of the variable substitution and the substituted expression. As before, we prove semantic soundness:

Theorem 91 (Semantic Soundness). *If* $\Delta\,;\Gamma \vdash e : A$, *then* $\Delta\,;\Gamma \vDash e : A$.

Proof. By induction on $\Delta\,;\Gamma \vdash e : A$ using compatibility lemmas. $\Box$

We discuss a few select compatibility lemmas.

Lemma 92 (Compatibility Application).

$$\frac{\Delta\,;\Gamma \vDash e_1 : A \to B \qquad \Delta\,;\Gamma \vDash e_2 : A}{\Delta\,;\Gamma \vDash e_1\ e_2 : B}$$

Proof.

Context:	Goal:
$\Delta ; \Gamma \vDash e_1 : (A \to B) * \Delta ; \Gamma \vDash e_2 : A$	$\Delta ; \Gamma \vDash e_1\ e_2 : B$
$\Delta ; \Gamma \vDash e_1 : (A \to B) * \Delta ; \Gamma \vDash e_2 : A * \gamma \in \mathcal{G}\llbracket\Gamma\rrbracket\delta$	$\gamma e_1\ \gamma e_2 \in \mathcal{E}\llbracket B\rrbracket\delta$
$\gamma e_1 \in \mathcal{E}\llbracket A \to B\rrbracket\delta * \gamma e_2 \in \mathcal{E}\llbracket A\rrbracket\delta * \gamma \in \mathcal{G}\llbracket\Gamma\rrbracket\delta$	$\gamma e_1\ \gamma e_2 \in \mathcal{E}\llbracket B\rrbracket\delta$
$\textsf{wp}\ \gamma e_1\ \{v.\, v \in \mathcal{V}\llbracket A \to B\rrbracket\delta\} * \textsf{wp}\ \gamma e_2\ \{v.\, v \in \mathcal{V}\llbracket A\rrbracket\delta\}$	$\textsf{wp}\ \gamma e_1\ \gamma e_2\ \{v.\, v \in \mathcal{V}\llbracket B\rrbracket\delta\}$
Bind γe_2	
$\textsf{wp}\ \gamma e_1\ \{v.\, v \in \mathcal{V}\llbracket A \to B\rrbracket\delta\} * \textsf{wp}\ \gamma e_2\ \{v.\, v \in \mathcal{V}\llbracket A\rrbracket\delta\}$	$\textsf{wp}\ \gamma e_2\ \{v_2.\, \textsf{wp}\ \gamma e_1\ v_2\ \{v.\, v \in \mathcal{V}\llbracket B\rrbracket\delta\}\}$
Wand	
$\textsf{wp}\ \gamma e_1\ \{v.\, v \in \mathcal{V}\llbracket A \to B\rrbracket\delta\} * v_2 \in \mathcal{V}\llbracket A\rrbracket\delta$	$\textsf{wp}\ \gamma e_1\ v_2\ \{v.\, v \in \mathcal{V}\llbracket B\rrbracket\delta\}$
Bind γe_1	
$\textsf{wp}\ \gamma e_1\ \{v.\, v \in \mathcal{V}\llbracket A \to B\rrbracket\delta\} * v_2 \in \mathcal{V}\llbracket A\rrbracket\delta$	$\textsf{wp}\ \gamma e_1\ \{v_1.\, \textsf{wp}\ v_1\ v_2\ \{v.\, v \in \mathcal{V}\llbracket B\rrbracket\delta\}\}$
Wand	
$v_1 \in \mathcal{V}\llbracket A \to B\rrbracket\delta * v_2 \in \mathcal{V}\llbracket A\rrbracket\delta$	$\textsf{wp}\ v_1\ v_2\ \{v.\, v \in \mathcal{V}\llbracket B\rrbracket\delta\}$
$(\forall w.\, w \in \mathcal{V}\llbracket A\rrbracket\delta \mathrel{-\!\!*} v_1\ w \in \mathcal{E}\llbracket B\rrbracket\delta) * v_2 \in \mathcal{V}\llbracket A\rrbracket\delta$	$\textsf{wp}\ v_1\ v_2\ \{v.\, v \in \mathcal{V}\llbracket B\rrbracket\delta\}$
Instantiate the wand with v_2	
$v_1\ v_2 \in \mathcal{E}\llbracket B\rrbracket\delta$	$\textsf{wp}\ v_1\ v_2\ \{v.\, v \in \mathcal{V}\llbracket B\rrbracket\delta\}$
We are done by definition of $\mathcal{E}\llbracket B\rrbracket\delta$.	

□

In the proof above, we have twice used a pattern that frequently comes up in all of the compatibility lemmas: From the expression relation of a subexpression e, we get a weakest precondition as an assumption. We use the bind rule to bind that subexpression and then use the wand rule to match up the postcondition. That way, we get to zap down that subexpression to a value v in the goal and get to assume that it is in the value relation. This closely matches the Bind lemma we saw before for explicitly defined logical relations, and we will shortcut this pattern in the following proofs.

Lemma 93 (Compatibility Type Application)**.**

$$\frac{\Delta ; \Gamma \vDash e : \forall \alpha.\, A}{\Delta ; \Gamma \vDash e\ \langle\rangle : A[B/\alpha]}$$

Proof.

Context:	Goal:
$\Delta ; \Gamma \vDash e : (\forall \alpha.\, A)$	$\Delta ; \Gamma \vDash e\ \langle\rangle : A[B/\alpha]$
$\Delta ; \Gamma \vDash e : (\forall \alpha.\, A) * \gamma \in \mathcal{G}\llbracket\Gamma\rrbracket\delta$	$(\gamma e)\ \langle\rangle \in \mathcal{E}\llbracket A[B/\alpha]\rrbracket\delta$
$\gamma e \in \mathcal{E}\llbracket\forall \alpha.\, A\rrbracket\delta$	$(\gamma e)\ \langle\rangle \in \mathcal{E}\llbracket A[B/\alpha]\rrbracket\delta$
Bind & wand	
$v \in \mathcal{V}\llbracket\forall \alpha.\, A\rrbracket\delta$	$v\ \langle\rangle \in \mathcal{E}\llbracket A[B/\alpha]\rrbracket\delta$
$\forall \tau.\, v\ \langle\rangle \in \mathcal{E}\llbracket A\rrbracket\delta, \alpha \mapsto \tau$	$v\ \langle\rangle \in \mathcal{E}\llbracket A[B/\alpha]\rrbracket\delta$
Boring lemma	
$\forall \tau.\, v\ \langle\rangle \in \mathcal{E}\llbracket A\rrbracket\delta, \alpha \mapsto \tau$	$v\ \langle\rangle \in \mathcal{E}\llbracket A\rrbracket\delta, \alpha \mapsto \mathcal{V}\llbracket B\rrbracket\delta$

□

Lemma 94 (Compatibility Unroll)**.**

$$\frac{\Delta ; \Gamma \vDash e : \mu\alpha.\, A}{\Delta ; \Gamma \vDash \mathsf{unroll}\; e : A[\mu\alpha.\, A/\alpha]}$$

Proof.

Context:	Goal:
$\Delta ; \Gamma \vDash e : (\mu\alpha.\, A)$	$\Delta ; \Gamma \vDash \mathsf{unroll}\; e : A[\mu\alpha.\, A/\alpha]$
$\Delta ; \Gamma \vDash e : (\mu\alpha.\, A) * \gamma \in \mathcal{G}[\![\Gamma]\!]\delta$	$\mathsf{unroll}\;(\gamma e) \in \mathcal{E}[\![A[\mu\alpha.\, A/\alpha]]\!]\delta$
$\gamma e \in \mathcal{E}[\![\mu\alpha.\, A]\!]\delta$	$\mathsf{wp}\;\mathsf{unroll}\;(\gamma e)\;\{u.\, u \in \mathcal{V}[\![A[\mu\alpha.\, A/\alpha]]\!]\delta\}$
Bind & wand	
$v \in \mathcal{V}[\![\mu\alpha.\, A]\!]\delta$	$\mathsf{wp}\;\mathsf{unroll}\; v\;\{u.\, u \in \mathcal{V}[\![A[\mu\alpha.\, A/\alpha]]\!]\delta\}$
Unfold fixpoint	
$\triangleright w \in \mathcal{V}[\![A]\!]\delta, \alpha \mapsto \mathcal{V}[\![\mu\alpha.\, A]\!]$	$\mathsf{wp}\;\mathsf{unroll}\;\mathsf{roll}\; w\;\{u.\, u \in \mathcal{V}[\![A[\mu\alpha.\, A/\alpha]]\!]\delta\}$
Take a step	
$w \in \mathcal{V}[\![A]\!]\delta, \alpha \mapsto \mathcal{V}[\![\mu\alpha.\, A]\!]$	$\mathsf{wp}\; w\;\{u.\, u \in \mathcal{V}[\![A[\mu\alpha.\, A/\alpha]]\!]\delta\}$
$w \in \mathcal{V}[\![A]\!]\delta, \alpha \mapsto \mathcal{V}[\![\mu\alpha.\, A]\!]$	$w \in \mathcal{V}[\![A[\mu\alpha.\, A/\alpha]]\!]\delta$
Boring lemma	
$w \in \mathcal{V}[\![A]\!]\delta, \alpha \mapsto \mathcal{V}[\![\mu\alpha.\, A]\!]$	$w \in \mathcal{V}[\![A]\!]\delta, \alpha \mapsto \mathcal{V}[\![\mu\alpha.\, A]\!]$

□

Lemma 95 (Compatibility Store)**.**

$$\frac{\Delta ; \Gamma \vDash e_1 : \mathsf{ref}\; A \qquad \Delta ; \Gamma \vDash e_2 : A}{\Delta ; \Gamma \vDash e_1 \leftarrow e_2 : \mathbf{1}}$$

Proof.

Context:	Goal:
$\Delta ; \Gamma \vDash e_1 : \mathsf{ref}\; A * \Delta ; \Gamma \vDash e_2 : A$	$\Delta ; \Gamma \vDash e_1 \leftarrow e_2 : \mathbf{1}$
$\Delta ; \Gamma \vDash e_1 : \mathsf{ref}\; A * \Delta ; \Gamma \vDash e_2 : A * \gamma \in \mathcal{G}[\![\Gamma]\!]\delta$	$\gamma e_1 \leftarrow \gamma e_2 \in \mathcal{E}[\![\mathbf{1}]\!]\delta$
$\gamma e_1 \in \mathcal{E}[\![\mathsf{ref}\; A]\!]\delta * \gamma e_2 \in \mathcal{E}[\![A]\!]\delta$	$\gamma e_1 \leftarrow \gamma e_2 \in \mathcal{E}[\![\mathbf{1}]\!]\delta$
Bind & wand (x2)	
$v_1 \in \mathcal{V}[\![\mathsf{ref}\; A]\!]\delta * v_2 \in \mathcal{V}[\![A]\!]\delta$	$\mathsf{wp}\; v_1 \leftarrow v_2\;\{v.\, v \in \mathcal{V}[\![\mathbf{1}]\!]\delta\}$
$\boxed{\exists w.\, \ell \mapsto w * w \in \mathcal{V}[\![A]\!]\delta}^{\mathcal{N}} * v_2 \in \mathcal{V}[\![A]\!]\delta$	$\mathsf{wp}\; \ell \leftarrow v_2\;\{v.\, v \in \mathcal{V}[\![\mathbf{1}]\!]\delta\}$
Set $I := \exists w.\, \ell \mapsto w * w \in \mathcal{V}[\![A]\!]\delta$	
$(\triangleright I) * v_2 \in \mathcal{V}[\![A]\!]\delta$	$\mathsf{wp}^{\top\setminus\mathcal{N}}\; \ell \leftarrow v_2\;\{v.\, \triangleright I * v \in \mathcal{V}[\![\mathbf{1}]\!]\delta\}$
$\ell \mapsto w * (\triangleright w \in \mathcal{V}[\![A]\!]\delta) * v_2 \in \mathcal{V}[\![A]\!]\delta$	$\mathsf{wp}^{\top\setminus\mathcal{N}}\; \ell \leftarrow v_2\;\{v.\, \triangleright I * v \in \mathcal{V}[\![\mathbf{1}]\!]\delta\}$
$\ell \mapsto v_2 * v_2 \in \mathcal{V}[\![A]\!]\delta$	$\mathsf{wp}^{\top\setminus\mathcal{N}}\; ()\;\{v.\, \triangleright I * v \in \mathcal{V}[\![\mathbf{1}]\!]\delta\}$
$\ell \mapsto v_2 * v_2 \in \mathcal{V}[\![A]\!]\delta$	$\triangleright I * () \in \mathcal{V}[\![\mathbf{1}]\!]\delta$
$\ell \mapsto v_2 * v_2 \in \mathcal{V}[\![A]\!]\delta$	$\exists w.\, \ell \mapsto w * w \in \mathcal{V}[\![A]\!]\delta$

□

Exercise 92 Extend the type interpretations with sum and product types and prove the compatibility lemmas. •

Similarly to our "old" logical relation, we can show the safety of MyMutBit (from Section 4.5):

$$\begin{aligned}\text{MUTBIT} &:= \{\text{flip} : \mathbf{1} \to \mathbf{1},\ \text{get} : \mathbf{1} \to \mathsf{bool}\}\\ \text{MyMutBit} &:= \mathsf{let}\, x = \mathsf{new}\, \overline{0}\\ &\qquad \mathsf{in}\,\{\text{flip} := \lambda y.\, \mathsf{assert}\,(!\,x == 0 \vee !\,x == 1)\,;\, x \leftarrow \overline{1} - !\,x,\\ &\qquad\quad \text{get} := \lambda y.\, \mathsf{assert}\,(!\,x == 0 \vee !\,x == 1)\,;\, !\,x > 0\}\end{aligned}$$

Lemma 96.

$$\Delta\,;\Gamma \vDash \text{MyMutBit} : \text{MUTBIT}$$

Proof.

Context:	Goal:
	$\Delta\,;\Gamma \vDash \text{MyMutBit} : \text{MUTBIT}$
$\gamma \in \mathcal{G}[\![\emptyset]\!]\delta$	$\gamma\text{MyMutBit} \in \mathcal{E}[\![\text{MUTBIT}]\!]\delta$
Set $g_x := \{\text{flip} := \lambda y.\, \mathsf{assert}\,(!\,x == 0 \vee !\,x == 1)\,;\, x \leftarrow \overline{1} - !\,x,$ $\text{get} := \lambda y.\, \mathsf{assert}\,(!\,x == 0 \vee !\,x == 1)\,;\, !\,x > 0\}$	$(\mathsf{let}\, x = \mathsf{new}\, \overline{0}\, \mathsf{in}\, g_x) \in \mathcal{E}[\![\text{MUTBIT}]\!]\delta$
	$\mathsf{wp}\ \mathsf{let}\, x = \mathsf{new}\, \overline{0}\, \mathsf{in}\, g_x\, \{v.\, v \in \mathcal{V}[\![\text{MUTBIT}]\!]\delta\}$
$\ell \mapsto \overline{0}$	$\mathsf{wp}\ g_\ell\, \{v.\, v \in \mathcal{V}[\![\text{MUTBIT}]\!]\delta\}$
Allocate an invariant $\boxed{\ell \mapsto \overline{0} \vee \ell \mapsto \overline{1}}^{\mathcal{N}}$	$g_\ell \in \mathcal{V}[\![\text{MUTBIT}]\!]\delta$
Case flip. $\boxed{\ell \mapsto \overline{0} \vee \ell \mapsto \overline{1}}^{\mathcal{N}}$	$(\lambda y.\, \mathsf{assert}\,(!\,\ell == 0 \vee !\,\ell == 1)\,;\, \ell \leftarrow \overline{1} - !\,\ell) \in \mathcal{V}[\![\mathbf{1} \to \mathbf{1}]\!]\delta$
$\boxed{\ell \mapsto \overline{0} \vee \ell \mapsto \overline{1}}^{\mathcal{N}}$	$\mathsf{assert}\,(!\,\ell == 0 \vee !\,\ell == 1)\,;\, \ell \leftarrow \overline{1} - !\,\ell \in \mathcal{E}[\![\mathbf{1}]\!]\delta$
$\boxed{\ell \mapsto \overline{0} \vee \ell \mapsto \overline{1}}^{\mathcal{N}}$	$\mathsf{wp}\ \mathsf{assert}\,(!\,\ell == 0 \vee !\,\ell == 1)\,;\, \ell \leftarrow \overline{1} - !\,\ell\, \{v.\, v \in \mathcal{V}[\![\mathbf{1}]\!]\delta\}$
$\ell \mapsto \overline{0} \vee \ell \mapsto \overline{1}$	$\mathsf{wp}^{\top\setminus\mathcal{N}}\ \mathsf{assert}\,(!\,\ell == 0 \vee !\,\ell == 1)\,;\, \ell \leftarrow \overline{1} - !\,\ell\, \left\{v.\, (\ell \mapsto \overline{0} \vee \ell \mapsto \overline{1}) * v \in \mathcal{V}[\![\mathbf{1}]\!]\delta\right\}$
Case flip, **case** $\ell \mapsto \overline{0}$ $\ell \mapsto \overline{0}$	$\mathsf{wp}^{\top\setminus\mathcal{N}}\ \mathsf{assert}\,(!\,\ell == 0 \vee !\,\ell == 1)\,;\, \ell \leftarrow \overline{1} - !\,\ell\, \left\{v.\, (\ell \mapsto \overline{0} \vee \ell \mapsto \overline{1}) * v \in \mathcal{V}[\![\mathbf{1}]\!]\delta\right\}$
Assert succeeds $\ell \mapsto \overline{0}$	$\mathsf{wp}^{\top\setminus\mathcal{N}}\ \ell \leftarrow \overline{1} - !\,\ell\, \left\{v.\, (\ell \mapsto \overline{0} \vee \ell \mapsto \overline{1}) * v \in \mathcal{V}[\![\mathbf{1}]\!]\delta\right\}$
$\ell \mapsto \overline{1}$	$\mathsf{wp}^{\top\setminus\mathcal{N}}\ ()\, \left\{v.\, (\ell \mapsto \overline{0} \vee \ell \mapsto \overline{1}) * v \in \mathcal{V}[\![\mathbf{1}]\!]\delta\right\}$
$\ell \mapsto \overline{1}$	$(\ell \mapsto \overline{0} \vee \ell \mapsto \overline{1}) * () \in \mathcal{V}[\![\mathbf{1}]\!]\delta$
Case flip, **case** $\ell \mapsto \overline{1}$ Similar.	
Case get. Similar.	

□

8 Ghost State

Recall that in Section 4.7, we briefly discussed what is required to enrich the logical relation presented there with protocols. More concretely, we discussed how we can add *ghost state* (*i.e.*, state which is not present in the program, but useful for its verification) to the logical relation in the form of state transition systems. In the following, we will see how we can work with ghost state (not only in the form of transition systems) in Iris.

To keep matters concrete, we consider an example, the Symbol ADT from Section 4.7:

$$\begin{aligned} \mathrm{SYMBOL} &:= \exists\alpha.\,\{\ \mathsf{mkSym} : \mathbf{1} \to \alpha,\\ &\qquad\quad \mathsf{check} : \alpha \to \mathbf{1}\ \} \end{aligned}$$

$$\begin{aligned} \mathrm{Symbol} &:= \mathsf{let}\ c = \mathsf{new}\ \overline{0}\ \mathsf{in}\\ &\qquad \mathsf{pack}\left\langle \mathsf{int}, \begin{array}{l}\{\ \mathsf{mkSym} := \lambda\_.\ \mathsf{let}\ x = !\,c\ \mathsf{in}\ c \leftarrow x + 1\,;\, x,\\ \quad\ \mathsf{check} := \lambda x.\ \mathsf{assert}\ (x < !\,c)\ \}\end{array} \right\rangle \mathsf{as}\ \mathrm{SYMBOL} \end{aligned}$$

Intuitively, to show that the Symbol ADT is in our logical relation, we would want to set up an invariant I which contains the ownership of the reference c and then pick for α the type of all natural numbers which are currently less than the value of the reference c. In Iris, we can do so by using a piece of ghost state called "monotonically growing natural numbers".

Monotonically growing natural numbers $\boxed{\mathsf{mono}_\gamma(n) \text{ and } \mathsf{lb}_\gamma(n)}$

Monotonically growing natural numbers come in the form of two propositions, $\mathsf{mono}_\gamma(n)$ and $\mathsf{lb}_\gamma(n)$, which represent two pieces of ghost state connected by the name γ. The ghost state $\mathsf{mono}_\gamma(n)$ expresses that the current value of γ is n and that it can only grow over time. The ghost state $\mathsf{lb}_\gamma(n)$ is a lower bound on the value of γ. It remains a lower bound on the value of γ over time since $\mathsf{mono}_\gamma(n)$ can only grow. This relationship is, formally, captured by the following rules:

MakeBound
$\mathsf{mono}_\gamma(n) \vdash \mathsf{mono}_\gamma(n) * \mathsf{lb}_\gamma(n)$

UseBound
$\mathsf{mono}_\gamma(n) * \mathsf{lb}_\gamma(m) \vdash n \geq m$

BoundPers
$\mathsf{lb}_\gamma(n) \vdash \Box\, \mathsf{lb}_\gamma(n)$

IncreaseVal
$\mathsf{mono}_\gamma(n) \vdash \Rrightarrow \mathsf{mono}_\gamma(n+1)$

NewMono
$\mathsf{True} \vdash \Rrightarrow \exists\gamma.\ \mathsf{mono}_\gamma(n)$

MonoTimeless
$\mathsf{timeless}(\mathsf{mono}_\gamma(n))$

BoundTimeless
$\mathsf{timeless}(\mathsf{lb}_\gamma(n))$

The rule MakeBound allows us to create a new lower bound $\mathsf{lb}_\gamma(n)$ from the current value $\mathsf{mono}_\gamma(n)$. The rule UseBound then later on allows us to show that the current value $\mathsf{mono}_\gamma(n)$ is not greater than any of the bounds we have created $\mathsf{lb}_\gamma(m)$. The rule BoundPers ensures that the lower bounds $\mathsf{lb}_\gamma(n)$ are persistent. The rules IncreaseVal and NewMono use a new modality of Iris that we have not discussed so far, the update modality $\Rrightarrow P$, which we will discuss below. Intuitively, IncreaseVal says that we can always increase the current value $\mathsf{mono}_\gamma(n)$ by one with an update. The rule NewMono allows us to create a new monotonically growing natural number $\mathsf{mono}_\gamma(n)$ where the rule picks (a fresh) name γ for us. The rules MonoTimeless and BoundTimeless ensure that both new connectives are timeless and, hence, easy to use in invariants.

The update modality $\boxed{\Rrightarrow P}$

Intuitively, the update modality $\Rrightarrow P$ means that P holds after (possibly) performing some updates to the current ghost state. Besides the ghost state specific rules like IncreaseVal and NewMono, the update modality has the following structural rules:

$$\frac{\textsc{UpdReturn}}{P \vdash \Rrightarrow P} \qquad \frac{\textsc{UpdBind}}{(\Rrightarrow P) * (P \mathrel{-\!\!*} \Rrightarrow Q) \vdash \Rrightarrow Q} \qquad \frac{\textsc{UpdWP}}{\Rrightarrow \mathsf{wp}\ e\ \{v.\, Q(v)\} \vdash \mathsf{wp}\ e\ \{v.\, Q(v)\}}$$

With UpdReturn, we can always update the current state P to itself (by doing nothing). With UpdBind, we can compose two updates into a single update. (Together, the rules turn the update modality $\Rrightarrow P$ into a monad.) With UpdWP, we can execute an update at a weakest precondition. To explain how this rule is used, it is helpful to derive some properties of the update modality first:

Exercise 93 Prove the following derived rules for the update modality:

$$\textsc{UpdWand}\quad (\Rrightarrow P) * (P \mathrel{-\!\!*} Q) \vdash \Rrightarrow Q \qquad \textsc{UpdMono}\quad \frac{P \vdash Q}{\Rrightarrow P \vdash \Rrightarrow Q} \qquad \textsc{UpdTrans}\quad \Rrightarrow \Rrightarrow P \vdash \Rrightarrow P \qquad \textsc{UpdFrame}\quad P * \Rrightarrow Q \vdash \Rrightarrow (P * Q)$$

●

Exercise 94 Derive the following rule for monotonically growing numbers:

$$\textsc{IncreaseMono}\quad \frac{n \leq m}{\mathsf{mono}_\gamma(n) \vdash \Rrightarrow \mathsf{mono}_\gamma(m)}$$

●

With these derived properties in hand, we will now look at an example how one can update ghost state during the proof of a weakest precondition:

Lemma 97.

$$\frac{n \leq m}{(\mathsf{mono}_\gamma(m) \mathrel{-\!\!*} \mathsf{wp}\ e\ \{v.\, Q(v)\}) \vdash (\mathsf{mono}_\gamma(n) \mathrel{-\!\!*} \mathsf{wp}\ e\ \{v.\, Q(v)\})}$$

Proof.

Context:	Goal:
$n \leq m * (\mathsf{mono}_\gamma(m) \mathrel{-\!\!*} \mathsf{wp}\ e\ \{v.\, Q(v)\}) * \mathsf{mono}_\gamma(n)$	$\mathsf{wp}\ e\ \{v.\, Q(v)\}$
By IncreaseMono	
$(\mathsf{mono}_\gamma(m) \mathrel{-\!\!*} \mathsf{wp}\ e\ \{v.\, Q(v)\}) * \Rrightarrow \mathsf{mono}_\gamma(m)$	$\mathsf{wp}\ e\ \{v.\, Q(v)\}$
By UpdWP	
$(\mathsf{mono}_\gamma(m) \mathrel{-\!\!*} \mathsf{wp}\ e\ \{v.\, Q(v)\}) * \Rrightarrow \mathsf{mono}_\gamma(m)$	$\Rrightarrow \mathsf{wp}\ e\ \{v.\, Q(v)\}$
Follows by UpdWand	

□

Exercise 95 Prove the following derived property:

$$(\forall\gamma.\, \mathsf{mono}_\gamma(n) \mathrel{-\!\!*} \mathsf{wp}\, e\, \{v.\, Q(v)\}) \vdash \mathsf{wp}\, e\, \{v.\, Q(v)\})$$

●

Iris Proof Mode The proof of the lemma above shows, in principle, how we update ghost state using the update modality based on its basic rules. Since the pattern used to update ghost state in proofs is very similar to the proof above, the IPM provides some tactic support for it. Specifically, it provides the tactic `iMod "H"` which will remove an update modality from the hypothesis `H` if the current goal is a weakest precondition. In fact, the entire proof above can essentially be condensed into an application of `iMod`: `iMod (increase_mono with "Hn") as "Hm"` updates the ghost state $\mathsf{mono}_\gamma(n)$ (named `Hn` in the context) to $\mathsf{mono}_\gamma(m)$ (named `Hm` in the context).

The Symbol ADT Let us return to the Symbol ADT. Equipped with updates and monotonically growing natural numbers, we can now proceed with the proof along the lines of the intuitive argument mentioned above.

Lemma 98.

Symbol $\in \mathcal{E}[\![\mathrm{SYMBOL}]\!]\delta$

Proof.

Context: Goal:

$$\mathsf{wp}\ \mathrm{Symbol}\ \{v.\, v \in \mathcal{V}[\![\mathrm{SYMBOL}]\!]\}$$

$\ell \mapsto \overline{0}$

$$\mathsf{pack}\{\ \mathsf{mkSym} := \lambda\_.\ \mathsf{let}\ x = !\,\ell\ \mathsf{in}\ \ell \leftarrow x+1\,;x,\ \ \mathsf{check} := \lambda x.\ \mathsf{assert}\ (x < !\,\ell)\ \}$$

$\ell \mapsto \overline{0} * \mathsf{mono}_\gamma(0)$

$$\mathsf{pack}\{\ \mathsf{mkSym} := \lambda\_.\ \mathsf{let}\ x = !\,\ell\ \mathsf{in}\ \ell \leftarrow x+1\,;x,\ \ \mathsf{check} := \lambda x.\ \mathsf{assert}\ (x < !\,\ell)\ \}$$

By NewMono and executing the update.

$\boxed{\exists n.\, \ell \mapsto \overline{n} * \mathsf{mono}_\gamma(n)}^{\mathcal{N}}$

$$\mathsf{pack}\{\ \mathsf{mkSym} := \lambda\_.\ \mathsf{let}\ x = !\,\ell\ \mathsf{in}\ \ell \leftarrow x+1\,;x,\ \ \mathsf{check} := \lambda x.\ \mathsf{assert}\ (x < !\,\ell)\ \}$$

$\boxed{\exists n.\, \ell \mapsto \overline{n} * \mathsf{mono}_\gamma(n)}^{\mathcal{N}}$

$$\exists \tau.\ \ (\lambda\_.\ \mathsf{let}\ x = !\,\ell\ \mathsf{in}\ \ell \leftarrow x+1\,;x) \in \mathcal{V}[\![\mathbf{1} \to \alpha]\!]\delta, \alpha \mapsto \tau$$
$$*(\lambda x.\ \mathsf{assert}\ (x < !\,\ell)) \in \mathcal{V}[\![\alpha \to \mathbf{1}]\!]\delta, \alpha \mapsto \tau$$

Pick $\tau := \{\overline{n} \mid \mathsf{lb}_\gamma(n+1)\}$.

Case mkSym

$\boxed{\exists n.\, \ell \mapsto \overline{n} * \mathsf{mono}_\gamma(n)}^{\mathcal{N}}$

$$(\lambda\_.\ \mathsf{let}\ x = !\,\ell\ \mathsf{in}\ \ell \leftarrow x+1\,;x) \in \mathcal{V}[\![\mathbf{1} \to \alpha]\!]\delta, \alpha \mapsto \tau$$

$\boxed{\exists n.\, \ell \mapsto \overline{n} * \mathsf{mono}_\gamma(n)}^{\mathcal{N}}$

$$\mathsf{wp}\ \mathsf{let}\ x = !\,\ell\ \mathsf{in}\ \ell \leftarrow x+1\,;x\ \{v.\, v \in \tau\}$$

$\exists n.\, \ell \mapsto \overline{n} * \mathsf{mono}_\gamma(n)$

$$\mathsf{wp}^{\top\setminus\mathcal{N}}\ (\mathsf{let}\ x = !\,\ell\ \mathsf{in}\ \ell \leftarrow x+1\,;x)\ \{v.\, v \in \tau * (\exists n.\, \ell \mapsto \overline{n} * \mathsf{mono}_\gamma(n))\}$$

$\ell \mapsto \overline{n+1} * \mathsf{mono}_\gamma(n)$

$$\mathsf{wp}^{\top\setminus\mathcal{N}}\ \overline{n}\ \{v.\, v \in \tau * (\exists n.\, \ell \mapsto \overline{n} * \mathsf{mono}_\gamma(n))\}$$

By Lemma 97 and MakeBound

$\ell \mapsto \overline{n+1} * \mathsf{mono}_\gamma(n+1) * \mathsf{lb}_\gamma(n+1)$

$$\mathsf{wp}^{\top\setminus\mathcal{N}}\ \overline{n}\ \{v.\, v \in \tau * (\exists n.\, \ell \mapsto \overline{n} * \mathsf{mono}_\gamma(n))\}$$

$\ell \mapsto \overline{n+1} * \mathsf{mono}_\gamma(n+1) * \mathsf{lb}_\gamma(n+1)$

$$\overline{n} \in \tau * (\exists n.\, \ell \mapsto \overline{n} * \mathsf{mono}_\gamma(n))$$

Case check

$\boxed{\exists n.\, \ell \mapsto \overline{n} * \mathsf{mono}_\gamma(n)}^{\mathcal{N}}$

$$(\lambda x.\ \mathsf{assert}\ (x < !\,\ell)) \in \mathcal{V}[\![\alpha \to \mathbf{1}]\!]\delta, \alpha \mapsto \tau$$

$\boxed{\exists n.\, \ell \mapsto \overline{n} * \mathsf{mono}_\gamma(n)}^{\mathcal{N}} * v \in \tau$

$$\mathsf{wp}\ (\mathsf{assert}\ (v < !\,\ell))\ \{w.\, w = ()\}$$

$(\exists n.\, \ell \mapsto \overline{n} * \mathsf{mono}_\gamma(n)) * v \in \tau$

$$\mathsf{wp}^{\top\setminus\mathcal{N}}\ (\mathsf{assert}\ (v < !\,\ell))\ \{w.\, w = () * \exists n.\, \ell \mapsto \overline{n} * \mathsf{mono}_\gamma(n)\}$$

$(\exists n.\, \ell \mapsto \overline{n} * \mathsf{mono}_\gamma(n)) * \mathsf{lb}_\gamma(m+1)$

$$\mathsf{wp}^{\top\setminus\mathcal{N}}\ (\mathsf{assert}\ (\overline{m} < !\,\ell))\ \{w.\, w = () * \exists n.\, \ell \mapsto \overline{n} * \mathsf{mono}_\gamma(n)\}$$

By UseBound

$\ell \mapsto \overline{n} * \mathsf{mono}_\gamma(n) * \mathsf{lb}_\gamma(m+1) * m+1 \leq n$

$$\mathsf{wp}^{\top\setminus\mathcal{N}}\ (\mathsf{assert}\ (\overline{m} < !\,\ell))\ \{w.\, w = () * \exists n.\, \ell \mapsto \overline{n} * \mathsf{mono}_\gamma(n)\}$$

Thus the assert succeeds

$\ell \mapsto \overline{n} * \mathsf{mono}_\gamma(n) * \mathsf{lb}_\gamma(m+1) * m+1 \leq n$

$$() = () * \exists n.\, \ell \mapsto \overline{n} * \mathsf{mono}_\gamma(n) \qquad \square$$

Exercise 96 Consider the following loop combinator:

```
loop_comb :=λf.
        let l := new(0̄) in
        let loop := fix loop _. let x := ! l in
                                if f(λ_. ! l)
                                then assert (! l = x);
                                     l ← x + 1̄;
                                     loop ()
                                else ! l
        in loop ()
```

Prove that $\emptyset ; \emptyset \vDash \text{loop\_comb} : ((\mathbf{1} \to \mathsf{int}) \to \mathsf{bool}) \to \mathsf{int}$. Intuitively, the function f receives a closure to get the current iterator value and then returns whether to continue looping.

During the proof, we will need to give the closure passed to f the permission to access the location l. But at the same time, we need to retain the knowledge that the value stored at l will not change, to handle the assertion. For this, a form of *synchronized* ghost state will be useful.

Let X be a fixed type. The ghost theory of *synchronized* ghost state has two elements $\mathsf{left}_\gamma(x : X)$ and $\mathsf{right}_\gamma(y : X)$ which always have the same value:

$$\mathsf{True} \vdash \rightrightarrows \exists \gamma.\, \mathsf{left}_\gamma(x) * \mathsf{right}_\gamma(x) \qquad \mathsf{left}_\gamma(x) * \mathsf{right}_\gamma(y) \vdash x = y$$

$$\mathsf{left}_\gamma(x) * \mathsf{right}_\gamma(y) \vdash \rightrightarrows \mathsf{left}_\gamma(z) * \mathsf{right}_\gamma(z) \qquad \mathsf{timeless}(\mathsf{left}_\gamma(x)) \qquad \mathsf{timeless}(\mathsf{right}_\gamma(x))$$

We can use it to construct a shared reference that everyone can read from, but only one party can mutate. To do so, we set up the invariant:

$$I_{\gamma,\ell} := \exists v.\, \ell \mapsto v * \mathsf{left}_\gamma(v)$$

The exclusive right to mutate the reference is conveyed through the ownership of $\mathsf{right}_\gamma(v)$.
●

Exercise 97 (You only got one shot) Let us consider the following expression e:

```
let x := new(42̄) in
λf. x ← 1337̄;
    f ();
    assert (! x = 1337)
```

Show that $\emptyset ; \emptyset \vDash e : (\mathbf{1} \to \mathbf{1}) \to \mathbf{1}$. During the proof, you will encounter a problem: to verify the assertion, we need to know that x points to the value 1337. But when we initially allocate the invariant containing x (we cannot retain full ownership, as the proof of the function type requires us to give up non-persistent resources), it will point to a different value! The idea is that we have a two-phase system: after the write of 1337 to x before the function call, x will always contain 1337, while the value before that may be different.

For the transition, a new kind of *one shot* ghost state will be helpful, that allows us to take this transition exactly once, and afterwards obtain a persistent certificate.

Let X be a fixed type. The ghost theory of the *one shot* ghost state has two constructors $\mathsf{pending}_\gamma$ and $\mathsf{shot}_\gamma(x : X)$. They obey the following rules:

$$\mathsf{True} \vdash \Rrightarrow \exists \gamma.\, \mathsf{pending}_\gamma \qquad \mathsf{pending}_\gamma \vdash \Rrightarrow \mathsf{shot}_\gamma(x) \qquad \mathsf{shot}_\gamma(x) \vdash \Box\, \mathsf{shot}_\gamma(x)$$

$$\mathsf{pending}_\gamma * \mathsf{shot}_\gamma(x) \vdash \mathsf{False} \qquad \mathsf{pending}_\gamma * \mathsf{pending}_\gamma \vdash \mathsf{False} \qquad \mathsf{shot}_\gamma(x) * \mathsf{shot}_\gamma(y) \vdash x = y$$

$$\mathsf{timeless}(\mathsf{pending}_\gamma) \qquad \mathsf{timeless}(\mathsf{shot}_\gamma(x))$$

●

Exercise 98 We consider the following ADT that generates "red" and "blue" tags. It does so by increasing an internal counter that gets incremented each time a new tag is allocated. This ensures that red and blue tags will always be disjoint, captured by a function that asserts this.

$$\begin{aligned} \text{Twin} := {}& \mathsf{let}\, c = \mathsf{new}\, \overline{0}\, \mathsf{in} \\ & \{\ \text{mkRed} := \lambda\_.\ \mathsf{let}\, x = {!\,c}\, \mathsf{in}\, c \leftarrow x + 1\,;\, x, \\ & \quad \text{mkBlue} := \lambda\_.\ \mathsf{let}\, x = {!\,c}\, \mathsf{in}\, c \leftarrow x + 1\,;\, x, \\ & \quad \text{check} := \lambda x\, y.\ \mathsf{assert}\ (x \neq y)\ \} \end{aligned}$$

Prove that Twin is semantically well-typed at the following type:

$$\begin{aligned} \text{TWIN} := \exists \alpha.\, \exists \beta.\, \{\ & \text{mkRed} : \mathbf{1} \to \alpha, \\ & \text{mkBlue} : \mathbf{1} \to \beta, \\ & \text{check} : \alpha \times \beta \to \mathbf{1}\ \} \end{aligned}$$

During the proof, you will have to pick an invariant (for managing the state of the counter) as well as an interpretation for the two existentially quantified types. You will need some way of linking up the semantic types to this invariant. For that, the theory of *agreement maps* will be useful, featuring connectives $\mathsf{AGM}_\gamma(M)$ and $a \hookrightarrow_\gamma b$. It essentially models a map between two types A and B. (In the case of TWIN, $A = \mathbb{N}$ and the type of two elements $B = \mathsf{red} \mid \mathsf{blue}$ will be useful). The *authoritative* element $\mathsf{AGM}_\gamma(M)$ states that the full map is M, while the persistent *fragments* $a \hookrightarrow_\gamma b$ state that M maps a to b.

It satisfies the following rules:

$$\mathsf{True} \vdash \Rrightarrow \exists \gamma.\, \mathsf{AGM}_\gamma(\emptyset) \qquad k \hookrightarrow_\gamma a * k \hookrightarrow_\gamma b \vdash a = b \qquad k \hookrightarrow_\gamma a \vdash \Box\, k \hookrightarrow_\gamma a$$

$$\frac{M[k] = \bot}{\mathsf{AGM}_\gamma(M) \vdash \Rrightarrow \mathsf{AGM}_\gamma(M[k \mapsto a]) * k \hookrightarrow_\gamma a} \qquad \mathsf{AGM}_\gamma(M) * k \hookrightarrow_\gamma a \vdash M[k] = a$$

$$\mathsf{timeless}(k \hookrightarrow_\gamma a) \qquad \mathsf{timeless}(\mathsf{AGM}_\gamma(M))$$

●

8.1 Resources

Seeing the different ghost state connectives (*i.e.*, $\mathsf{mono}_{\gamma}(n)$, $\mathsf{lb}_{\gamma}(m)$, $\mathsf{shot}_{\gamma}(x)$, $\mathsf{pending}_{\gamma}$, etc.) naturally begs the question what other ghost state Iris has to offer? Instead of a few select ghost theories, Iris has built in an extensible mechanism to introduce and work with ghost state. At the heart of this mechanism is the ghost state connective $\boxed{a}^{\gamma}$ which expresses ownership of the *resource* a (explained below) with name γ. From this ghost state connective other forms of ghost state (*e.g.*, $\mathsf{mono}_{\gamma}(n)$ and $\mathsf{lb}_{\gamma}(m)$) can then be derived by choosing the right kinds of resources.

With monotonically growing natural numbers, we have barely touched the surface of ghost state in Iris. To understand which objects a qualify as resources and which rules the corresponding ghost theory (*i.e.*, rules about the ghost state of the resource) has, it is instructive to think about the interaction of ghost state with the connectives of our logic:

$$\textsc{ResAlloc}\ \dfrac{???}{\vdash \Rrightarrow \exists\gamma.\, \boxed{a}^{\gamma}} \qquad \textsc{ResSep}\ \dfrac{???}{\boxed{a_1}^{\gamma} * \boxed{a_2}^{\gamma} \dashv\vdash \boxed{a_3}^{\gamma}} \qquad \textsc{ResUpd}\ \dfrac{???}{\boxed{a}^{\gamma} \vdash \Rrightarrow \boxed{a'}^{\gamma}} \qquad \textsc{ResTimeless}\ \dfrac{???}{\mathsf{timeless}(\boxed{a}^{\gamma})}$$

$$\textsc{ResPers}\ \ \boxed{a}^{\gamma} \vdash \Box\, \boxed{???}^{\gamma} \qquad \textsc{ResValid}\ \ \boxed{a}^{\gamma} \vdash ???$$

When can we allocate a fresh piece of ghost state γ with resource a (see ResAlloc)? What does it mean to own two resources a_1 and a_2 of the same ghost state γ (see ResSep)? Which fraction of a resource is persistent and can, hence, be duplicated freely (see ResPers)? What does it mean to own a resource a (see ResValid)? When can we update the resource a to a' (see ResUpd)? When is ghost state timeless (see ResTimeless)?

When we introduce new kinds of resources, we have to answer all of the above questions. More concretely, to introduce a new resource kind, we define a so-called *resource algebra* $M = (\mathcal{A}, \cdot, \overline{\mathcal{V}}, |\_|)$. The elements $a, b, c, \ldots$ of a resource algebra $M = (\mathcal{A}, \cdot, \overline{\mathcal{V}}, |\_|)$ are drawn from the carrier type $\mathcal{A}$. The carrier type together with the binary operation $(\cdot)$ forms a partial commutative monoid which governs how separating conjunction behaves on the resources of the algebra. The (meta-level) predicate $\overline{\mathcal{V}}$ encapsulates what it means to own a resource—it must be a "valid" resource, meaning $a \in \overline{\mathcal{V}}$. Finally, the the core $|\_|$ maps resources to their duplicable part (*i.e.*, $|a|$ is obtained from a by striping off all non-duplicable parts). For now, ghost state will always be *timeless*. (We come back to the question of timelessness in Section 9.)

Ghost State Rules $\boxed{\boxed{a}^{\gamma}}$

Concretely, we obtain the following rules for resources from a resource algebra $(\mathcal{A}, \cdot, \overline{\mathcal{V}}, |\_|)$:

$$\textsc{ResAlloc}\ \dfrac{a \in \overline{\mathcal{V}}}{\vdash \Rrightarrow \exists\gamma.\, \boxed{a}^{\gamma}} \qquad \textsc{ResUpd}\ \dfrac{a \rightsquigarrow B}{\boxed{a}^{\gamma} \vdash \Rrightarrow \exists b \in B.\, \boxed{b}^{\gamma}} \qquad \textsc{ResPers}\ \dfrac{|a| \neq \bot}{\boxed{a}^{\gamma} \vdash \Box\, \boxed{|a|}^{\gamma}} \qquad \textsc{ResSep}\ \ \boxed{a}^{\gamma} * \boxed{b}^{\gamma} \dashv\vdash \boxed{a \cdot b}^{\gamma}$$

$$\textsc{ResValid}\ \ \boxed{a}^{\gamma} \vdash a \in \overline{\mathcal{V}} \qquad \textsc{ResTimeless}\ \ \mathsf{timeless}(\boxed{a}^{\gamma})$$

We will discuss how the update $a \rightsquigarrow B$ (used in ResUpd) is defined below once we have discussed the full definition of a resource algebra. A resource algebra cannot be any quadruple $(\mathcal{A}, \cdot, \overline{\mathcal{V}}, |\_|)$. It needs to obey additional rules to ensure soundness of the logic.

Definition 99 (Resource Algebra)**.** *A* resource algebra *(RA) is a quadruple* $(\mathcal{A}, (\cdot) : \mathcal{A} \times \mathcal{A} \to \mathcal{A}, \overline{\mathcal{V}} : \mathcal{A} \to \mathit{Prop}, |{-}| : \mathcal{A} \to \mathcal{A}^?)$ *satisfying:*

$$\forall a, b, c.\, (a \cdot b) \cdot c = a \cdot (b \cdot c) \tag{RA-ASSOC}$$

$$\forall a, b.\, a \cdot b = b \cdot a \tag{RA-COMM}$$

$$\forall a.\, |a| \in \mathcal{A} \Rightarrow |a| \cdot a = a \tag{RA-CORE-ID}$$

$$\forall a.\, |a| \in \mathcal{A} \Rightarrow ||a|| = |a| \tag{RA-CORE-IDEM}$$

$$\forall a, b.\, |a| \in \mathcal{A} \wedge a \preccurlyeq b \Rightarrow |b| \in \mathcal{A} \wedge |a| \preccurlyeq |b| \tag{RA-CORE-MONO}$$

$$\forall a, b.\, a \cdot b \in \overline{\mathcal{V}} \Rightarrow a \in \overline{\mathcal{V}} \tag{RA-VALID-OP}$$

$$\textit{where} \qquad \mathcal{A}^? := \mathcal{A} \uplus \{\bot\} \qquad\qquad x \cdot \bot := \bot \cdot x := x$$

$$a \preccurlyeq b := \exists c \in \mathcal{A}.\, b = a \cdot c \tag{RA-INCL}$$

Unital Resource Algebras Sometimes, it will be useful to work with *unital resource algebras*, resource algebras with a unit $\varepsilon : \mathcal{A}$ such that $\varepsilon \cdot a = a$ and $\varepsilon \in \overline{\mathcal{V}}$ and $|\varepsilon| = \varepsilon$.

Resource Updates The components of a resource algebra precisely characterize how ghost state interacts with the separating conjunction $P * Q$, the box modality $\square P$, and what it means to own a resource. But when can we update a piece of ghost state to another (*i.e.*, when does $a \rightsquigarrow B$ hold)? As it turns out, we do not get to choose an arbitrary update relation $a \rightsquigarrow B$ when we define a resource algebra. Instead, the updates of a resource algebra are already determined by the choice of validity $\overline{\mathcal{V}}$ and the binary operation $(\cdot)$. To understand why, we have to take a closer look at validity.

Validity $a \in \overline{\mathcal{V}}$ characterizes what it means to be a *valid* element of the resource algebra. For example, we will later see that in the resource algebra of monotonically growing natural numbers the resource given by $\mathsf{mono}_\gamma(n) * \mathsf{lb}_\gamma(m)$ is valid iff $n \geq m$ and that the resource given by $\mathsf{mono}_\gamma(n) * \mathsf{mono}_\gamma(n')$ is just invalid regardless of the choice of n and n' (because $\mathsf{mono}_\gamma(n)$ is exclusive like $\ell \mapsto v$, there can only ever be one). Validity is implicitly maintained throughout proofs by Iris: we initially choose a valid resource (with ResAlloc), we maintain validity (implicitly) when we update resources (with ResUpd), and ownership of a resource entails it is valid (with ResValid).

Thus, one might think that we can update ownership of a resource a to an arbitrary other valid resource a'. But this is not the case. To understand why, we consider an example. Suppose we own the ghost state $\mathsf{mono}_\gamma(42)$. What prevents us from updating it to $\mathsf{mono}_\gamma(2)$, so what prevents us from violating the monotonicity baked into the ghost state? The resource behind $\mathsf{mono}_\gamma(2)$ is certainly valid, so we do not violate validity by going from $\mathsf{mono}_\gamma(42)$ to $\mathsf{mono}_\gamma(2)$. But in doing so, we do ignore that there are potential frames of the ghost state $\mathsf{mono}_\gamma(42)$ which we would violate. For instance, other parts of the program could be relying on $\mathsf{lb}_\gamma(41)$ so initially $\mathsf{mono}_\gamma(42) * \mathsf{lb}_\gamma(41)$ would be valid. If we then update $\mathsf{mono}_\gamma(42)$ to $\mathsf{mono}_\gamma(2)$, then the ghost state named γ suddenly becomes invalid (since $2 \not\geq 41$).

To remedy this predicament, we account for arbitrary frames in the definition of *frame preserving updates* $a \rightsquigarrow B$:

Definition 100. *It is possible to do a* frame-preserving update *from $a \in \mathcal{A}$ to $B \subseteq \mathcal{A}$, written $a \rightsquigarrow B$, if $\forall x_\mathsf{f} \in \mathcal{A}^?.\, a \cdot x_\mathsf{f} \in \overline{\mathcal{V}} \Rightarrow \exists b \in B.\, b \cdot x_\mathsf{f} \in \overline{\mathcal{V}}$. We define $a \rightsquigarrow b := a \rightsquigarrow \{b\}$.*

Note that x_f could be $\bot$, so the frame-preserving update can also be applied to elements that have *no* frame. Those elements are called *exclusive resources*.

8.2 Common Resource Algebras

Let us now fill the definition of resource algebras with life. As it turns out, many of the commutative monoids the reader may be familiar with are resource algebras and we will discuss them below. On their own, these resource algebras (and many others that we will see) may seem useless because they do not offer interesting updates $a \rightsquigarrow b$. We will soon see that they are, nevertheless, useful building blocks for obtaining composite resource algebras with useful ghost theories such as the monotonically growing natural numbers.

8.2.1 Numbers

Natural Number Addition $\boxed{(\mathbb{N}, +)}$

The monoid $(\mathbb{N}, +)$ (read "nat plus") forms a unital resource algebra:

$$m \cdot n := m + n \qquad m \in \overline{\mathcal{V}} := \mathsf{True} \qquad |n| := 0 \qquad \varepsilon := 0$$

$$\forall m, n.\, m \rightsquigarrow n \qquad m \preccurlyeq n \iff m \leq n$$

Note that we do not define $m \rightsquigarrow n$ and $m \preccurlyeq n$ here. Their characterization follows from the other definitions.

Natural Number Maximum $\boxed{(\mathbb{N}, \max)}$

The monoid $(\mathbb{N}, \max)$ (read "nat max") forms a unital resource algebra:

$$m \cdot n := \max(m, n) \qquad m \in \overline{\mathcal{V}} := \mathsf{True} \qquad |n| := n \qquad \varepsilon := 0$$

$$\forall m, n.\, m \rightsquigarrow n \qquad m \preccurlyeq n \iff m \leq n$$

Fractions $\boxed{((0, 1], +)}$

Just like for natural numbers, addition on the positive rational numbers $(\mathbb{Q}^+, +)$ forms a resource algebra (without a unit). We obtain a particularly useful resource algebra if we restrict to the interval $(0, 1]$.

$$q_1 \cdot q_2 := q_1 + q_2 \qquad q \in \overline{\mathcal{V}} := q \leq 1 \qquad |q| := \bot$$

$$0 < q_2 \leq q_1 \Rightarrow q_1 \rightsquigarrow q_2 \qquad q_1 \preccurlyeq q_2 \iff q_1 < q_2$$

Fractions do not have a core (*i.e.*, $|q| = \bot$) since addition on positive rational numbers is never idempotent and, similarly, they do not have a unit.

Exercise 99 Show that $(\mathbb{Z}, +)$ and $(\mathbb{N}, \mathsf{min})$ are resource algebras. Derive properties for updates and inclusions. Do they have units? •

8.2.2 Free Resource Algebras

Next, we will discuss some resource algebras that the reader is most likely not familiar with yet. We can use them to equip an arbitrary type X with a resource algebra structure. On their own, they may seem strange at first, but they will turn out to be very useful later on.

Exclusive Resource Algebra $\boxed{\mathit{Ex}(X)}$

The first resource algebra will be the *exclusive resource algebra*. The carrier type of the exclusive resource algebra is $\mathit{Ex}(X) := \mathsf{ex}(x : X) \mid \lightning$. It consists of exclusive elements $\mathsf{ex}(x : X)$ and an invalid element $\lightning$. Its operations are pretty simple:

$$a \cdot b := \lightning \qquad a \in \overline{\mathcal{V}} := a \neq \lightning \qquad |a| := \bot$$

$$\forall x, y.\, \mathsf{ex}(x : X) \rightsquigarrow \mathsf{ex}(y : X) \qquad a \preccurlyeq b \iff b = \lightning$$

The exclusive resource algebra has no unit.

As the name indicates, the exclusive resource algebra ensures that there can always be only one resource $\mathsf{ex}(x : X)$, which gives us the right to update it freely (without violating the assumptions about the current ghost state of other program parts). Thus, the resource algebra is similar to a points-to assertion $\ell \mapsto v$ in that it conveys exclusive ownership.

We say that an element a of an RA A is *exclusive* if $(a \cdot b) \notin \overline{\mathcal{V}}$ for any b (*i.e.*, a has no frame). For instance, all elements of the exclusive algebra $\mathit{Ex}(X)$ are exclusive.

Exercise 100 Show that an exclusive element a can be updated to any other valid element: $a \rightsquigarrow b$ whenever $b \in \overline{\mathcal{V}}$. ●

Agreement Resource Algebra $\boxed{\mathit{Ag}(X)}$

The carrier type of the *agreement algebra* are finite, non-empty sets of elements in X:

$$\mathit{Ag}(X) := \left\{ A \in \mathcal{P}^{\text{fin}}(X) \,\middle|\, X \text{ non-empty} \right\} \qquad \mathsf{ag}(x) := \{x\}$$

The operations on the elements of the resource algebra are given by:

$$A \cdot B := A \cup B \qquad A \in \overline{\mathcal{V}} := \exists x : X.\, A = \{x\} \qquad |A| := A$$

$$\mathsf{ag}(x : X) \rightsquigarrow \mathsf{ag}(y : X) \iff x = y \qquad A \preccurlyeq B \iff A \subseteq B$$

The agreement resource algebra is in some sense the opposite of the exclusive resource algebra: its elements can be freely duplicated, but in exchange we can never update them, as the derived properties below show.

$$\mathsf{ag}(x) \cdot \mathsf{ag}(x) = \mathsf{ag}(x) \qquad \mathsf{ag}(x) \cdot \mathsf{ag}(y) \in \overline{\mathcal{V}} \iff x = y \qquad \mathsf{ag}(x) \preccurlyeq \mathsf{ag}(y) \iff x = y$$

In this sense, resource algebras are similar to invariants $\boxed{P}^{\mathcal{N}}$ in that they can be freely duplicated, but no one can change the statement of the invariant P.

8.2.3 Authoritative Resource Algebra

Let us now turn to one of the most widely used resource algebras of Iris: the *authoritative resource algebra* $\mathit{Auth}(M)$. The idea of this resource algebra is that for a unital resource algebra M, the elements of the resource algebra $\mathit{Auth}(M)$ are either the authoritative element $\bullet a$ or fragments $\circ b$. The relationship between the two kinds of elements is that, at any given point, all fragments $\circ b$ are *included* in the authoritative element $\bullet a$ (*i.e.*, $b \preccurlyeq a$). Moreover, fragments $\circ b$ can only be updated if the corresponding part of the authoritative element $\bullet a$ is also updated.

Authoritative Resource Algebra $\boxed{\mathit{Auth}(M)}$

The carrier type, its elements, and its operations are given by:

$$\mathit{Auth}(M) := \mathit{Ex}(M)^? \times M \qquad \bullet a := (\mathsf{ex}(a), \varepsilon_M) \qquad \circ b := (\bot, b)$$

$$(x, a) \cdot (y, b) := (x \cdot y, a \cdot b) \qquad \overline{\mathcal{V}} := \left\{(\bot, b) \,\middle|\, b \in \overline{\mathcal{V}}_M\right\} \cup \left\{(\mathsf{ex}(a), b) \,\middle|\, b \preccurlyeq_M a \wedge a \in \overline{\mathcal{V}}_M\right\}$$

$$\varepsilon := (\bot, \varepsilon_M) \qquad |(x, a)| := (\bot, |a|)$$

The definition of *Auth*(M) is arguably somewhat mystifying. It does, however, allow us to derive the following useful collection of properties:

fragment rules

$$\circ(a \cdot b) = \circ a \cdot \circ b \qquad |\circ a| = \circ|a| \qquad \circ a \in \overline{\mathcal{V}} \iff a \in \overline{\mathcal{V}}_M$$

$$\circ\varepsilon_M = \varepsilon \qquad \circ a \preccurlyeq \circ b \iff a \preccurlyeq b$$

authoritative element rules

$$\bullet a \in \overline{\mathcal{V}} \iff a \in \overline{\mathcal{V}}_M \qquad \bullet a \cdot \bullet b \in \overline{\mathcal{V}} \iff \mathsf{False}$$

interaction rules

$$\bullet a \cdot \circ b \in \overline{\mathcal{V}} \iff b \preccurlyeq_M a \wedge a \in \overline{\mathcal{V}}_M \qquad (a, b) \rightsquigarrow_{\mathsf{L}} (a', b') \Rightarrow \bullet a \cdot \circ b \rightsquigarrow \bullet a' \cdot \circ b'$$

The fragment rules show that the resource algebra M embeds into the resource algebra *Auth*(M) via the fragments injection $\circ b$ in a sensible way (i.e., preserving all the properties of the original algebra). The rules for the authoritative element $\bullet a$ show that the authoritative element injection embeds the elements of M as exclusive elements (*i.e.*, there can never be two authoritative elements). The interaction rules are the most interesting rules. The first one says that, as explained above, every fragment must be included in the authoritative element. The second one states a condition on when it is possible to update a fragment inside and the authoritative element, the so-called *local update.*

Local Updates $\boxed{(a, b) \rightsquigarrow_{\mathsf{L}} (a', b')}$

It is possible to update a fragment and its corresponding part in the authoritative element whenever we can prove a *local update*:

$$(a, b) \rightsquigarrow_{\mathsf{L}} (a', b') := \forall x \in \mathcal{A}^?.\, a \in \overline{\mathcal{V}}_M \wedge a = b \cdot x \Rightarrow a' \in \overline{\mathcal{V}}_M \wedge a' = b' \cdot x$$

Exercise 101 Prove that if $(a, b) \rightsquigarrow_{\mathsf{L}} (a', b')$, then $\bullet a \cdot \circ b \rightsquigarrow \bullet a' \cdot \circ b'$ from the definitions. Then derive the following properties:

a) If $(a, \varepsilon) \rightsquigarrow_{\mathsf{L}} (a', b)$, then $\bullet a \rightsquigarrow \bullet a' \cdot \circ b$.

b) If $(a, \varepsilon) \rightsquigarrow_{\mathsf{L}} (a', b)$, then $\bullet a \rightsquigarrow \bullet a'$.

c) If $(a, b) \rightsquigarrow_{\mathsf{L}} (a', \varepsilon)$, then $\bullet a \cdot \circ b \rightsquigarrow \bullet a'$. ●

Observe that while ordinary updates $a \rightsquigarrow B$ just refer to validity and are (hence) useless for some of the resource algebras we have seen, the local updates also impose requirements that do not involve validity (*i.e.*, requirements on the composition of the elements). Thus, we obtain some interesting rules for local updates for the resource algebras that we have discussed already:

$$\frac{n + m' = n' + m}{(n, m) \rightsquigarrow_{\mathsf{L}} (n', m')}(\mathbb{N}, +) \qquad\qquad \frac{n \leq k}{(n, m) \rightsquigarrow_{\mathsf{L}} (k, k)}(\mathbb{N}, \max)$$

Monotonically Growing Natural Numbers We now have all the puzzle pieces together to define the ghost theory of monotonically growing natural numbers from our resource algebra combinators. The resource algebra we will use is *Auth*$(\mathbb{N}, \max)$. We define:

$$\mathsf{mono}_\gamma(n) := \boxed{\bullet n}^{\gamma} * \boxed{\circ n}^{\gamma} \qquad\qquad \mathsf{lb}_\gamma(n) := \boxed{\circ n}^{\gamma}$$

With the definition in hand, we can revisit the properties of the ghost theory of monotonically growing natural numbers. We show:

Lemma 101.

UseBound	BoundPers
$\mathsf{mono}_\gamma(n) * \mathsf{lb}_\gamma(m) \vdash n \geq m$	$\mathsf{lb}_\gamma(n) \vdash \Box\, \mathsf{lb}_\gamma(n)$

Proof. UseBound. Observe that $\mathsf{mono}_\gamma(n) * \mathsf{lb}_\gamma(m) \vdash \boxed{\bullet n \cdot \circ n \cdot \circ m}^{\gamma}$. Thus $\bullet n \cdot \circ n \cdot \circ m \in \overline{\mathcal{V}}$ and, hence, $\bullet n \cdot \circ m \in \overline{\mathcal{V}}$. By definition of validity of the authoritative resource algebra, we obtain $m \preccurlyeq n$ and $m \in \overline{\mathcal{V}}$. Thus, since $m \preccurlyeq n \iff m \leq n$ in the resource algebra $(\mathbb{N}, \max)$, we obtain $m \leq n$.

BoundPers. Observe that $|\circ n| = \circ|n| = \circ n$ by the definition of the core in the authoritative resource algebra and the $(\mathbb{N}, \max)$ resource algebra. $\square$

Exercise 102 Prove the remaining lemmas of the ghost theory for monotonically growing natural numbers:

MakeBound	IncreaseVal	NewMono
$\mathsf{mono}_\gamma(n) \vdash \mathsf{mono}_\gamma(n) * \mathsf{lb}_\gamma(n)$	$\mathsf{mono}_\gamma(n) \vdash \Rrightarrow \mathsf{mono}_\gamma(n+1)$	$\mathsf{True} \vdash \Rrightarrow \exists \gamma.\, \mathsf{mono}_\gamma(n)$

MonoTimeless	BoundTimeless
$\mathsf{timeless}(\mathsf{mono}_\gamma(n))$	$\mathsf{timeless}(\mathsf{lb}_\gamma(n))$

●

8.2.4 Common Type Formers

We turn to some common type formers: options, sums, products, and (finite) functions, and show how they can be used to define interesting resource algebras.

Options

$\boxed{\mathsf{option}(M)}$

We define the options resource algebra in such a way that it extends the resource algebra M with a new unit.

$$\begin{aligned}
\mathsf{None} \cdot o &:= o \\
o \cdot \mathsf{None} &:= o \\
\mathsf{Some}(a) \cdot \mathsf{Some}(b) &:= \mathsf{Some}(a \cdot b) \\
\mathsf{None} \in \overline{\mathcal{V}} &:= \mathsf{True} \\
\mathsf{Some}(a) \in \overline{\mathcal{V}} &:= a \in \overline{\mathcal{V}}_M \\
|\mathsf{None}| &:= \mathsf{None} \\
|\mathsf{Some}(a)| &:= \mathsf{Some}(|a|) \\
\varepsilon &:= \mathsf{None}
\end{aligned}$$

We derive some properties about the resource algebra:

$$\mathsf{None} \preccurlyeq o \iff \mathsf{True} \qquad \mathsf{Some}(a) \preccurlyeq o \iff \exists b.\, o = \mathsf{Some}(b) \wedge (a \preccurlyeq b \vee a = b)$$

Exercise 103 What happens when we extend a *unital* resource algebra with an additional unit? Does the resulting resource algebra have *two* units? •

Sums $\boxed{M_1 +_{\lightning} M_2}$

With the *sum* resource algebra, we can express a form of disjunction of two resource algebras M_1 and M_2. Its elements can either be from the first or the second resource algebra (or an invalid element):

$$M_1 +_{\lightning} M_2 := \mathsf{inj}_1(a_1 : M_1) \;\mid\; \mathsf{inj}_2(a_2 : M_2) \;\mid\; \lightning$$

The operations are given by:

$$\begin{aligned}
\mathsf{inj}_i(a_i) \cdot \mathsf{inj}_i(b_i) &:= \mathsf{inj}_i(a_i \cdot_{M_i} b_i) \\
x \cdot y &:= \lightning && \text{otherwise} \\
\overline{\mathcal{V}} &:= \left\{\mathsf{inj}_1(a_1) \,\middle|\, a_1 \in \overline{\mathcal{V}}_{M_1}\right\} \cup \left\{\mathsf{inj}_2(a_2) \,\middle|\, a_2 \in \overline{\mathcal{V}}_{M_2}\right\} \\
|\mathsf{inj}_i(a_i)| &:= \mathsf{inj}_i(|a_i|) \\
\mathsf{inj}_i(a_i) \preccurlyeq \mathsf{inj}_i(b_i) &\iff a_i \preccurlyeq_{M_i} b_i
\end{aligned}$$

We obtain three interesting update rules for this resource algebra:

$$\frac{a_i \rightsquigarrow B_i}{\mathsf{inj}_i(a_i) \rightsquigarrow \{\mathsf{inj}_i(b_i) \mid b_i \in B_i\}}\,\textsc{UpdInj} \qquad \frac{\mathrm{excl}_{M_1}(a_1) \qquad a_2 \in \overline{\mathcal{V}}_{M_2}}{\mathsf{inj}_1(a_1) \rightsquigarrow \mathsf{inj}_2(a_2)}\,\textsc{UpdFlipLR} \qquad \frac{\mathrm{excl}_{M_2}(a_2) \qquad a_1 \in \overline{\mathcal{V}}_{M_1}}{\mathsf{inj}_2(a_2) \rightsquigarrow \mathsf{inj}_1(a_1)}\,\textsc{UpdFlipRL}$$

where $\mathrm{excl}(a) := \forall b.\, (a \cdot b) \notin \overline{\mathcal{V}}$ means a is exclusive. That is, there cannot be any possible frame and, hence, it is sound to flip the sides of the disjunction. The exclusive algebra has exclusive elements (*i.e.*, $\mathrm{excl}(\mathsf{ex}(x : X))$) and various algebras derived from the exclusive algebra. If a resource algebra has a unit, then there are no interesting exclusive elements (*i.e.*, no valid exclusive elements).

Exercise 104 (Products and Functions) The resource algebras for products and functions are straightforward pointwise liftings. Define them and prove that they form a resource algebra. When do they have units? When are their elements exclusive? ●

Finite Functions $\boxed{K \xrightarrow{\mathsf{fin}} M}$

For an countably infinite set K and a resource algebra M, the resource algebra of finite functions $K \xrightarrow{\mathsf{fin}} M$ lifts the resource algebra structure of M to finite maps. This resource algebra is used, for example, to obtain a ghost state version of heaps. The operations on the resource algebra are given by:

$$\begin{aligned} h_1 \cdot h_2 &:= [k \mapsto a \mid k \mapsto a \in h_1, k \notin \mathsf{dom}\, h_2] \\ &\quad \cup [k \mapsto a \mid k \mapsto a \in h_2, k \notin \mathsf{dom}\, h_1] \\ &\quad \cup [k \mapsto a \cdot b \mid k \mapsto a \in h_1, k \mapsto b \in h_2] \\ h \in \overline{\mathcal{V}} &:= \forall k \in \mathsf{dom}\, h.\, h(k) \in \overline{\mathcal{V}}_M \\ |h| &:= [k \mapsto |a| \mid k \mapsto a \in h, |a| \neq \bot] \\ \varepsilon &:= \emptyset \end{aligned}$$

We derive:

$$h_1 \preccurlyeq h_2 \iff \forall k \in \mathsf{dom}\, h_1.\, k \in \mathsf{dom}\, h_2 \wedge (h_1(k) = h_2(k) \vee h_1(k) \preccurlyeq h_2(k))$$

$$\textsc{Alloc}\quad \frac{G \text{ infinite} \qquad a \in \overline{\mathcal{V}}}{\emptyset \rightsquigarrow \{[k \mapsto a] \mid k \in G\}} \qquad\qquad \textsc{Update}\quad \frac{a \rightsquigarrow B}{h[k \mapsto a] \rightsquigarrow \{h[k \mapsto b] \mid b \in B\}}$$

Note that the update rule Alloc is the first rule that truly makes use of the fact that frame preserving updates $a \rightsquigarrow B$ go from an element of the resource algebra a to *a set of elements* of the resource algebra B. We need a set of elements, because there is no single key k such that $\emptyset \rightsquigarrow [k \mapsto a]$, since k could always be used as part of the frame. In other words, since updates need to be frame preserving, picking specific keys is impossible because we cannot ensure that they have not already been picked by some frame. We can, however, pick a set of elements. For every potential frame, since the frame is a *finite* map, there exists some fresh key k in the *infinite* set G that is not contained in the domain of the frame.

Exercise 105 Use Alloc to prove:

$$\textsc{Extend}\quad \frac{G \text{ infinite} \qquad a \in \overline{\mathcal{V}}}{h \rightsquigarrow \{h[k \mapsto a] \mid k \in G\}}$$

●

8.3 Examples

Let us now turn to a number of examples that show case useful ghost theories that we can derive from our resource algebras.

Synchronized Ghost State Recall the ghost theory for synchronized ghost state:

$$\mathsf{True} \vdash \Rrightarrow \exists \gamma.\, \mathsf{left}_\gamma(x) * \mathsf{right}_\gamma(x) \qquad \mathsf{left}_\gamma(x) * \mathsf{right}_\gamma(y) \vdash x = y$$

$$\mathsf{left}_\gamma(x) * \mathsf{right}_\gamma(y) \vdash \Rrightarrow \mathsf{left}_\gamma(z) * \mathsf{right}_\gamma(z)$$

We can derive it from the resource algebra $\mathsf{Sync} := \mathit{Auth}(\mathsf{option}(\mathit{Ex}(X)))$.

Exercise 106 Verify that the resource algebra Sync indeed yields the desired ghost theory with $\mathsf{left}_\gamma(x) := \boxed{\bullet\mathsf{Some}(\mathsf{ex}(x))}^{\gamma}$ and $\mathsf{right}_\gamma(x) := \boxed{\circ\mathsf{Some}(\mathsf{ex}(x))}^{\gamma}$. •

Ghost variables with fractions The synchronized ghost state shown above can be generalized: what if we do not only want to have two parts $\mathsf{left}_\gamma(x)$ and $\mathsf{right}_\gamma(x)$ that need to agree, but more than that? We can use fractions to achieve this. Concretely, consider the following ghost theory:

$$\mathsf{True} \vdash \Rrightarrow \exists \gamma.\, \gamma \overset{1}{\hookrightarrow} x \qquad \gamma \overset{q}{\hookrightarrow} x * \gamma \overset{q'}{\hookrightarrow} y \vdash x = y * q + q' \leq 1$$

$$\gamma \overset{q}{\hookrightarrow} x * \gamma \overset{q'}{\hookrightarrow} x \dashv\vdash \gamma \overset{q+q'}{\hookrightarrow} x \qquad \gamma \overset{1}{\hookrightarrow} x \vdash \Rrightarrow \gamma \overset{1}{\hookrightarrow} y$$

We can derive it with the resource algebra $\mathsf{GVar} := ((0, 1], +) \times \mathit{Ag}(X)$.

We define $\gamma \overset{q}{\hookrightarrow} x := \boxed{(q, \mathsf{ag}(x))}^{\gamma}$. Note that the update $(1, \mathsf{ag}(x)) \rightsquigarrow (1, \mathsf{ag}(y))$ needed to prove the update rule relies on the fact that the fraction 1 rules out any non-trivial frame.

Exercise 107 Verify that the resource algebra GVar indeed yields the desired ghost theory for ghost variables with fractions. •

Oneshot Consider again the oneshot ghost theory:

$$\mathsf{True} \vdash \Rrightarrow \exists \gamma.\, \mathsf{pending}_\gamma \qquad \mathsf{pending}_\gamma \vdash \Rrightarrow \mathsf{shot}_\gamma(x) \qquad \mathsf{shot}_\gamma(x) \vdash \Box\, \mathsf{shot}_\gamma(x)$$

$$\mathsf{pending}_\gamma * \mathsf{shot}_\gamma(x) \vdash \mathsf{False} \qquad \mathsf{pending}_\gamma * \mathsf{pending}_\gamma \vdash \mathsf{False} \qquad \mathsf{shot}_\gamma(x) * \mathsf{shot}_\gamma(y) \vdash x = y$$

$$\mathsf{timeless}(\mathsf{pending}_\gamma) \qquad \mathsf{timeless}(\mathsf{shot}_\gamma(x))$$

We can derive it from the resource algebra $\mathsf{OneShot} := \mathit{Ex}(\mathbf{1}) +_{\lightning} \mathit{Ag}(X)$.

Exercise 108 Verify that the resource algebra $\mathsf{OneShot}$ indeed yields the desired ghost theory with $\mathsf{pending}_\gamma := \boxed{\mathsf{inj}_1(\mathsf{ex}())}^{\gamma}$ and $\mathsf{shot}_\gamma(x) := \boxed{\mathsf{inj}_2(\mathsf{ag}(x))}^{\gamma}$. •

Agreement maps Recall the agreement map theory:

$$\mathsf{True} \vdash \Rrightarrow \exists \gamma.\, \mathsf{AGM}_\gamma(\emptyset) \qquad k \hookrightarrow_\gamma a * k \hookrightarrow_\gamma b \vdash a = b \qquad k \hookrightarrow_\gamma a \vdash \Box\, k \hookrightarrow_\gamma a$$

$$\frac{M[k] = \bot}{\mathsf{AGM}_\gamma(M) \vdash \Rrightarrow \mathsf{AGM}_\gamma(M[k \mapsto a]) * k \hookrightarrow_\gamma a} \qquad \mathsf{AGM}_\gamma(M) * k \hookrightarrow_\gamma a \vdash M[k] = a$$

$$\mathsf{timeless}(k \hookrightarrow_\gamma a) \qquad \mathsf{timeless}(\mathsf{AGM}_\gamma(M))$$

It can be derived with the resource algebra $\mathsf{AgMap} := \mathit{Auth}(A \xrightharpoonup{\mathsf{fin}} \mathit{Ag}(B))$.

Exercise 109 Verify that the resource algebra AgMap indeed yields the desired ghost theory with $\mathsf{AGM}_\gamma(M) := \boxed{\bullet[k \mapsto \mathsf{ag}(v) | (k \mapsto v) \in M]}^{\gamma}$ and $k \hookrightarrow_\gamma a := \boxed{\circ[k \mapsto \mathsf{ag}(a)]}^{\gamma}$. ●

Updatable maps We consider a modification of the agreement map theory which allows updates to the map when having ownership of a fragment. Essentially, with this algebra we can model the points-to connective.

$$\mathsf{True} \vdash \mid\!\Rrightarrow \exists \gamma.\, \mathsf{EGM}_\gamma(\emptyset) \qquad\qquad k \Mapsto_\gamma a * k \Mapsto_\gamma b \vdash \mathsf{False}$$

$$\mathsf{EGM}_\gamma(M) * k \Mapsto_\gamma a \vdash \mid\!\Rrightarrow \mathsf{EGM}_\gamma(M[k \mapsto b]) * k \Mapsto_\gamma b$$

$$\frac{M[k] = \bot}{\mathsf{EGM}_\gamma(M) \vdash \mid\!\Rrightarrow \mathsf{EGM}_\gamma(M[k \mapsto a]) * k \Mapsto_\gamma a} \qquad \mathsf{EGM}_\gamma(M) * k \Mapsto_\gamma a \vdash M[k] = a$$

$$\mathsf{timeless}(k \Mapsto_\gamma a) \qquad\qquad \mathsf{timeless}(\mathsf{EGM}_\gamma(M))$$

It can be derived with the resource algebra $\mathsf{ExMap} := \mathit{Auth}(A \xrightharpoonup{\mathsf{fin}} \mathit{Ex}(B))$.

Exercise 110 Verify that the resource algebra ExMap indeed yields the desired ghost theory with $\mathsf{EGM}_\gamma(M) := \boxed{\bullet[k \mapsto \mathsf{ex}(v) | (k \mapsto v) \in M]}^{\gamma}$ and $k \Mapsto_\gamma a := \boxed{\circ[k \mapsto \mathsf{ex}(v)]}^{\gamma}$. ●

Exercise 111 What ghost theory do we obtain when we modify the AgMap resource algebra to $\mathsf{GhostMap} := \mathit{Auth}(A \xrightharpoonup{\mathsf{fin}} ((0,1],+) \times \mathit{Ag}(B))$? How does this theory relate to the one for ExMap?

Challenge: Can you come up with a modified notion of fractions that allows you to obtain one map algebra to rule them all, *i.e.*, an algebra that allows you to combine the reasoning principles of the ghost theories of ExMap, AgMap, and $\mathsf{GhostMap}$? (Think about the effect that the product with fractions has on the agreement RA). ●

8.4 Advanced Ghost State

In the following, we will discuss one of the more advanced applications of ghost state and invariants. Concretely, wee will use them to built a *binary logical relation* on top of our existing program logic. We will define a binary logical relation $\Delta ; \Gamma \vDash e_i \preceq e_s : A$ which can be used to prove contextual refinement $\Delta ; \Gamma \vDash e_i \leq_{\mathrm{ctx}} e_s : A$.

$$\begin{aligned} \Delta ; \Gamma \vDash e_i \leq_{\mathrm{ctx}} e_s : A := \forall C :(\Delta ; \Gamma ; A) \rightsquigarrow (\emptyset ; \emptyset ; \mathbf{1}), h, h'.& \\ (C[e_i], h) \downarrow ((), h') \Rightarrow \exists h''. (C[e_s], h) \downarrow ((), h'')& \end{aligned}$$

Contextual refinement ensures that the behavior of the expression e_i is included in the behavior of the expression e_s. For example, one can think of e_s as a specification program and of e_i as the implementation. Then contextual refinement ensures that the implementation (in any potential context) does not have more behaviors than those allowed by the specification.

Once again, we make use of (typed) program contexts C here. In this case, we use them to express the capabilities of a potential program in which we could replace the specification expression e_s with the implementation expression e_i. (We can use the type unit $\mathbf{1}$ here, because if the implementation returns a different result, we can usually construct a context Since our language has various constructs for diverging programs, it suffices to return unit

Exercise 112 The function $\lambda x.\, x + \overline{1}$ does not contextually refine the function $\lambda x.\, x - \overline{1}$ at type $\mathbb{N} \to \mathbb{N}$. Prove it by giving a distinguishing context. •

8.4.1 A Binary Logical Relation

To simplify proving contextual refinement, we set up a logical relation—this time in Iris. Most of the cases of the logical relation are straightforward generalizations of the unary logical relation from Section 7:

$$\begin{aligned}
\mathcal{V}[\![\alpha]\!]\delta &:= \delta(\alpha) \\
\mathcal{V}[\![\mathbf{1}]\!]\delta &:= \{((), ())\} \\
\mathcal{V}[\![\mathsf{int}]\!]\delta &:= \{(\overline{n}, \overline{n}) \mid n \in \mathbb{Z}\} \\
\mathcal{V}[\![\mathsf{bool}]\!]\delta &:= \left\{(\overline{b}, \overline{b}) \,\middle|\, b \in \mathbb{B}\right\} \\
\mathcal{V}[\![A \to B]\!]\delta &:= \left\{(v, v') \,\middle|\, \Box(\forall w, w'.\, (w, w') \in \mathcal{V}[\![A]\!]\delta \mathrel{-\!\!*} (v\, w, v'\, w') \in \mathcal{E}[\![B]\!]\delta)\right\} \\
\mathcal{V}[\![\forall \alpha.\, A]\!]\delta &:= \{(v, w) \mid \Box(\forall \tau.\, (v\, \langle\rangle, w\, \langle\rangle) \in \mathcal{E}[\![A]\!]\delta, \alpha \mapsto \tau)\} \\
\mathcal{V}[\![\exists \alpha.\, A]\!]\delta &:= \{(\mathsf{pack}\, v, \mathsf{pack}\, w) \mid \exists \tau.\, (v, w) \in \mathcal{V}[\![A]\!]\delta, \alpha \mapsto \tau\} \\
\mathcal{V}[\![\mu \alpha. A]\!]\delta &:= \mu \tau.\, \{(\mathsf{roll}\, v, \mathsf{roll}\, w) \mid \triangleright(v, w) \in \mathcal{V}[\![A]\!]\delta, \alpha \mapsto \tau\} \\
\mathcal{V}[\![\mathsf{ref}\, A]\!]\delta &:= \left\{(\ell, r) \,\middle|\, \boxed{\exists v, w.\, \ell \mapsto v * r \mapsto_{\mathrm{s}} w * (v, w) \in \mathcal{V}[\![A]\!]\delta}^{\mathcal{N}}\right\} \\
\mathcal{E}[\![A]\!]\delta &:= \left\{(e, e') \,\middle|\, e \preceq e' : \{v, w.\, (v, w) \in \mathcal{V}[\![A]\!]\delta\}\right\} \\
\mathcal{G}[\![\Gamma]\!]\delta &:= \left\{(\gamma, \gamma') \,\middle|\, \mathop{\Huge *}_{x:A \in \Gamma} (\gamma x, \gamma' x) \in \mathcal{V}[\![A]\!]\delta\right\}
\end{aligned}$$

Finally, we define:

$$\Delta ; \Gamma \vDash e_i \preceq e_s : A := \forall \delta, \gamma.\, \boxed{\mathcal{R}}^{\mathcal{N}_{\mathrm{R}}} * (\gamma_i, \gamma_s) \in \mathcal{G}[\![\Gamma]\!]\delta \vdash (\gamma_i e_i, \gamma_s e_s) \in \mathcal{E}[\![A]\!]\delta$$

where $\boxed{\mathcal{R}}^{\mathcal{N}_{\mathrm{R}}}$ is an invariant that we will discuss shortly (in Section 8.4.3).

Binary Simulation $\boxed{e \preceq e' : \{v, w.\, Q(v, w)\}}$

The interesting question is of course now:

"*How is the binary simulation* $e \preceq e' : \{v, w.\, Q(v, w)\}$ *defined?*"

Here, we can truly see the power of Iris's ghost state for the first time. The binary simulation (and by extension the logical relation) is defined on top of the program logic of Iris just by using a clever combination of *invariants* and *ghost state* [**?**, **?**]:

$$e \preceq e' : \{v, w.\, Q(v, w)\} := \forall K.\, \mathsf{spec}(K[e']) \mathrel{-\!\!*} \mathsf{wp}\, e\, \{v.\, \exists w.\, \mathsf{spec}(K[w]) * Q(v, w)\}$$

Let us discuss this definition by first focusing on the new connectives. The connectives $\mathsf{spec}(e)$ (used in the binary simulation) and $\ell \mapsto_{\mathrm{s}} v$ (used in the reference case of the logical relation) are two new forms of ghost state. They are used to turn the specification program into ghost state—into a *"ghost program"* if you will. The connective $\mathsf{spec}(e)$ expresses that the expression in the ghost program is currently e. The connective $\ell \mapsto_{\mathrm{s}} v$ expresses that the heap in the ghost program currently stores the value v in location ℓ. We will discuss the precise details of the ghost theory (and its implementation) shortly. For now, it suffices to understand that if we own the ghost expression $\mathsf{spec}(e)$ and some ghost heap fragments, then we can "step" the specification program by updating our ghost state. For example, if we own $\mathsf{spec}((\lambda x.\, x)\, \overline{42})$, then we can update the source program to $\mathsf{spec}(\overline{42})$. Similarly, if we own $\mathsf{spec}(!\,\ell)$ and $\ell \mapsto_{\mathrm{s}} \overline{42}$, then we can update the source program to $\mathsf{spec}(\overline{42})$ and while retaining our ownership of $\ell \mapsto_{\mathrm{s}} \overline{42}$.

Equipped with connectives to reason about our specification program (*i.e.*, the ghost program), let us now turn to the definition of the binary simulation $e \preceq e' : \{v, w.\, Q(v, w)\}$. As in the case of the unary relation, we need to prove a weakest precondition—this time of our our implementation e. What is new here is that we *get to assume* the specification program is e' (in some evaluation context) K. We then have to make sure that by the time we have finished executing e, we have executed the ghost program to a state where e' has become a value w (still in the evaluation context K). Finally, the resulting values of e and e' need to be related by Q so we can make sure that they are, for example, the same integer.

To familiarize ourselves more with the definition of the binary simulation, we prove two lemmas that we have encountered already for other logical relations (in slightly modified form):

Lemma 102 (Value Inclusion)**.** $Q(v, w) \vdash v \preceq w : \{v, w.\, Q(v, w)\}$

Proof.

Context:	Goal:
$Q(v, w)$	$v \preceq w : \{v, w.\, Q(v, w)\}$
$Q(v, w) * \mathsf{spec}(K[w])$	$\mathsf{wp}\, v\, \{v.\, \exists w.\, \mathsf{spec}(K[w]) * Q(v, w)\}$
Which follows by Value.	

□

Lemma 103 (Bind)**.**

$$e_1 \preceq e_2 : \{v_1, v_2.\, K_1[v_1] \preceq K_2[v_2] : \{v, w.\, Q(v, w)\}\} \vdash K_1[e_1] \preceq K_2[e_2] : \{v, w.\, Q(v, w)\}$$

Proof.

Context:	Goal:
$e_1 \preceq e_2 : \{v_1, v_2.\, K_1[v_1] \preceq K_2[v_2] : \{v, w.\, Q(v,w)\}\}$	$K_1[e_1] \preceq K_2[e_2] : \{v, w.\, Q(v,w)\}$
$e_1 \preceq e_2 : \{v_1, v_2.\, K_1[v_1] \preceq K_2[v_2] : \{v, w.\, Q(v,w)\}\}$ $\mathsf{spec}(K[K_2[e_2]])$	$\mathsf{wp}\ K_1[e_1]\ \{v.\, \exists w.\, \mathsf{spec}(K[w]) * Q(v,w)\}$
Since $K[K_2[e_2]] = K[K_2][e_2]$ we can instantiate the premise. $\mathsf{wp}\ e_2\ \{v_1.\, \exists v_2.\, \mathsf{spec}(K[K_2][v_2]) * K_1[v_1] \preceq K_2[v_2] : \{v, w.\, Q(v,w)\}\}$	$\mathsf{wp}\ K_1[e_1]\ \{v.\, \exists w.\, \mathsf{spec}(K[w]) * Q(v,w)\}$
By WpBind $\exists v_2.\, \mathsf{spec}(K[K_2][v_2]) * K_1[v_1] \preceq K_2[v_2] : \{v, w.\, Q(v,w)\}$	$\mathsf{wp}\ K_1[v_1]\ \{v.\, \exists w.\, \mathsf{spec}(K[w]) * Q(v,w)\}$
Since $K[K_2][v_2] = K[K_2[v_2]]$, we can instantiate the binary simulation. $\mathsf{wp}\ K_1[v_1]\ \{v.\, \exists w.\, \mathsf{spec}(K[w]) * Q(v,w)\}$	$\mathsf{wp}\ K_1[v_1]\ \{v.\, \exists w.\, \mathsf{spec}(K[w]) * Q(v,w)\}$

$\square$

The Ghost Theory Let us now turn to the ghost theory. Compared to the ghost theories that we have discussed previously, there is something special about this ghost theory. The ghost update rules *only hold in the presence of an invariant* $\boxed{\mathcal{R}}^{\mathcal{N}_\mathrm{R}}$. We will discuss which proposition $\mathcal{R}$ we need to choose to make the ghost theory work in Section 8.4.3. For now, the reader may think of the invariant $\boxed{\mathcal{R}}^{\mathcal{N}_\mathrm{R}}$ as the puzzle piece that ties the ghost program connectives $\mathsf{spec}(e)$ and $\ell \mapsto_\mathrm{s} v$ to an actual program execution (of the specification) such that the updates on the ghost state correspond to actual computation steps. We obtain the following ghost theory:

SourcePure
$$\frac{e \to_{\mathsf{pure}} e' \qquad \mathcal{N}_\mathrm{R} \subseteq \mathcal{E}}{\boxed{\mathcal{R}}^{\mathcal{N}_\mathrm{R}} * \mathsf{spec}(e) \vdash \Rrightarrow_{\mathcal{E}} \mathsf{spec}(e')}$$

SourceAlloc
$$\frac{\mathcal{N}_\mathrm{R} \subseteq \mathcal{E}}{\boxed{\mathcal{R}}^{\mathcal{N}_\mathrm{R}} * \mathsf{spec}(K[\mathsf{new}(v)]) \vdash \Rrightarrow_{\mathcal{E}} \exists \ell.\, \mathsf{spec}(K[\ell]) * \ell \mapsto_\mathrm{s} v}$$

SourceLoad
$$\frac{\mathcal{N}_\mathrm{R} \subseteq \mathcal{E}}{\boxed{\mathcal{R}}^{\mathcal{N}_\mathrm{R}} * \mathsf{spec}(K[!\,\ell]) * \ell \mapsto_\mathrm{s} v \vdash \Rrightarrow_{\mathcal{E}} \mathsf{spec}(K[v]) * \ell \mapsto_\mathrm{s} v}$$

SourceStore
$$\frac{\mathcal{N}_\mathrm{R} \subseteq \mathcal{E}}{\boxed{\mathcal{R}}^{\mathcal{N}_\mathrm{R}} * \mathsf{spec}(K[\ell \leftarrow v]) * \ell \mapsto_\mathrm{s} w \vdash \Rrightarrow_{\mathcal{E}} \mathsf{spec}(K[()]) * \ell \mapsto_\mathrm{s} v}$$

Besides the invariant $\boxed{\mathcal{R}}^{\mathcal{N}_\mathrm{R}}$ there is something else that is peculiar about this ghost theory. The update modalities carry a mask $\mathcal{E}$. The reason for this mask is quite simple: to interact with the invariant $\boxed{\mathcal{R}}^{\mathcal{N}_\mathrm{R}}$, we need to be able to *open it*. With the updates that we have discussed so far, $\Rrightarrow P$, opening invariants is impossible—we can only use them to update our ghost state. To additionally interact with invariants when we update our ghost state, Iris offers an additional update modality—the *fancy update*.

8.4.2 Fancy Updates

In general, fancy updates are of the form ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} P$. The updates we have seen (*i.e.*, $\Rrightarrow_{\mathcal{E}} P$) are a derived form $\Rrightarrow_{\mathcal{E}} P := {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}} P$. The idea of the fancy update ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} P$ is that P holds after updating the ghost state and after going from the current mask $\mathcal{E}_1$ to the mask $\mathcal{E}_2$. More concretely, when we prove an update ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} P$, we can *open* all the invariants which are in $\mathcal{E}_1$, perform ghost state updates, and then have to *close* all the invariants that are in $\mathcal{E}_2$. For example, in proving ${}^{\top}{\Rrightarrow}^{\top\setminus\mathcal{N}_{\mathrm{R}}} P$, we get to open $\boxed{\mathcal{R}}^{\mathcal{N}_{\mathrm{R}}}$ and in proving ${}^{\top\setminus\mathcal{N}_{\mathrm{R}}}{\Rrightarrow}^{\top}$, we have to close $\boxed{\mathcal{R}}^{\mathcal{N}_{\mathrm{R}}}$ again. Let us fill this idea with life by discussing the rules of the fancy update modality ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} P$:

$$\textsc{FancyReturn}\quad P \vdash {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}} P \qquad \textsc{FancyBind}\quad ({}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} P) * (P \mathrel{-\!\!*} {}^{\mathcal{E}_2}{\Rrightarrow}^{\mathcal{E}_3} Q) \vdash {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_3} Q \qquad \textsc{FancyUpd}\quad \Rrightarrow P \vdash {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}} P$$

$$\textsc{FancyWp}\quad {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}} \mathsf{wp}^{\mathcal{E}}\, e\, \{v.\, Q(v)\} \vdash \mathsf{wp}^{\mathcal{E}}\, e\, \{v.\, Q(v)\} \qquad \textsc{FancyTimeless}\quad \frac{\mathsf{timeless}(P)}{\triangleright P \vdash {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}} P}$$

$$\textsc{FancyInv}\quad \frac{\mathcal{N} \subseteq \mathcal{E}}{\boxed{P}^{\mathcal{N}} \vdash {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}\setminus\mathcal{N}} \triangleright P * (\triangleright P \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}}{\Rrightarrow}^{\mathcal{E}} \mathsf{True})}$$

$$\textsc{FancyMaskFrame}\quad {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1 \uplus \mathcal{E}}{\Rrightarrow}^{\mathcal{E}_2 \uplus \mathcal{E}} P \qquad \textsc{FancyIntroMask}\quad \frac{\mathcal{E}_2 \subseteq \mathcal{E}_1}{{}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_1} P \vdash {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2}\, {}^{\mathcal{E}_2}{\Rrightarrow}^{\mathcal{E}_1} P}$$

Just like the ordinary update modality $\Rrightarrow P$, the fancy update modality enjoys the return (*i.e.*, FancyReturn) and bind (*i.e.*, FancyBind) rules. For the fancy update, the masks compose transitively in the FancyBind rule: if we can get to P by going from $\mathcal{E}_1$ to $\mathcal{E}_2$ and then we can use P to get to Q at mask $\mathcal{E}_3$, then we can directly go to from $\mathcal{E}_1$ to Q at $\mathcal{E}_3$. The rule FancyUpd allows us to turn an update into a fancy update, allowing us to reuse all of our existing ghost theories. The rule FancyWp allows us to eliminate (*i.e.*, to execute) a fancy update at a weakest precondition. One can think of the fancy update $\Rrightarrow_{\mathcal{E}} P$ as an update collecting a number of modifications to the invariants in the mask $\mathcal{E}$. When we get to a weakest precondition, we can execute them all with FancyWp. Similar to previous ghost theories (on standard updates $\Rrightarrow P$), the rule FancyWp allows us to execute the updates from our ghost theory (*e.g.*, the ghost program theory) while our goal is a weakest precondition. We will see an example of such a use case shortly.

The rule FancyInv is the most interesting one. Together with the other rules, we can use this rule to open, modify, and then close invariants again all as part of proving a fancy update ${}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}} P$. That is, the rule FancyInv allows to open the invariant $\mathcal{N}$ (as long as its namespace is still contained in the mask) to get $\triangleright P$. Moreover, we get a funny looking wand $\triangleright P \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}}{\Rrightarrow}^{\mathcal{E}} \mathsf{True}$. Once we are done manipulating the contents of the invariant, we can close it again using the wand (with the closing update). We will also see an example of such a use case shortly.

After opening an invariant, it is typically very useful to eliminate laters from timeless propositions. The rule FancyTimeless lets us do just that (as we will see below).

Finally, the rule FancyMaskFrame allows us to frame part of the mask (*e.g.*, to open the invariants in $\mathcal{E}$ later), and the rule FancyIntroMask allows us to introduce an intermediate mask where some invariants could have been opened. These last two rules are useful in rare occasions, so we list them here. However, since they are seldomly used, the reader may spare them little thought.

Exercise 113 Prove the following derived rules for the fancy update modality:

FancyWand
$$({}^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_2} P) * (P \twoheadrightarrow Q) \vdash {}^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_2} Q$$

FancyMono
$$\frac{P \vdash Q}{{}^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_2} Q}$$

FancyTrans
$${}^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_2}\, {}^{\mathcal{E}_2}\Rrightarrow^{\mathcal{E}_3} P \vdash {}^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_3} P$$

FancyFrame
$$P * {}^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_2} Q \vdash {}^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_2} P * Q$$

●

Example: Executing Fancy Updates To understand better how we *use* ghost theories with fancy updates, let us discuss a concrete example: advancing the specification program in the binary simulation by one, pure step.

Lemma 104.

$$\frac{e_s \rightarrow_{\mathsf{pure}} e_s'}{\boxed{\mathcal{R}}^{\mathcal{N}_R} * e_i \preceq e_s' : \{v, w.\, Q(v, w)\} \vdash e_i \preceq e_s : \{v, w.\, Q(v, w)\}}$$

Proof. The proof proceeds analogously to Lemma 97.

Context: Goal:

$\boxed{\mathcal{R}}^{\mathcal{N}_\mathrm{R}} * e_i \preceq e_s' : \{v, w.\, Q(v, w)\} * \mathsf{spec}(K[e_s])$ $\mathsf{wp}\ e_i\ \{v.\, \exists w.\, \mathsf{spec}(K[w]) * Q(v, w)\}$

Since $e_s \rightarrow_{\mathsf{pure}} e_s'$, we have $K[e_s] \rightarrow_{\mathsf{pure}} K[e_s']$.
By SourcePure, we obtain:

$\boxed{\mathcal{R}}^{\mathcal{N}_\mathrm{R}} * e_i \preceq e_s' : \{v, w.\, Q(v, w)\} * \Rrightarrow_\top \mathsf{spec}(K[e_s'])$ $\mathsf{wp}\ e_i\ \{v.\, \exists w.\, \mathsf{spec}(K[w]) * Q(v, w)\}$

By FancyWp

$\boxed{\mathcal{R}}^{\mathcal{N}_\mathrm{R}} * e_i \preceq e_s' : \{v, w.\, Q(v, w)\} * \Rrightarrow_\top \mathsf{spec}(K[e_s'])$ $\Rrightarrow_\top \mathsf{wp}\ e_i\ \{v.\, \exists w.\, \mathsf{spec}(K[w]) * Q(v, w)\}$

Follows by FancyWand

□

Example: Proving Fancy Updates Let us now consider the other side of the coin: *proving* ghost theories with fancy updates. To this end, let us discuss the concrete example of proving the rule SourcePure of the ghost program ghost theory. Since we still have not defined $\mathcal{R}$, we do so under suitable assumptions about $\mathcal{R}$:

Lemma 105. *Suppose that (1) if* $e \to_{\mathsf{pure}} e'$*, then* $\mathcal{R} * \mathsf{spec}(e) \vdash \Rrightarrow \mathsf{spec}(e') * \mathcal{R}$*, and (2)* $\mathsf{timeless}(\mathcal{R})$*. We prove:*

$$\frac{\mathcal{N}_R \subseteq \mathcal{E} \qquad e \to_{\mathsf{pure}} e'}{\boxed{\mathcal{R}}^{\mathcal{N}_R} * \mathsf{spec}(e) \vdash \Rrightarrow_{\mathcal{E}} \mathsf{spec}(e')}$$

Proof.

Context:	Goal:
$\boxed{\mathcal{R}}^{\mathcal{N}_\mathrm{R}} * \mathsf{spec}(e)$	$\Rrightarrow_{\mathcal{E}} \mathsf{spec}(e')$
By FancyInv.	
$\mathsf{spec}(e) * {}^{\mathcal{E}}\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}_\mathrm{R}} \triangleright\mathcal{R} * (\triangleright\mathcal{R} \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}_\mathrm{R}}\Rrightarrow^{\mathcal{E}} \mathsf{True})$	$\Rrightarrow_{\mathcal{E}} \mathsf{spec}(e')$
By FancyTrans.	
$\mathsf{spec}(e) * {}^{\mathcal{E}}\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}_\mathrm{R}} \triangleright\mathcal{R} * (\triangleright\mathcal{R} \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}_\mathrm{R}}\Rrightarrow^{\mathcal{E}} \mathsf{True})$	${}^{\mathcal{E}}\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}_\mathrm{R}}\, {}^{\mathcal{E}}\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}_\mathrm{R}} \mathsf{spec}(e')$
By FancyWand.	
$\mathsf{spec}(e) * \triangleright\mathcal{R} * (\triangleright\mathcal{R} \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}_\mathrm{R}}\Rrightarrow^{\mathcal{E}} \mathsf{True})$	${}^{\mathcal{E}}\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}_\mathrm{R}} \mathsf{spec}(e')$
By FancyTimeless using timelessness of $\mathcal{R}$.	
$\mathsf{spec}(e) * \mathcal{R} * (\triangleright\mathcal{R} \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}_\mathrm{R}}\Rrightarrow^{\mathcal{E}} \mathsf{True})$	${}^{\mathcal{E}}\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}_\mathrm{R}} \mathsf{spec}(e')$
By assumption.	
$(\Rrightarrow \mathsf{spec}(e') * \mathcal{R}) * (\triangleright\mathcal{R} \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}_\mathrm{R}}\Rrightarrow^{\mathcal{E}} \mathsf{True})$	${}^{\mathcal{E}}\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}_\mathrm{R}} \mathsf{spec}(e')$
By FancyUpd.	
$(\Rrightarrow \mathsf{spec}(e') * \mathcal{R}) * (\triangleright\mathcal{R} \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}_\mathrm{R}}\Rrightarrow^{\mathcal{E}} \mathsf{True})$	$\Rrightarrow\, {}^{\mathcal{E}}\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}_\mathrm{R}} \mathsf{spec}(e')$
By UpdWand.	
$(\mathsf{spec}(e') * \mathcal{R}) * (\triangleright\mathcal{R} \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}_\mathrm{R}}\Rrightarrow^{\mathcal{E}} \mathsf{True})$	${}^{\mathcal{E}}\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}_\mathrm{R}} \mathsf{spec}(e')$
By LaterIntro.	
$\mathsf{spec}(e') * {}^{\mathcal{E}\setminus\mathcal{N}_\mathrm{R}}\Rrightarrow^{\mathcal{E}} \mathsf{True}$	${}^{\mathcal{E}}\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}_\mathrm{R}} \mathsf{spec}(e')$
By FancyWand.	
$\mathsf{spec}(e') * \mathsf{True}$	$\mathsf{spec}(e')$
Which is trivial.	

□

Note that in the last two step, we make use of the funny looking wand. Even though, its conclusion looks like it is trivial, its conclusion is actually what allows us to *close* the invariant again (*i.e.*, to get rid of the fancy update in the goal whose masks are growing). After opening an invariant, the funny wand is essentially a "callback" that we can use to close the invariant again now that we have done all of our updates.

Note that in the IPM, we have considerable automation simplifying most of the mask handling for us. At an intuitive level, what happens in the above proof is very similar to just working with plain updates. At some places, we have to do some shuffling with the updates, but in those instances in Coq the IPM is there to help.

8.4.3 Ghost State Model

Now that we have seen the ghost theory for the ghost program in action, let us turn to its model: the definition of the invariant $\boxed{\mathcal{R}}^{\mathcal{N}_\mathrm{R}}$ and the ghost state connectives $\mathsf{spec}(e)$ and $\ell \mapsto_{\mathrm{s}} v$. We fix an initial configuration of the specification program (e_0, h_0) and assume the specification executes with the operational semantics $(e, h) \rightsquigarrow (e', h')$. Under these assumptions, we define:

$$\begin{aligned} \mathcal{R} &:= \exists e', h'.\, (e_0, h_0) \rightsquigarrow^* (e', h') * \mathsf{left}_{\gamma_{\text{expr}}}(e') * \mathsf{EGM}_\gamma(h') \\ \mathsf{spec}(e) &:= \mathsf{right}_{\gamma_{\text{expr}}}(e) \\ \ell \mapsto_{\mathrm{s}} v &:= \ell \Mapsto_{\gamma_{\text{heap}}} v \end{aligned}$$

The high-level idea of this definition is that the invariant $\boxed{\mathcal{R}}^{\mathcal{N}_{\mathrm{R}}}$ stores the execution of the initial source program (e_0, h_0) to the current configuration (e', h'). The current expression e' and the current heap h' are then made available outside of the invariant through ghost state—the ghost state γ_{heap} and γ_{expr} to be precise.

Ghost Theories The ghost state connectives $\mathsf{left}_\gamma(e)$ and $\mathsf{right}_\gamma(e)$ belong to the theory of *synchronized ghost state*. Recall that they arise from the resource algebra $\mathit{Auth}(\mathsf{option}(\mathit{Ex}(\mathrm{Expr})))$ with $\mathsf{left}_\gamma(e) := \boxed{\bullet\mathsf{Some}(\mathsf{ex}(e))}^\gamma$ and $\mathsf{right}_\gamma(e) := \boxed{\circ\mathsf{Some}(\mathsf{ex}(e))}^\gamma$. Moreover, they obey the following rules:

$$\mathsf{True} \vdash \Rrightarrow \exists \gamma.\, \mathsf{left}_\gamma(x) * \mathsf{right}_\gamma(x) \qquad \mathsf{left}_\gamma(x) * \mathsf{right}_\gamma(y) \vdash \Rrightarrow \mathsf{left}_\gamma(z) * \mathsf{right}_\gamma(z)$$

$$\mathsf{left}_\gamma(x) * \mathsf{right}_\gamma(y) \vdash x = y \qquad \mathsf{timeless}(\mathsf{left}_\gamma(x)) \qquad \mathsf{timeless}(\mathsf{right}_\gamma(x))$$

The connectives $\mathsf{EGM}_\gamma(h)$ and $\ell \Mapsto_\gamma v$ belong to the ghost theory of *mutable heaps*.

Recall that the theory consists of an authoritative heap $\mathsf{EGM}_\gamma(h)$ and its *mutable* fragments $\ell \Mapsto_\gamma v$. The underlying resource algebra is $\mathit{Auth}(\mathit{Loc} \xrightharpoonup{\mathsf{fin}} \mathit{Ex}(\mathit{Val}))$ where $\mathsf{EGM}_\gamma(h) := \boxed{\bullet[\ell \mapsto \mathsf{ex}(v) \mid \ell \mapsto v \in h]}^\gamma$ and $\ell \Mapsto_\gamma v := \boxed{\circ\ell \mapsto \mathsf{ex}(v)}^\gamma$. The rules of the ghost theory are given by:

$$\mathsf{True} \vdash \Rrightarrow \exists \gamma.\, \mathsf{EGM}_\gamma(h) \qquad \frac{\ell \notin \mathsf{dom}\, h}{\mathsf{EGM}_\gamma(h) \vdash \Rrightarrow \mathsf{EGM}_\gamma(h[\ell \mapsto v]) * \ell \Mapsto_\gamma v}$$

$$\mathsf{EGM}_\gamma(h) * \ell \Mapsto_\gamma v \vdash h(\ell) = v \qquad \mathsf{EGM}_\gamma(h) * \ell \Mapsto_\gamma v \vdash \Rrightarrow \mathsf{EGM}_\gamma(h[\ell \mapsto w]) * \ell \Mapsto_\gamma w$$

$$\mathsf{timeless}(\mathsf{EGM}_\gamma(h)) \qquad \mathsf{timeless}(\ell \Mapsto_\gamma v)$$

Fixed Ghost Names Note that in the definition of our ghost theory for the ghost program (*i.e.*, in $\mathcal{R}$, $\mathsf{spec}(e)$, and $\ell \mapsto_{\mathrm{s}} v$), we use some fixed names γ_{expr} and γ_{heap}. For the ghost state names γ that we have seen so far were always picked dynamically during the proof (*e.g.*, by using ResAlloc). In this instance, we do not need multiple copies of the same ghost theory, so a single pair of ghost names suffices that we fix alongside with the initial configuration (e_0, h_0). In terms of the rules we have discussed so far, one may imagine that the names γ_{expr} and γ_{heap} are picked once in the beginning (using ResAlloc) and then all the proofs proceed parametrically over the ghost names and the initial configuration.

That is, to be precise, the definition of our logical relation needs to quantify over the initial configuration and the ghost names for the heap and expression:

$$\Delta\,;\Gamma \vDash e_i \preceq e_s : A :=$$
$$\forall \gamma_{\text{expr}}, \gamma_{\text{heap}}, e_0, h_0, \delta, \gamma.\ \boxed{\mathcal{R}_{\gamma_{\text{expr}},\gamma_{\text{heap}},e_0,h_0}}^{\mathcal{N}_{\text{R}}} * (\gamma_i, \gamma_s) \in \mathcal{G}[\![\Gamma]\!]\delta \vdash (\gamma_i e_i, \gamma_s e_s) \in \mathcal{E}[\![A]\!]\delta$$

Ghost Program Rules Let us now return to the rules from 8.4.1 for executing the ghost program. We now have all the puzzle pieces in hand to prove them.

Lemma 106.

$$\textsc{SourcePure}\quad \frac{e \rightarrow_{\mathsf{pure}} e' \qquad \mathcal{N}_R \subseteq \mathcal{E}}{\boxed{\mathcal{R}}^{\mathcal{N}_R} * \mathsf{spec}(e) \vdash \Rrightarrow_{\mathcal{E}} \mathsf{spec}(e')}$$

Proof. Since $\mathcal{R}$ is trivially timeless, it suffices to prove:

$$\textit{if } e \rightarrow_{\mathsf{pure}} e', \text{then } \mathcal{R} * \mathsf{spec}(e) \vdash \Rrightarrow \mathsf{spec}(e') * \mathcal{R}$$

by Lemma 105. We proceed with the proof:

Context:	Goal:
$\mathcal{R} * \mathsf{spec}(e)$	$\Rrightarrow \mathsf{spec}(e') * \mathcal{R}$
$(e_0, h_0) \rightsquigarrow^* (e'', h'') * \mathsf{left}_{\gamma_{\text{expr}}}(e'') * \mathsf{EGM}_\gamma(h'') * \mathsf{right}_{\gamma_{\text{expr}}}(e)$	$\Rrightarrow \mathsf{spec}(e') * \mathcal{R}$
Using the rules of the synchronized ghost state theory, we know $e'' = e$.	
$(e_0, h_0) \rightsquigarrow^* (e, h'') * \mathsf{left}_{\gamma_{\text{expr}}}(e) * \mathsf{EGM}_\gamma(h'') * \mathsf{right}_{\gamma_{\text{expr}}}(e)$	$\Rrightarrow \mathsf{spec}(e') * \mathcal{R}$
Since $e \rightarrow_{\mathsf{pure}} e'$, we have $(e, h'') \rightsquigarrow (e', h'')$.	
$(e_0, h_0) \rightsquigarrow^* (e', h'') * \mathsf{left}_{\gamma_{\text{expr}}}(e) * \mathsf{EGM}_\gamma(h'') * \mathsf{right}_{\gamma_{\text{expr}}}(e)$	$\Rrightarrow \mathsf{spec}(e') * \mathcal{R}$
By updating the synchronized ghost state.	
$(e_0, h_0) \rightsquigarrow^* (e', h'') * \mathsf{left}_{\gamma_{\text{expr}}}(e') * \mathsf{EGM}_\gamma(h'') * \mathsf{right}_{\gamma_{\text{expr}}}(e')$	$\mathsf{spec}(e') * \mathcal{R}$
Which follows by definition.	

□

Exercise 114 Prove the remaining ghost program rules: SourceAlloc, SourceLoad, and SourceStore. •

8.4.4 Fundamental Property and Adequacy

With the ghost theory in hand, we can proceed to prove the fundamental property and the closure under program contexts:

Theorem 107 (Fundamental Property)**.** *If $\Delta\,;\Gamma \vdash e : A$, then $\Delta\,;\Gamma \vDash e \preceq e : A$.*

Proof. By induction on $\Delta\,;\Gamma \vdash e : A$ using compatibility lemmas for each case. □

Theorem 108 (Compatibility with Program Contexts)**.** *If $\Delta\,;\Gamma \vDash e_i \preceq e_s : A$ and $C : (\Delta\,;\Gamma\,;A) \rightsquigarrow (\Delta'\,;\Gamma'\,;A')$, then $\Delta'\,;\Gamma' \vDash C[e_i] \preceq C[e_s] : A'$.*

Proof. By induction on $C : (\Delta\,;\Gamma\,;A) \rightsquigarrow (\emptyset\,;\emptyset\,;\mathbf{1})$ using compatibility lemmas for each case. □

As for the other logical relations that we have seen in earlier sections, the proofs above rely on compatibility lemmas for each typing rule. We discuss the lemma for function application explicitly.

Lemma 109.

$$\frac{\Delta\,;\Gamma \vDash e_1 \preceq e_1' : A \to B \qquad \Delta\,;\Gamma \vDash e_2 \preceq e_2' : A}{\Delta\,;\Gamma \vDash e_1\, e_2 \preceq e_1'\, e_2' : B}$$

Proof.

Context: Goal:

$\boxed{\mathcal{R}}^{\mathcal{N}_\mathrm{R}} * (\gamma_i, \gamma_s) \in \mathcal{G}[\![\Gamma]\!]\delta$ $\qquad ((\gamma_i e_1)\,(\gamma_i e_2), (\gamma_s e_1')\,(\gamma_s e_2')) \in \mathcal{E}[\![B]\!]\delta$

$\boxed{\mathcal{R}}^{\mathcal{N}_\mathrm{R}} * (\gamma_i e_1, \gamma_s e_1') \in \mathcal{E}[\![A \to B]\!]\delta * (\gamma_i e_2, \gamma_s e_2') \in \mathcal{E}[\![A]\!]\delta$

$((\gamma_i e_1)\,(\gamma_i e_2), (\gamma_s e_1')\,(\gamma_s e_2')) \in \mathcal{E}[\![B]\!]\delta$

$\boxed{\mathcal{R}}^{\mathcal{N}_\mathrm{R}}$

$\gamma_i e_1 \preceq \gamma_s e_1' : \{v, w.\, (v, w) \in \mathcal{V}[\![A \to B]\!]\delta\}$

$\gamma_i e_2 \preceq \gamma_s e_2' : \{v, w.\, (v, w) \in \mathcal{V}[\![A]\!]\delta\}$

$((\gamma_i e_1)\,(\gamma_i e_2), (\gamma_s e_1')\,(\gamma_s e_2')) \in \mathcal{E}[\![B]\!]\delta$

By Lemma 103.

$\boxed{\mathcal{R}}^{\mathcal{N}_\mathrm{R}}$

$\gamma_i e_1 \preceq \gamma_s e_1' : \{v, w.\, (v, w) \in \mathcal{V}[\![A \to B]\!]\delta\}$

$(v_2, v_2') \in \mathcal{V}[\![A]\!]\delta$

$((\gamma_i e_1)\, v_2, (\gamma_s e_1')\, v_2') \in \mathcal{E}[\![B]\!]\delta$

By Lemma 103.

$\boxed{\mathcal{R}}^{\mathcal{N}_\mathrm{R}}$

$(v_1, v_1') \in \mathcal{V}[\![A \to B]\!]\delta$

$(v_2, v_2') \in \mathcal{V}[\![A]\!]\delta$

$(v_1\, v_2, v_1'\, v_2') \in \mathcal{E}[\![B]\!]\delta$

Which follows by definition.

□

Exercise 115 Prove the remaining compatibility lemmas. •

Adequacy Finally, let us now show that our binary logical relation can be used to prove contextual refinement. Concretely, we prove:

Theorem 110 (Compatibility of $\preceq$ w.r.t. $\leq_{\mathrm{ctx}}$).
If $\Delta\,;\Gamma \vDash e_i \preceq e_s : A$, then $\Delta\,;\Gamma \vdash e_i \leq_{ctx} e_s : A$.

Proof. Let $C : (\Delta\,;\Gamma\,;A) \rightsquigarrow (\emptyset\,;\emptyset\,;\mathbf{1})$ such that $(C[e_i], h) \downarrow ((), h')$. Then by Theorem 108, we have $\vDash C[e_i] \preceq C[e_s] : \mathbf{1}$. We allocate the required ghost state such that $\mathsf{True} \vdash \,\Rrightarrow \exists \gamma_{\mathrm{heap}}, \gamma_{\mathrm{expr}}.\, \mathcal{R}_{\gamma_{\mathrm{expr}}, \gamma_{\mathrm{heap}}, C[e_2], h} * \mathsf{spec}(C[e_2])$. and put $\mathcal{R}$ into an invariant, obtaining $\mathsf{True} \vdash \,\Rrightarrow_\top \exists \gamma_{\mathrm{heap}}, \gamma_{\mathrm{expr}}.\, \boxed{\mathcal{R}_{\gamma_{\mathrm{expr}}, \gamma_{\mathrm{heap}}, C[e_s], h}}^{\mathcal{N}_\mathrm{R}} * \mathsf{spec}(C[e_s])$. By $\vDash C[e_i] \preceq C[e_s] : \mathbf{1}$ (and duplicability of invariants), we thus obtain:

$$\mathsf{True} \vdash \,\Rrightarrow_\top \exists \gamma_{\mathrm{heap}}, \gamma_{\mathrm{expr}}.\, \mathsf{wp}\; C[e_i] \left\{v.\, \exists w.\, \boxed{\mathcal{R}_{\gamma_{\mathrm{expr}}, \gamma_{\mathrm{heap}}, C[e_s], h}}^{\mathcal{N}_\mathrm{R}} * \mathsf{spec}(w) * v = w = ()\right\}$$

By opening the invariant in the post condition, we derive:

$$\mathsf{True} \vdash \,\Rrightarrow_\top \exists \gamma_{\mathrm{heap}}, \gamma_{\mathrm{expr}}.\, \mathsf{wp}\; C[e_i] \left\{v.\, \exists w, h''.\, \Rrightarrow_\top (C[e_s], h) \rightsquigarrow^* (w, h'') * w = ()\right\}$$

Using a slight generalization of the adequacy theorem of Iris, we can then use the execution $(C[e_i], h) \downarrow ((), h')$ to get to the postcondition and obtain the execution of e_s. $\square$

Exercise 116 Just as with our unary logical relation, we can use the binary semantic model to reason about results that do not simply follow from the compatiblity lemma, for instance to show interesting and non-trivial contextual refinements!

Come up with an interesting refinement that you would like to prove, and verify it. Bonus points if it makes non-trivial use of impredicate invariants! $\bullet$

9 Iris's Model

After spending several sections discussing the features of Iris, in this section, we take a closer look at the model of Iris. That is, we discuss how Iris's propositions and its connectives are defined. Unfortunately, the full model of Iris is quite a mouthful and a thorough discussion is beyond the scope of these notes. Instead, we discuss a simplified model without impredicative invariants (in Section 9.1) and then sketch what is required to extend the simplified model to handle them (in Section 9.2).

9.1 Simplified Model

The model of Iris consists of two parts: the *program logic* and the *base logic*. The program logic will be concerned with programs, heaps, and the weakest precondition. The base logic is agnostic about all of those and merely concerned with resources, resource management, and step-indexing. The base logic forms a simple foundation, upon which we build the program logic.

9.1.1 Base Logic

In the previous sections, we have worked with the type of Iris's propositions *iProp*. In the model of Iris, *iProp* is obtained from a more general construction: *uniform predicates over a unital resource algebra M*, written $\mathit{UPred}(M)$. One obtains *iProp* from $\mathit{UPred}(M)$, as explained below, by picking a specific unital resource algebra M.

Uniform Predicates $\boxed{\mathit{UPred}(M)}$

The type $\mathit{UPred}(M)$ consists of predicates over step-indices and resources (from M) which are down-closed with respect to the step-index and up-closed with respect to the resource:

$$\mathit{UPred}(M) := \{P \in \mathbb{P}(\mathbb{N}, M) \mid \forall (n,a) \in P.\, \forall m, b.\, m \leq n \Rightarrow a \preccurlyeq b \Rightarrow (m,b) \in P\}$$

The entailment relation $P \vdash Q$ is given by

$$P \vdash Q := \forall n, a.\, a \in \overline{\mathcal{V}} \Rightarrow (n,a) \in P \Rightarrow (n,a) \in Q$$

and we define the connectives:

$$\begin{aligned}
\phi &:= \{(n,a) \mid \phi\} \\
P \wedge Q &:= \{(n,a) \mid (n,a) \in P \wedge (n,a) \in Q\} \\
P \vee Q &:= \{(n,a) \mid (n,a) \in P \vee (n,a) \in Q\} \\
P \Rightarrow Q &:= \{(n,a) \mid \forall m, b.\, m \leq n \Rightarrow a \preccurlyeq b \Rightarrow (m,b) \in P \Rightarrow (m,b) \in Q\} \\
\forall x : X.\, P(x) &:= \{(n,a) \mid \forall x : X.\, (n,a) \in P(x)\} \\
\exists x : X.\, P(x) &:= \{(n,a) \mid \exists x : X.\, (n,a) \in P(x)\} \\
P * Q &:= \{(n,a) \mid \exists a_1, a_2.\, a = a_1 \cdot a_2 \wedge (n,a_1) \in P \wedge (n,a_2) \in Q\} \\
P \mathrel{-\!\!*} Q &:= \left\{(n,a) \,\middle|\, \forall m, b.\, m \leq n \Rightarrow a \cdot b \in \overline{\mathcal{V}} \Rightarrow (m,b) \in P \Rightarrow (m, a \cdot b) \in Q\right\} \\
\square\, P &:= \{(n,a) \mid (n, |a|) \in P\} \\
\triangleright P &:= \{(n,a) \mid \forall m < n.\, (m,a) \in P\} \\
\mathrel{\mid\!\Rrightarrow} P &:= \{(n,a) \mid a \rightsquigarrow \{b \mid (n,b) \in P\}\} \\
\mathsf{Own}\,(a) &:= \{(n,b) \mid a \preccurlyeq b\}
\end{aligned}$$

Soundness Equipped with the definition of uniform predicates, we can prove many of the rules that we have encountered in previous sections:

$$\textsc{Refl}\quad P \vdash P$$

$$\textsc{Trans}\quad \frac{P \vdash Q \qquad Q \vdash R}{P \vdash R}$$

$$\textsc{Pure}\quad \frac{\phi}{P \vdash \phi}$$

$$\textsc{FromPure}\quad \frac{P \vdash \phi \qquad \phi \Rightarrow (P \vdash Q)}{P \vdash Q}$$

$$\textsc{AndElimL}\quad P \wedge Q \vdash P$$

$$\textsc{AndElimR}\quad P \wedge Q \vdash Q$$

$$\textsc{AndIntro}\quad \frac{P \vdash Q \qquad P \vdash R}{P \vdash Q \wedge R}$$

$$\textsc{OrIntroL}\quad P \vdash P \vee Q$$

$$\textsc{OrIntroR}\quad Q \vdash P \vee Q$$

$$\textsc{OrElim}\quad \frac{P \vdash R \qquad Q \vdash R}{P \vee Q \vdash R}$$

$$\textsc{AllIntro}\quad \frac{\forall x : X.\, (P \vdash Q(x))}{P \vdash \forall x : X.\, Q(x)}$$

$$\textsc{AllElim}\quad \frac{y : X}{(\forall x : X.\ P(x)) \vdash P(y)}$$

$$\textsc{ExistIntro}\quad \frac{y : X \qquad P \vdash Q(y)}{P \vdash \exists x : X.\, Q(x)}$$

$$\textsc{ExistElim}\quad \frac{\forall x : X.\, (P(x) \vdash Q)}{\exists x : X.\, P(x) \vdash Q}$$

$$\textsc{WandIntro}\quad \frac{P * Q \vdash R}{P \vdash Q \mathrel{-\!\!*} R}$$

$$\textsc{WandElim}\quad \frac{P \vdash Q \mathrel{-\!\!*} R}{P * Q \vdash R}$$

$$\textsc{SepWeaken}\quad P * Q \vdash P$$

$$\textsc{SepTrue}\quad P \vdash P * \mathsf{True}$$

$$\textsc{SepComm}\quad P * Q \vdash Q * P$$

$$\textsc{SepSplit}\quad \frac{P \vdash P' \qquad Q \vdash Q'}{P * Q \vdash P' * Q'}$$

$$\textsc{SepAssoc}\quad P * (Q * R) \dashv\vdash (P * Q) * R$$

$$\textsc{PersElim}\quad \Box P \vdash P$$

$$\textsc{PersMono}\quad \frac{P \vdash Q}{\Box P \vdash \Box Q}$$

$$\textsc{PersPure}\quad \phi \vdash \Box \phi$$

$$\textsc{PersAndSep}\quad (\Box P) \wedge Q \vdash (\Box P) * Q$$

$$\textsc{PersIdemp}\quad \Box P \vdash \Box \Box P$$

$$\textsc{PersAll}\quad \forall x : X.\ \Box P(x) \vdash \Box \forall x : X.\, P(x)$$

$$\textsc{PersExists}\quad \Box \exists x : X.\, P(x) \vdash \exists x : X.\ \Box P(x)$$

$$\textsc{LaterIntro}\quad P \vdash \triangleright P$$

$$\textsc{LaterMono}\quad \frac{P \vdash Q}{\triangleright P \vdash \triangleright Q}$$

$$\textsc{Loeb}\quad \frac{\triangleright P \vdash P}{\vdash P}$$

$$\textsc{LaterSep}\quad \triangleright(P * Q) \dashv\vdash \triangleright P * \triangleright Q$$

$$\textsc{LaterExists}\quad \frac{X \text{ non-empty}}{\triangleright(\exists x : X.\, P(x)) \dashv\vdash \exists x : X.\ \triangleright P(x)}$$

$$\textsc{LaterAll}\quad \triangleright(\forall x : X.\, P(x)) \dashv\vdash \forall x : X.\ \triangleright P(x)$$

$$\textsc{LaterPers}\quad \triangleright \Box P \dashv\vdash \Box \triangleright P$$

$$\textsc{UpdReturn}\quad P \vdash \Rrightarrow P$$

$$\textsc{UpdBind}\quad \Rrightarrow P * (P \mathrel{-\!\!*} \Rrightarrow Q) \vdash \Rrightarrow Q$$

Exercise 117 Prove the soundness of the rules Refl, Trans, Pure, AndIntro, ExistElim, SepSplit, WandElim, PersElim, PersAndSep, LaterIntro, LaterMono, Loeb, UpdReturn, and UpdBind. ●

Ghost State Notably, none of the rules above mention ghost state or resources in any form. Let us have a closer look at resources in $UPred(M)$. Instead of the ghost state connective $\boxed{a}^{\gamma}$, the type $UPred(M)$ has the ownership connective $\mathsf{Own}\,(a)$. This ownership connective enjoys the following rules:

$$\textsc{OwnEmpty}\quad \mathsf{True} \vdash \mathsf{Own}\,(\varepsilon) \qquad \textsc{OwnPers}\quad \mathsf{Own}\,(a) \vdash \Box\,\mathsf{Own}\,(|a|) \qquad \textsc{OwnSep}\quad \mathsf{Own}\,(a) * \mathsf{Own}\,(b) \dashv\vdash \mathsf{Own}\,(a \cdot b)$$

$$\textsc{OwnValid}\quad \mathsf{Own}\,(a) \vdash a \in \overline{\mathcal{V}} \qquad \textsc{OwnUpd}\quad \frac{a \rightsquigarrow B}{\mathsf{Own}\,(a) \vdash \Rrightarrow \exists b \in B.\, \mathsf{Own}\,(b)}$$

Exercise 118 Verify OwnEmpty, OwnPers, OwnSep, OwnValid, and OwnUpd. •

Iris Propositions $\boxed{iProp}$

From the general construction of $UPred(M)$, we obtain the type of Iris propositions $iProp$ and the ghost state ownership connective $\boxed{a}^{\gamma}$ by picking a specific resource algebra:

$$iProp := UPred(\mathbb{N} \xrightharpoonup{\mathsf{fin}} \mathcal{M})$$

where $\mathcal{M}$ combines the resource algebras that we want to use (see below). By using finite maps to $\mathcal{M}$, we can associate ghost state with a name $\gamma : \mathbb{N}$ and have multiple instances of the same ghost state (*e.g.*, $\mathsf{mono}_{\gamma_1}(n_1)$, $\mathsf{mono}_{\gamma_2}(n_2)$, and $\mathsf{mono}_{\gamma_3}(n_3)$).

To understand better how one defines $\boxed{a}^{\gamma}$ in terms of $\mathsf{Own}\,(b)$, we need to make precise what the resource algebra $\mathcal{M}$ is. Unfortunately, the precise definition of $\mathcal{M}$ is a bit gnarly (see [5]), because we need to combine all the resource algebras that we want to use into a single resource algebra $\mathcal{M}$. To nevertheless illustrate the connection between $\boxed{a}^{\gamma}$ and $\mathsf{Own}\,(b)$, we pick a single kind of resources here, $\mathcal{M} := Auth(\mathbb{N}, \max)$, for the sake of simplicity. For this concrete choice of ghost state, we can define:

$$\boxed{a}^{\gamma} := \mathsf{Own}\,([\gamma \mapsto a])$$

where $[\gamma \mapsto a]$ is a singleton map. We can then prove the standard ghost state rules:

$$\textsc{ResAlloc}\quad \frac{a \in \overline{\mathcal{V}}}{\vdash \Rrightarrow \exists \gamma.\, \boxed{a}^{\gamma}} \qquad \textsc{ResSep}\quad \boxed{a}^{\gamma} * \boxed{b}^{\gamma} \dashv\vdash \boxed{a \cdot b}^{\gamma} \qquad \textsc{ResPers}\quad \boxed{a}^{\gamma} \vdash \Box\, \boxed{|a|}^{\gamma} \qquad \textsc{ResValid}\quad \boxed{a}^{\gamma} \vdash a \in \overline{\mathcal{V}}$$

$$\textsc{ResUpd}\quad \frac{a \rightsquigarrow B}{\boxed{a}^{\gamma} \vdash \Rrightarrow \exists b \in B.\, \boxed{b}^{\gamma}}$$

Exercise 119 Verify the rules ResAlloc, ResSep, ResPers, ResValid, and ResUpd. •

Guarded Fixpoints The last piece missing of the base logic are *guarded fixpoints* $\mu x.P$. The general construction of these fixpoints requires some additional machinery that we touch on in Section 9.2. But for fixpoints of type $X \to UPred(M)$, we can sketch their construction. For a function $F : (X \to UPred(M)) \to (X \to UPred(M))$, we define step-indexed approximations of its fixpoint by:

$$f_0(x) := \mathsf{True} \qquad f_{n+1}(x) := F(f_n)(x) \qquad f(x) := \{(n, r) \mid (n, r) \in f_{n+1}(x)\}$$

If every recursive occurrence in F is guarded by a later, one can show that it is *guarded* in the following sense:

$$\text{guarded}(F) := \forall g_1, g_2.\, \triangleright(\forall x.\, g_1(x) \iff g_2(x)) \vdash \forall x.\, F(g_1)(x) \iff F(g_2)(x)$$

which suffices to prove the fixpoint equation:

Lemma 111. *If F is guarded, then f is a fixpoint, meaning $F(f)(x) = f(x)$.*

9.1.2 Program Logic

Let us now turn to the *program logic* of Iris, which we define using the connectives of the base logic. We start with the weakest precondition.

Weakest Precondition $\boxed{\mathsf{wp}\ e\ \{v.\, Q(v)\}}$

We define the weakest precondition as a guarded fixpoint:

$$\begin{aligned}
\mathsf{wp}\ v\ \{v.\, Q(v)\} &:= \Rrightarrow Q(v) \\
\mathsf{wp}\ e\ \{v.\, Q(v)\} &:= \forall h.\, \mathrm{SI}(h) \mathrel{-\!\!*} \Rrightarrow \text{progress}(e, h) && \text{if } e \notin \mathit{Val} \\
&\qquad * \ \forall e', h'.\, (e, h) \leadsto (e', h') \mathrel{-\!\!*} \Rrightarrow \triangleright\, \mathrm{SI}(h') * \mathsf{wp}\ e'\ \{v.\, Q(v)\}
\end{aligned}$$

In the base case, when the argument is a value v, we have to prove the postcondition $Q(v)$ (after potentially) updating the ghost state. Otherwise, if e is a proper expression, we get to assume the state interpretation $\mathrm{SI}(h)$ (explained below) and have to show two conditions: (1) the current expression e can make progress in the heap h where $\text{progress}(e, h) := \exists e', h'.\, (e, h) \leadsto (e', h')$ and (2) for any successor expression e' and heap h', we have to show the weakest precondition and the state interpretation after *an update to the ghost state* and after *a later*.

The updates in both cases makes sure that we can always update our ghost state when we prove a weakest precondition. These updates are instrumental for working with the state interpretation below and for verifying code which relies on auxiliary ghost state.

The later in the second case ensures that the weakest precondition can be defined as a guarded fixpoint. Moreover, it ties program steps to laters in our program logic (*i.e.*, in the rules LaterPureStep, LaterNew, LaterLoad, and LaterStore). In fact, this later in the definition of the weakest precondition is responsible for the intuition: *"$\triangleright P$ means P holds after the next step of computation"*. More concretely, if one proves a weakest precondition $\mathsf{wp}\ e\ \{v.\, Q(v)\}$ under the assumption $\triangleright P$ then, after the next step of computation, the goal becomes $\triangleright \mathsf{wp}\ e'\ \{v.\, Q(v)\}$. We can then use the rule LaterMono to remove the later in front of $\mathsf{wp}\ e'\ \{v.\, Q(v)\}$ *and* in front of $\triangleright P$.

The State Interpretation The state interpretation answers two questions: (1) "how can we prove progress of e in arbitrary heaps?" and (2) "how does the points-to assertion $\ell \mapsto v$ relate to the weakest precondition?" With the state interpretation $\mathrm{SI}(h)$, we can control which heaps h we have to consider in the weakest precondition. It is like a small invariant that we maintain throughout the execution of e (*i.e.*, we will show that it holds initially and the weakest precondition preserves it for every step). We use the state interpretation $\mathrm{SI}(h)$ to tie the heap h in the weakest precondition to the points-to assertions $\ell \mapsto v$ in our program logic.

To tie $\mathrm{SI}(h)$ and $\ell \mapsto v$ together, we use *ghost state*. More concretely, we use the $\mathsf{ExMap} := \mathit{Auth}(\mathit{Loc} \xrightharpoonup{\text{fin}} \mathit{Ex}(\mathit{Val}))$ resource algebra and fix a ghost name γ_{heap}:

$$\mathrm{SI}(h) := \mathsf{EGM}_{\gamma_{\text{heap}}}(h) \qquad\qquad \ell \mapsto v := \ell \mapsto_{\gamma_{\text{heap}}} v$$

From the ExMap resource algebra, we then obtain the following ghost theory for the state interpretation and the points-to assertions:

$$\ell \mapsto v * \ell \mapsto w \vdash \mathsf{False} \qquad\qquad \ell \mapsto v * \mathrm{SI}(h) \vdash h(\ell) = v$$

$$\ell \mapsto v * \mathrm{SI}(h) \vdash \Rrightarrow \ell \mapsto w * \mathrm{SI}(h[\ell \mapsto w]) \qquad\qquad \frac{\ell \notin \mathsf{dom}\, h}{\mathrm{SI}(h) \vdash \Rrightarrow \mathrm{SI}(h[\ell \mapsto v]) * \ell \mapsto v}$$

Exercise 120 Suppose we use $\mathsf{GhostMap}$ instead of ExMap in the state interpretation. Are there interesting examples that make use of the additional fractions? •

Soundness With the state interpretation in hand, we can proceed to prove the soundness of the rules of the program logic.

Value
$$Q(v) \vdash \mathsf{wp}\ v\ \{w.\, Q(w)\}$$

Wand
$$(\forall v.\, Q(v) \mathrel{-\!\!*} Q'(v)) * \mathsf{wp}\ e\ \{w.\, Q(w)\} \vdash \mathsf{wp}\ e\ \{w.\, Q'(w)\}$$

WpBind
$$\mathsf{wp}\ e\ \{v.\, \mathsf{wp}\ K[v]\ \{w.\, Q(w)\}\} \vdash \mathsf{wp}\ K[e]\ \{w.\, Q(w)\}$$

WpUpd
$$\Rrightarrow \mathsf{wp}\ e\ \{v.\, Q(v)\} \vdash \mathsf{wp}\ e\ \{v.\, Q(v)\}$$

LaterPureStep
$$\frac{e \rightarrow_{\mathsf{pure}} e'}{\triangleright \mathsf{wp}\ e'\ \{v.\, P(v)\} \vdash \mathsf{wp}\ e\ \{v.\, P(v)\}}$$

LaterNew
$$\triangleright(\forall \ell.\, \ell \mapsto v \mathrel{-\!\!*} Q(\ell)) \vdash \mathsf{wp}\ \mathsf{new}(v)\ \{w.\, Q(w)\}$$

LaterLoad
$$\ell \mapsto v * \triangleright(\ell \mapsto v \mathrel{-\!\!*} Q(v)) \vdash \mathsf{wp}\ !\,\ell\ \{w.\, Q(w)\}$$

LaterStore
$$\ell \mapsto v * \triangleright(\ell \mapsto w \mathrel{-\!\!*} Q()) \vdash \mathsf{wp}\ \ell \leftarrow w\ \{w.\, Q(w)\}$$

The rules Value, Wand, WpBind, and WpUpd follow from the definition of the weakest precondition. They are completely agnostic about the language that we use (aside from knowing that it exists) and agnostic about the state interpretation that we have chosen.

Exercise 121 Prove the rules Value, Wand, WpBind, and WpUpd.
Hint: Two of the rules require Löb induction. •

The rules LaterPureStep, LaterNew, LaterLoad, and LaterStore are specific to the language. They depend on our concrete choice of the state interpretation and constructs of the language. We define the notion of pure steps $e \to_{\mathsf{pure}} e'$ as:

$$e \to_{\mathsf{pure}} e' := (\forall h.\, (e,h) \leadsto (e',h)) \land (\forall h, h'', e''.\, (e,h) \leadsto (e'',h'') \Rightarrow h'' = h \land e'' = e')$$

The first part ensures progress and the second part that there are no possible steps to expressions which are not e'.

Exercise 122 Prove the rules LaterPureStep, LaterNew, LaterLoad, and LaterStore.
●

9.1.3 Adequacy

Now that we have seen the model of Iris propositions and the model of the weakest precondition, one interesting question remains: "What do we prove when we prove entailments in Iris?" To answer this question, we will prove a series of adequacy results that can be used to lift results proven in Iris to results about programs (and their executions) in the meta logic. Concretely, we prove the following adequacy results:

Lemma 112 (Adequacy of the Program Logic). *If $(e,h) \leadsto^n (e',h')$ and $\vdash \Rrightarrow SI(h) * \mathsf{wp}\, e\, \{v.\, \phi(v)\}$, then (e',h') is either progressive or e' is a value v and $\phi(v)$ holds.*

Proof. We unfold the weakest precondition n times and obtain the desired result about e' and h' under a number of laters and updates. The main result then follows with Lemma 113. □

Note that technically these theorems need to quantify over the ghost name γ_{heap} that is chosen for the heap. We have omitted it here for simplicity.

Lemma 113 (Adequacy for the Base Logic). *If $\vdash (\Rrightarrow \triangleright)^n \phi$, then ϕ holds.*

The adequacy theorem for the base logic states that pure propositions (under a potential nesting of later modalities and update modalities) are true at the meta level. From it and the adequacy result of the program logic (*i.e.*, Lemma 112), we can deduce the following two corollaries:

Theorem 114 (Safety). *If $\{\mathsf{True}\}\ e\ \{v.\, \phi(v)\}$, then e safely terminates in (potentially multiple) values v such that $\phi(v)$ holds, or e diverges.*

Theorem 115 (Consistency). *It is impossible to prove falsity, meaning $\mathsf{True} \nvdash \mathsf{False}$.*

9.2 Full Model

Notably, what is still missing from our discussion above is how invariants and timelessness are integrated into the program logic. As it turns out, compared to what we have discussed so far, the addition is a (relatively) localized change:

we use fancy updates ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} P$ *in the definition of the weakest precondition.*

Since fancy updates (as we will recall below) have built-in support for invariants and timelessness, the weakest precondition will inherit this support.

Integrating Fancy Updates At a high level, the integration of fancy updates into the weakest precondition is very simple: we replace every update ${\Rrightarrow} P$ with a fancy update ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} P$. This change, however, means that we have to decide which masks $\mathcal{E}_1$ and $\mathcal{E}_2$ we want to use and where we want to place them. As it turns out, there is ample room for different design decisions as it comes to the masks and these decisions impact *when* and *how long* invariants can be opened. For example, the choice that has been made for the weakest precondition $\mathsf{wp}^{\mathcal{E}}\ e\ \{v.\, Q(v)\}$ that we have been working with is that invariants can be opened around the entire expression—from the start to the postcondition. In contrast, for a concurrent language, we would have to ensure that invariants can only be opened for a single step (otherwise other threads could observe inconsistent states).

While the choice of masks in the weakest precondition is important, it will *not* be the focus of these notes. We refer the reader to Jung et al. [5] for a more detailed discussion of the construction of the weakest precondition with fancy updates. Instead, in these notes, we will focus on the model of the fancy updates and how it allows us to work with invariants and timelessness.

9.2.1 Fancy Updates

Before we proceed to the model of fancy updates ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} P$, let us briefly recall their meaning and their rules. In short, the idea of a fancy update ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} P$ is, among others, that P holds after *opening* all the invariants which are in $\mathcal{E}_1$, performing some ghost state updates, and then *closing* all the invariants that are in $\mathcal{E}_2$. This idea is also reflected by the rules for the modality:

FancyInv
$$\frac{\mathcal{N} \subseteq \mathcal{E}}{\boxed{P}^{\mathcal{N}} \vdash {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}\setminus\mathcal{N}} \triangleright P * (\triangleright P \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}}{\Rrightarrow}^{\mathcal{E}} \mathsf{True})}$$

FancyReturn
$$P \vdash {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}} P$$

FancyBind
$${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} P * (P \mathrel{-\!\!*} {}^{\mathcal{E}_2}{\Rrightarrow}^{\mathcal{E}_3} Q) \vdash {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_3} Q$$

FancyUpd
$${\Rrightarrow} P \vdash {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}} P$$

FancyMaskFrame
$${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1 \uplus \mathcal{E}}{\Rrightarrow}^{\mathcal{E}_2 \uplus \mathcal{E}} P$$

FancyIntroMask
$$\frac{\mathcal{E}_2 \subseteq \mathcal{E}_1}{{}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_1} P \vdash {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2}\, {}^{\mathcal{E}_2}{\Rrightarrow}^{\mathcal{E}_1} P}$$

FancyTimeless
$$\frac{\mathsf{timeless}(P)}{\triangleright P \vdash {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}} P}$$

FancyWp
$${}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}} \mathsf{wp}^{\mathcal{E}}\ e\ \{v.\, Q(v)\} \vdash \mathsf{wp}^{\mathcal{E}}\ e\ \{v.\, Q(v)\}$$

The most important rule is, of course, FancyInv. The rule FancyInv allows to open the invariant $\mathcal{N}$ (as long as its namespace is still contained in the mask) and we get $\triangleright P$ and a way to close the invariant again (*i.e.*, to restore the mask again) ($\triangleright P \twoheadrightarrow {}^{\mathcal{E}\setminus\mathcal{N}}{\Rrightarrow}^{\mathcal{E}} \mathsf{True}$). Together with the other rules, we can use this rule to open, modify, and then close invariants again all as part of proving a fancy update ${}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}} P$. Intuitively, the fancy update ${}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}} P$ collects all the updates to the invariants in $\mathcal{E}$ and updates to ghost state that we do while proving ${}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}} P$ and we can then apply them by executing the fancy update at a weakest precondition.

The rules FancyReturn and FancyBind are mirrored from the update modality $\Rrightarrow P$ and offer us similar compositionality. The rule FancyUpd allows us to turn an update into a fancy update, the rule FancyMaskFrame allows us to frame part of our mask (to open the invariants in $\mathcal{E}$ later), and the rule FancyIntroMask allows us to break a fancy update into two separate fancy updates with a smaller mask. The rule FancyTimeless allows us to eliminate the later in front of timeless propositions and the rule FancyWp allows us to eliminate a fancy update at a weakest precondition.[4]

Exercise 123 Derive the rule TimelessStrip. •

Timelessness To discuss the model of fancy updates, we need to shed some light on timelessness in Iris. Intuitively, a proposition P is timeless if its complete behavior is determined by its behavior at step-index 0. In terms of the base logic, we express this property as:

$$\mathsf{timeless}(P) := \triangleright P \vdash \triangleright \mathsf{False} \vee P$$

The definition says that P is timeless if $\triangleright P$ either means the step-index is 0 (the left branch) or P already holds at the current step-index (the right branch).

Exercise 124 Why can we not define $\mathsf{timeless}(P) := \triangleright P \vdash P$? •

Exercise 125 Prove the following rules for $\mathit{UPred}(M)$ using the definition of the model:

$$\textsc{TimelessPure}\quad \mathsf{timeless}(\phi) \qquad\qquad \textsc{TimelessOwn}\quad \mathsf{timeless}(\mathsf{Own}\ (b))$$

•

Exercise 126 Derive

$$\textsc{TimelessPers}\ \frac{\mathsf{timeless}(P)}{\mathsf{timeless}(\Box P)} \qquad \textsc{TimelessSep}\ \frac{\mathsf{timeless}(P) \qquad \mathsf{timeless}(Q)}{\mathsf{timeless}(P * Q)} \qquad \textsc{TimelessOr}\ \frac{\mathsf{timeless}(P) \qquad \mathsf{timeless}(Q)}{\mathsf{timeless}(P \vee Q)}$$

$$\textsc{TimelessAnd}\ \frac{\mathsf{timeless}(P) \qquad \mathsf{timeless}(Q)}{\mathsf{timeless}(P \wedge Q)} \qquad \textsc{TimelessAll}\ \frac{\forall x.\, \mathsf{timeless}(P(x))}{\mathsf{timeless}(\forall x.\, P)} \qquad \textsc{TimelessExists}\ \frac{\forall x.\, \mathsf{timeless}(P(x))}{\mathsf{timeless}(\exists x.\, P)}$$

•

[4]This rule requires the above mentioned change to the definition of the weakest precondition compared to the simplified model presented in the previous section.

So why can we eliminate laters from timeless propositions when we prove weakest preconditions and fancy updates? The reason is that we are working with a hierarchy of monads in Iris: the weakest precondition $\mathsf{wp}\, e\, \{v.\, Q(v)\}$ is defined in terms of fancy updates ${}^{\mathcal{E}_1}\!\!\mid\!\!\Rrightarrow^{\mathcal{E}_2} P$, which in turn are built up from a composition of the update monad $\mid\!\!\Rrightarrow P$ and the *timeless monad* $\diamond\, P$.

The timeless monad $\diamond\, P := \triangleright\, \mathsf{False} \vee P$ satisfies the following properties:

$$\textsc{TimelessReturn}\quad P \vdash \diamond\, P \qquad\qquad \textsc{TimelessBind}\quad (\diamond\, P) * (P \mathrel{-\!\!*} \diamond\, Q) \vdash \diamond\, Q \qquad\qquad \textsc{TimelessLater}\quad \frac{\mathsf{timeless}(P)}{\triangleright\, P \vdash \diamond\, P}$$

Various connectives of the logic, as we have seen, are timeless and the timeless monad allows us to strip laters off of them.

Exercise 127 Prove TimelessReturn, TimelessBind, and TimelessLater. ●

Fancy Update Model Given the timelessness monad, we are now fully equipped to discuss the model of the fancy update modality ${}^{\mathcal{E}_1}\!\!\mid\!\!\Rrightarrow^{\mathcal{E}_2} P$:

$$ {}^{\mathcal{E}_1}\!\!\mid\!\!\Rrightarrow^{\mathcal{E}_2} P := \mathsf{wsat} * \boxed{\mathcal{E}_1}^{\gamma_{\mathrm{en}}} \mathrel{-\!\!*} \mid\!\!\Rrightarrow \diamond\, (\mathsf{wsat} * \boxed{\mathcal{E}_2}^{\gamma_{\mathrm{en}}} * P)$$

Let us unpack this definition step by step. First, similar to the model of invariants for our logical relation in Section 4, we have a notion of *world satisfaction* wsat (discussed below) that will store the invariants. And just as before, world satisfaction is threaded through our proofs (here by fancy updates). Besides world satisfaction, the fancy updates consist of a composition of the monad for ghost state updates $\mid\!\!\Rrightarrow P$ and the monad for timelessness $\diamond\, P$. Thus, it inherits many of the properties of both monads. The last piece to the puzzle is the ghost state $\boxed{\mathcal{E}}^{\gamma_{\mathrm{en}}}$. This piece of ghost state tracks which invariants are currently closed. We will use it, together with an additional piece of ghost state $\boxed{\mathcal{E}}^{\gamma_{\mathrm{dis}}}$, in the definition of wsat to distinguish open and closed invariants.

Exercise 128 Derive the rules FancyReturn, FancyBind, FancyUpd, and FancyTimeless. ●

World Satisfaction Ghost Theory Before we discuss the model of world satisfaction, let us turn to the ghost theory that we need to make invariants and fancy updates work. In this ghost theory, invariants will have a *name* $\iota : \mathbb{N}$ instead of a namespace $\mathcal{N}$. The namespace invariants are obtained as $\boxed{P}^{\mathcal{N}} := \exists \iota \in \mathcal{N}.\, \boxed{P}^{\iota}$.

The ghost theory of world satisfaction wsat has three interesting rules:

$$\textsc{InvAlloc}\quad \frac{\mathcal{E} \text{ infinite}}{\mathsf{wsat} * (\triangleright\, P) \vdash \mid\!\!\Rrightarrow \exists \iota \in \mathcal{E}.\, \boxed{P}^{\iota} * \mathsf{wsat}} \qquad\qquad \textsc{InvOpen}\quad \boxed{P}^{\iota} * \mathsf{wsat} * \boxed{\{\iota\}}^{\gamma_{\mathrm{en}}} \vdash \mid\!\!\Rrightarrow (\triangleright\, P) * \mathsf{wsat} * \boxed{\{\iota\}}^{\gamma_{\mathrm{dis}}}$$

$$\textsc{InvClose}\quad \boxed{P}^{\iota} * \mathsf{wsat} * (\triangleright\, P) * \boxed{\{\iota\}}^{\gamma_{\mathrm{dis}}} \vdash \mid\!\!\Rrightarrow \mathsf{wsat} * \boxed{\{\iota\}}^{\gamma_{\mathrm{en}}}$$

The rule InvAlloc allows us to allocate a new invariant P if we own $\triangleright\, P$. The rule InvOpen allows us to open the invariant ι and obtain $\triangleright\, P$. The rule InvClose allows us the close the invariant ι by returning $\triangleright P$. To use InvOpen, we need to know that ι is currently enabled and we get back a token $\boxed{\{\iota\}}^{\gamma_{\text{dis}}}$ witnessing that ι is now disabled. To use InvClose, we need to know that ι is currently disabled and we get back a token $\boxed{\{\iota\}}^{\gamma_{\text{en}}}$ witnessing that ι is now enabled.

The ghost state for the enabled and disabled invariants are sets of invariant names $\mathcal{E}$ with $\uplus$ as the monoid operation. More concretely, for the enabled invariants γ_{en} we pick the resource algebra $(\mathbb{P}(\mathbb{N}), \uplus)$ and for the disabled invariants, we pick the resource algebra $(\mathbb{P}^{\text{fin}}(\mathbb{N}), \uplus)$. At any given point, we can have infinitely many enabled tokens (*e.g.*, with $\boxed{\top}^{\gamma_{\text{en}}}$), but there can only ever be finitely many disabled tokens (*i.e.*, $\boxed{\{\iota\}}^{\gamma_{\text{dis}}}$), which is sufficient because there will only be finitely many invariants at a time.

Exercise 129 Derive the rules FancyMaskFrame, FancyIntroMask, and FancyInv. •

Broken World Satisfaction Given the intuition about invariants and the ghost theory of world satisfaction, let us now attempt to define them:

$$\boxed{P}^{\iota} := \boxed{\circ[\iota \mapsto \mathsf{ag}(P)]}^{\gamma_{\text{inv}}}$$
$$\mathsf{wsat} := \exists I : \mathbb{N} \xrightarrow{\text{fin}} iProp.\ \boxed{\bullet[\iota \mapsto \mathsf{ag}(P) \mid \iota \mapsto P \in I]}^{\gamma_{\text{inv}}} * \mathop{\ast\!\!\!\ast}_{\iota \mapsto P \in I} (\triangleright P * \boxed{\{\iota\}}^{\gamma_{\text{dis}}}) \lor (\boxed{\{\iota\}}^{\gamma_{\text{en}}})$$

In this definition, we introduce an additional piece of ghost state γ_{inv} which connects the invariant connective $\boxed{P}^{\iota}$ to world satisfaction. The idea is that we use the *agreement resource algebra* to synchronize the P in $\boxed{P}^{\iota}$ with the P that we work with in the definition of world satisfaction. And for every invariant $\iota \mapsto P \in I$ (*i.e.*, every currently allocated invariant), we are in one of two states: either (1) the invariant is currently closed, which means we store $\triangleright P$ and $\boxed{\{\iota\}}^{\gamma_{\text{dis}}}$ in world satisfaction, or (2) the invariant is currently open, which means we only store the token $\boxed{\{\iota\}}^{\gamma_{\text{en}}}$ in world satisfaction. In either case, we can facilitate the token exchange witnessed by InvOpen and InvClose.

Exercise 130 Derive the rules InvAlloc, InvOpen and InvClose. •

Unfortunately, this model of world satisfaction is broken!

9.2.2 Step-Indexed Types

To understand why the model of world satisfaction (from Section 9.2.1) is broken, we have to carefully look at the resource algebras that are involved. To model invariants, we use the following three resource algebras:

$$(\mathbb{P}(\mathbb{N}), \uplus) \qquad (\mathbb{P}^{\text{fin}}(\mathbb{N}), \uplus) \qquad Auth(\mathbb{N} \xrightarrow{\text{fin}} Ag(iProp))$$

The first two resource algebras are not problematic, but with the third one we have to be careful. Recall that the model of Iris propositions *iProp* is $UPred(M)$ for a specific resource algebra M. If we now refer to *iProp* in the resource algebra M, then we have constructed a cycle. Concretely, ignoring the details of multiple copies of the same ghost state for a moment, we have constructed the following cycle:

$$iProp := UPred(M) \qquad M := (\mathbb{P}(\mathbb{N}), \uplus) \times (\mathbb{P}^{\text{fin}}(\mathbb{N}), \uplus) \times Auth(\mathbb{N} \xrightarrow{\text{fin}} Ag(iProp))$$

Through M, the the type *iProp* refers to itself in its own definition. And what is even worse, this form of a cyclic definition neither has an inductive nor a coinductive solution, because $UPred(M)$ uses M in a *negative occurrence.* In fact, strongly simplified $UPred(M)$ consists of *sets of elements of* M, which has a strictly larger cardinality than M—and *iProp* cannot have a larger cardinality than itself.

So what do we do? How can we build a model of impredicative invariants nonetheless? The answer to these questions is *step-indexing.* Beyond using step-indexing in the definition of $UPred(M)$, we have to use a form of step-indexing in our types and our resources (akin to the kind of step-indexed worlds that are required for logical relations for languages with higher-order state).

Step-Indexed Types and Resources Applying step-indexing to types and resource algebras is beyond the scope of these notes. It requires generalizing our notion of types to "*(complete) ordered families of equivalences*" and generalizing our notion of resource algebras to "*cameras*". The rough idea is that equality $x = y$, validity $\overline{\mathcal{V}}$, and and various other definitions of the model all become parametric in a step-index, which is threaded through every construction. For a detailed description, we refer the reader to Jung et al. [5].

Once the generalization to step-indexed types and resources is completed, one can introduce a new type former $\blacktriangleright A$ which roughly does the same as the later modality $\triangleright P$ on propositions. And just like there are guarded fixpoints $\mu x.P$ of those propositions where recursive occurrences are guarded by a later $\triangleright P$, one can show that there are guarded fixpoints of those types where every recursive occurrence is guarded by a type-level later $\blacktriangleright A$. For Iris, the resulting equation that can be solved in the step-indexed world is (roughly):

$$iProp := UPred(M) \qquad M := (\mathbb{P}(\mathbb{N}), \uplus) \times (\mathbb{P}^{\text{fin}}(\mathbb{N}), \uplus) \times Auth(\mathbb{N} \xrightarrow{\text{fin}} Ag(\blacktriangleright iProp))$$

The price one has to pay for working with step-indexed types is that it would be *unsound* to put P into the definition of wsat without a guarding $\triangleright$. The reader may have wondered why we include $\triangleright P$ in the definition of wsat and not just P. The answer is that, if one wants to give a model of wsat in terms of step-indexed types, then the model only makes sense if P is guarded by a later. For a more extensive discussion of world satisfaction and invariants, we refer again to Jung et al. [5].

10 Concurrency

Up to this point, all the programs and languages that we have considered were *sequential*: there was only ever a single thread of execution. We will now develop techniques to reason about *concurrent* languages and programs. Before we dive into the details of concurrent languages (see Section 10.1), program logics for concurrent programs (see Section 10.2), and logical relations for concurrent languages (see Section 10.3), we start with a concrete example:

Concurrent Increment Consider the following code snippet, which *concurrently* increments a counter:

$$e_{\mathsf{count}} := \mathsf{let}\, r = \mathsf{new}(\overline{0})\, \mathsf{in}\, (\mathrm{inc}(r) \parallel \mathrm{inc}(r)) \qquad \textit{where}\, \mathrm{inc}(r) := r \leftarrow !\, r + 1.$$

The expression e_{count} creates a new reference r and then, *in parallel*, increments the value of r twice. Executing two expressions e_1 and e_2 in parallel (*i.e.*, $e_1 \parallel e_2$) here means that the execution of e_1 and e_2 are interleaved in an arbitrary order. For example, the execution could proceed by first reading r on the left side (currently 0), then r could be read on the right side (still 0), then r could be set to 1 on the right side, and, finally, r could be set to 1 (again) on the left side. In the end, following this interleaving, r will store the value 1 and *not*, as one might hope, the value 2.

The fact that concurrent programs execute in an arbitrary interleaving makes them notoriously hard to reason about, because different interleavings can result in different outcomes. For example, the state after executing e_{count} can either be $r \mapsto \overline{1}$ or $r \mapsto \overline{2}$, depending on the order in which we execute the left and right side. Since we do not know upfront which interleaving will be chosen during the execution, we have to take *all interleavings* into account when we reason about the behavior of e_{count}. This may sound easy for an example as simple as e_{count}, but if we imagine using e_{count} in a larger program, then the number of interleavings can quickly get out of hand.

In this section, we will develop modular reasoning principles for concurrent programs based on separation logic. We will see how we can reason about the interleavings of e_{count}, how to modularly compose concurrent programs, and how we can get back to more sequential reasoning by, for example, using locks around the reference r.

A note on data races For readers familiar with concurrent programming in languages such as C and Java, the expression e_{count} may seem horribly broken: *it has a data race.* Typically, one speaks of a *data race* if two threads can access the same location ℓ and at least one of them is writing to ℓ. In languages such as C and Java, data races mean the behavior of the program becomes somewhat unpredictable and, in the case of C, even undefined. In the following, we will work with *a sequentially consistent memory model*, which does not prohibit data races, since they are required to implement high-level primitives such as locks and channels. In languages such as C and Java, the corresponding constructs are known as *atomics* and can be used in the same fashion as the references in our language. The normal references in these languages, known as *non-atomics*, are not modeled in our language for the sake of simplicity (and, similarly, we also do not model weak memory).

10.1 A Concurrent Language

We extend the terms of our language by two new constructs:

$$\begin{array}{rcl} \text{Terms } e & ::= & \cdots \mid \mathsf{CAS}(e_1, e_2, e_3) \mid \mathsf{fork}\{e\} \\ \text{Evaluation Contexts } K & ::= & \cdots \mid \mathsf{CAS}(e_1, e_2, K) \mid \mathsf{CAS}(e_1, K, v_3) \mid \mathsf{CAS}(K, v_2, v_3) \end{array}$$

The operation $\mathsf{CAS}(e_1, e_2, e_3)$ is a "compare-and-set" operation. It evaluates e_1, e_2, and e_3 to a location ℓ, a value v, and a value w. Afterwards, it replaces the value at location ℓ by the value w if it is currently v. The compare-and-set operation is a synchronization primitive, which we can use to communicate between threads. We will come back to how it enables synchronization below.

The operation $\mathsf{fork}\{e\}$ forks off a new thread executing e and immediately returns. For example, after executing $\mathsf{fork}\{e_1\}\,; e_2$ the expressions e_1 and e_2 are executing in parallel. The fork operation is analogous to, for example, spawn in C-like languages. We will see how to derive the parallel connective $e_1 \parallel e_2$ from the fork primitive below.

Operational Semantics To handle concurrency, we extend the operational semantics by several new rules and a new type of reduction:

$$\frac{h(\ell) = v \qquad v \text{ comparable}}{(\mathsf{CAS}(\ell, v, w), h) \leadsto_{\mathrm{b}} (\mathsf{true}, h[\ell \mapsto w])} \qquad \frac{h(\ell) \neq v \qquad v \text{ comparable}}{(\mathsf{CAS}(\ell, v, w), h) \leadsto_{\mathrm{b}} (\mathsf{false}, h)}$$

$$\frac{(e, h) \leadsto_{\mathrm{b}} (e', h')}{(K[e], h) \leadsto (K[e'], h', [])} \qquad (K[\mathsf{fork}\{e\}], h) \leadsto (K[()], h, [e])$$

$$\frac{(e, h) \leadsto (e', h', T_f)}{(T_l \mathbin{+\!\!+} [e] \mathbin{+\!\!+} T_r, h) \leadsto_t (T_l \mathbin{+\!\!+} [e'] \mathbin{+\!\!+} T_r \mathbin{+\!\!+} T_f, h')}$$

We extend the base reduction $(e, h) \leadsto_{\mathrm{b}} (e, h')$ by two rules for compare-and-set: if the value in the heap and v are equal then we replace it and otherwise, the heap remains untouched. The compare-and-set operation returns a boolean indicating whether it was successful or not. The values that we want to compare with a compare and swap must be *comparable*, meaning they are integers, booleans, locations, unit, or simple options.

Moreover, we extend the contextual reduction $\leadsto$. Before, we just executed base reductions in an arbitrary evaluation context. Now, additionally, it becomes possible to fork off new threads with $\mathsf{fork}\{e\}$. To keep track of the forked off threads, the contextual semantics $(e, h) \leadsto (e', h', T)$ steps to a *triple* with the third component being a list of the newly forked threads.

Finally, the *thread pool reduction* $(T, h) \leadsto_t (T', h')$ reduces an arbitrary expression in the thread pool using the contextual semantics. As a consequence, the thread pool reduction enables *any interleaving* between the threads, since it does not dictate an order in which the threads have to be executed.

Synchronization and Communication From the perspective of someone who is used to programming in concurrent languages, the above extension to our sequential language may seem a bit strange: seemingly, there is no built-in way to *share data* (*e.g.*, via a lock or a channel). As it turns out, we do not need to include these (more user-friendly) communication primitives, because we can derive them from our low-level "compare-and-set" $\mathsf{CAS}(e_1, e_2, e_3)$ operation. To illustrate this point, let us implement a "spin lock":

$$\begin{aligned} \text{mklock}() &:= \mathsf{ref}(\mathsf{false}) \\ \text{lock}(l) &:= \mathsf{if}\, \mathsf{CAS}(l, \mathsf{false}, \mathsf{true})\, \mathsf{then}\, ()\, \mathsf{else}\, \text{lock}(l) \\ \text{unlock}(l) &:= l \leftarrow \mathsf{false} \end{aligned}$$

We can create a spin lock with mklock, acquire a lock with lock and release it again with unlock. Internally, the lock is just represented as a reference to a boolean such that the reference is true while someone is holding the lock and false while the lock is available. To acquire the lock, we "spin" on the location ℓ until it becomes false, so until the lock is released. To release the lock, we simply set the location ℓ to false.

For the lock implementation to be correct, we need to know that it is not possible for two threads to acquire the lock at the same time. That is where our synchronization primitive $\mathsf{CAS}(e_1, e_2, e_3)$ comes in: it checks whether the value stored at ℓ is currently false and, in the same instruction, sets it to true so other lock attempts will not succeed. If $\mathsf{CAS}(e_1, e_2, e_3)$ was not a single instruction (*i.e.*, if it would take more than one step to execute), then we could end up in a similar situation as for the expression e_{count} where two threads first read and then write a new value without coordination (or synchronization) between them.

By taking the low-level route of including primitives such as compare-and-set instead of high-level primitives such as locks, we work closer to the instructions that are offered by modern processors. It also means we can *verify* the implementation of a spin lock or a ticket lock. In fact, there exists an entire field dedicated to implementing data structures (not just locks) directly using low-level synchronization primitives (for performance reasons) known as *fine-grained concurrency.* By working directly with "compare-and-set" (or similar primitives), we retain the option to verify fine-grained concurrent data structures while, at the same time, not giving up the option to work with high-level synchronization primitives (since we can derive them).

In the implementation of the spin lock, we use a form of *"message passing"* to communicate: one thread waits for another to send him a signal in the form of updating state. We can use the same pattern to implement the parallel connective $e_1 \parallel e_2$:

$$\begin{aligned} e_1 \parallel e_2 &:= \mathsf{let}\, r = \mathsf{new}(\mathsf{false})\, \mathsf{in}\, \mathsf{fork}\{e_2\,;\, r \leftarrow \mathsf{true}\}\,;\, e_1\,;\, \text{await}(r) \\ \text{await}(x) &:= \mathsf{if}\, !x\, \mathsf{then}\, ()\, \mathsf{else}\, \text{await}(x) \end{aligned}$$

We create a reference r and fork off e_2 as a new thread. We then use r to signal that the new thread is finished and await the signal in the original thread (after executing e_1). This way, we can be sure that at the end of the execution in the first thread also the execution of e_2 is completed.

Exercise 131 An alternative to the paradigm of shared-memory and locking is message-passing via so-called *channels*:

$$\text{CHAN}(A) := \exists \alpha.\, \{\text{chan} : \mathbf{1} \to \alpha, \text{send} : A \to \alpha \to \mathbf{1}, \text{receive} : \alpha \to A\}$$

A channel can be created with the operation chan. A value of type A can be exchanged over a channel via the methods send and receive. Both send and receive wait until the other party (*i.e.*, another thread) is ready and then transfer the value in a "hand shake".

In this exercise, it is your task to implement channels. Make sure that if two threads try to receive a value at the same time, but only one thread is ready to send, then only one thread will receive the value while the other waits for another sender to arrive. Similarly, make sure that if two threads try to send a value at the same time, but only one thread is ready to receive, then only one thread will send the value while the other waits for another receiver to arrive. ●

10.2 A Concurrent Separation Logic

So now that we have concurrency in our language, how can we prove anything about concurrent programs? As it turns out, separation logic is especially well equipped to reason about concurrent programs, because we already have built-in, fine-grained control over which data is *shared* (using invariants) and which data is *exclusively owned* (using ordinary points-to assertions). To reason about concurrent programs, we extend our logic with the following rules for our new primitives:

SucCAS
$$\frac{v_1 \text{ comparable}}{\ell \mapsto v_1 * \triangleright(\ell \mapsto v_2 \twoheadrightarrow Q(\mathsf{true})) \vdash \mathsf{wp}\ \mathsf{CAS}(\ell, v_1, v_2)\ \{v.\, Q(v)\}}$$

FailCAS
$$\frac{v_1 \neq v_3 \qquad v_1 \text{ comparable}}{\ell \mapsto v_3 * \triangleright(\ell \mapsto v_3 \twoheadrightarrow Q(\mathsf{false})) \vdash \mathsf{wp}\ \mathsf{CAS}(\ell, v_1, v_2)\ \{v.\, Q(v)\}}$$

Fork
$$\mathsf{wp}\ e\ \{\_.\ \mathsf{True}\} * \triangleright Q() \vdash \mathsf{wp}\ \mathsf{fork}\{e\}\ \{v.\, Q(v)\}$$

Coq Mechanization In Coq, the CAS primitive is derived from the more general expression CmpXchg, which will not only return the result of the comparison, but also the current value (as a pair of the value and a boolean). We will always use it in the CAS form $\mathsf{CAS}(e_1, e_2, e_3) := \pi_2(\mathsf{CmpXchg}(e_1, e_2, e_3))$. The tactics to manipulate it are `wp_cmpxchg`, `wp_cmpxchg_suc`, and `wp_cmpxchg_fail`.

Invariants And that is it—well, *almost*. There is one more piece of our logic that has to change to support concurrency.[5] In a concurrent setting, the invariant opening rule InvOpen is no longer sound. In the presence of other threads, we can no longer assume the expression e in a weakest precondition $\mathsf{wp}\ e\ \{v.\, Q(v)\}$ executes from e to a value v without interruption and, hence, we can no longer keep invariants open for the entire execution. Other threads may execute at arbitrary points during the execution of e. As a consequence, we have to make sure that they cannot observe inconsistent states which violate

[5]Of course, in the model of the logic more than just the one rule changes. But in terms of the rules that we have discussed so far, only a single rule needs to change to support concurrency.

the invariant. To remain sound, we modify the invariant opening rule as follows:

$$\textsc{InvOpen}\quad \frac{P * \triangleright R \vdash \mathsf{wp}^{\mathcal{E}\setminus\mathcal{N}}\ e\ \{v.\ \triangleright R * Q(v)\} \qquad \mathcal{N} \subseteq \mathcal{E} \qquad e \text{ atomic}}{P * \boxed{R}^{\mathcal{N}} \vdash \mathsf{wp}^{\mathcal{E}}\ e\ \{v.\ Q(v)\}}$$

We only allow opening invariants around *atomic* expressions e. An expression e is atomic if it executes in a single step to a value. For example, the expressions $\mathsf{CAS}(\ell, v, w)$, $\ell \leftarrow v$, and $!\,\ell$ are all atomic. Since atomic expressions can, by definition, not be interrupted (since they are a single, atomic step), we can open invariants around them and, if needed, temporarily break them before we restore them immediately afterwards.

Example: Coin Flip Let us now see our new logic in action. For our first example, we consider a *coin flip*[6]:

$$\mathsf{flip}() = \mathsf{let}\ r = \mathsf{new}(\overline{0})\ \mathsf{in}\ \mathsf{fork}\{r \leftarrow \overline{1}\}\ ;\ !\,r$$

In a coin flip, we allocate a reference r storing 0 initially. We then fork off a new thread which *eventually* will write 1 to r. Subsequently, we dereference r, leading to two possible outcomes: if the forked-off thread has executed already, then the result will be 1, otherwise it will be 0.

Lemma 116.

$$\{\mathsf{True}\}\ \mathsf{flip}()\ \{v.\ v = \overline{0} \lor v = \overline{1}\}$$

Proof.

Context:	Goal:
	$\mathsf{wp}\ \mathsf{flip}()\ \{v.\ v = \overline{0} \lor v = \overline{1}\}$
$r \mapsto \overline{0}$	$\mathsf{fork}\{r \leftarrow \overline{1}\}\ ;\ !\,r$
We allocate the invariant:	
$\boxed{r \mapsto \overline{0} \lor r \mapsto \overline{1}}^{\mathcal{N}}$	$\mathsf{fork}\{r \leftarrow \overline{1}\}\ ;\ !\,r$
Using WpBind and Fork.	
$\boxed{r \mapsto \overline{0} \lor r \mapsto \overline{1}}^{\mathcal{N}}$	$\mathsf{wp}\ r \leftarrow \overline{1}\ \{\_.\ \mathsf{True}\} * \mathsf{wp}\ !\,r\ \{v.\ v = \overline{0} \lor v = \overline{1}\}$
First Goal	
$\boxed{r \mapsto \overline{0} \lor r \mapsto \overline{1}}^{\mathcal{N}}$	$\mathsf{wp}\ r \leftarrow \overline{1}\ \{\_.\ \mathsf{True}\}$
Using InvOpen (and timelessness).	
$r \mapsto \overline{0} \lor r \mapsto \overline{1}$	$\mathsf{wp}^{\top\setminus\mathcal{N}}\ r \leftarrow \overline{1}\ \{\_.\ r \mapsto \overline{0} \lor r \mapsto \overline{1}\}$
Follows trivially with Store.	
Second Goal	
$\boxed{r \mapsto \overline{0} \lor r \mapsto \overline{1}}^{\mathcal{N}}$	$\mathsf{wp}\ !\,r\ \{v.\ v = \overline{0} \lor v = \overline{1}\}$
Using InvOpen (and timelessness).	
$r \mapsto \overline{0} \lor r \mapsto \overline{1}$	$\mathsf{wp}^{\top\setminus\mathcal{N}}\ !\,r\ \{v.\ (v = \overline{0} \lor v = \overline{1}) * (r \mapsto \overline{0} \lor r \mapsto \overline{1})\}$
Follows trivially with Load.	

□

[6]Note that this is not a cryptographically secure technique for sampling random numbers.

Example: Lock For our next example, we return to the spin lock example:

$$\begin{aligned}\mathrm{mklock}() &:= \mathsf{ref}(\mathsf{false})\\ \mathrm{lock}(l) &:= \mathsf{if}\,\mathsf{CAS}(l, \mathsf{false}, \mathsf{true})\,\mathsf{then}\,()\,\mathsf{else}\,\mathrm{lock}(l)\\ \mathrm{unlock}(l) &:= l \leftarrow \mathsf{false}\end{aligned}$$

Typically, a lock is supposed to guard some exclusive piece of data (*e.g.*, a reference to a mutable list) which "becomes available" (*i.e.*, we may access it) upon acquiring the lock and has to be returned upon releasing the lock (*i.e.*, we may no longer access it).

In Iris, we will specify a lock with the predicate $\mathsf{lock}(\ell, P)$, which means that ℓ is a lock guarding the resource P (an Iris assertion). For this notion of a lock, we then want to show the following specification witnessing the exchange of P between lock and unlock:

$$\{P\}\,\mathrm{mklock}()\,\{v.\, \exists \ell.\, v = \ell * \mathsf{lock}(\ell, P)\} \qquad \{\mathsf{lock}(\ell, P)\}\,\mathrm{lock}(\ell)\,\{\_.\, P\}$$

$$\{\mathsf{lock}(\ell, P) * P\}\,\mathrm{unlock}(\ell)\,\{\_.\, \mathsf{True}\}$$

One can think of a lock $\mathsf{lock}(\ell, P)$ as an invariant P that is tied to program code. We open it with lock and we, subsequently, close it again with unlock. While the lock is "open", we can freely use (and break) P, but at the time when we want to "close" it again, we have to return (and restore) P. In fact, that is exactly how we define $\mathsf{lock}(\ell, P)$:

$$\mathsf{lock}(\ell, P) := \boxed{\ell \mapsto \mathsf{true} \vee (\ell \mapsto \mathsf{false} * P)}^{\mathcal{N}}$$

Given the definition of $\mathsf{lock}(\ell, P)$, the verification of the lock operations is straightforward. We show the case for lock.

Lemma 117.

$$\{\mathsf{lock}(\ell, P)\}\,\mathrm{lock}(\ell)\,\{\_.\, P\}$$

Proof.

Context: Goal:

$\mathsf{lock}(\ell, P)$ $\mathsf{wp}\ \mathrm{lock}(\ell)\ \{\_.\ P\}$

By Löb induction.

$\mathsf{lock}(\ell, P) * \triangleright \Box\, \mathsf{wp}\ \mathrm{lock}(\ell)\ \{\_.\ P\}$ $\mathsf{wp}\ \mathrm{lock}(\ell)\ \{\_.\ P\}$

Executing for one step.

$\mathsf{lock}(\ell, P) * \Box\, \mathsf{wp}\ \mathrm{lock}(\ell)\ \{\_.\ P\}$ $\mathsf{if}\ \mathsf{CAS}(\ell, \mathsf{false}, \mathsf{true})\ \mathsf{then}\ ()\ \mathsf{else}\ \mathrm{lock}(\ell)$

By binding on CAS.

$\mathsf{lock}(\ell, P) * \Box\, \mathsf{wp}\ \mathrm{lock}(\ell)\ \{\_.\ P\}$

$\mathsf{wp}\ \mathsf{CAS}(\ell, \mathsf{false}, \mathsf{true})\ \{v.\ \mathsf{wp}\ \mathsf{if}\ v\ \mathsf{then}\ ()\ \mathsf{else}\ \mathrm{lock}(\ell)\ \{\_.\ P\}\}$

By opening the invariant.

$(\ell \mapsto \mathsf{true} \lor \ell \mapsto \mathsf{false} * \triangleright P) * \Box\, \mathsf{wp}\ \mathrm{lock}(\ell)\ \{\_.\ P\}$

$\mathsf{wp}\ \mathsf{CAS}(\ell, \mathsf{false}, \mathsf{true})\ \{v.\ (\ell \mapsto \mathsf{true} \lor \ell \mapsto \mathsf{false} * P) * \mathsf{wp}\ \mathsf{if}\ v\ \mathsf{then}\ ()\ \mathsf{else}\ \mathrm{lock}(\ell)\ \{\_.\ P\}\}$

Case $\ell \mapsto \mathsf{true}$.

By FailCAS.

$\ell \mapsto \mathsf{true} * \Box\, \mathsf{wp}\ \mathrm{lock}(\ell)\ \{\_.\ P\}$

$(\ell \mapsto \mathsf{true} \lor \ell \mapsto \mathsf{false} * \triangleright P) * \mathsf{wp}\ \mathsf{if}\ \mathsf{false}\ \mathsf{then}\ ()\ \mathsf{else}\ \mathrm{lock}(\ell)\ \{\_.\ P\}$

$\Box\, \mathsf{wp}\ \mathrm{lock}(\ell)\ \{\_.\ P\}$ $\mathsf{wp}\ \mathsf{if}\ \mathsf{false}\ \mathsf{then}\ ()\ \mathsf{else}\ \mathrm{lock}(\ell)\ \{\_.\ P\}$

Which is trivial.

Case $\ell \mapsto \mathsf{false} * \triangleright P$.

By SucCAS.

$\ell \mapsto \mathsf{true} * P * \Box\, \mathsf{wp}\ \mathrm{lock}(\ell)\ \{\_.\ P\}$

$(\ell \mapsto \mathsf{true} \lor \ell \mapsto \mathsf{false} * \triangleright P) * \mathsf{wp}\ \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ ()\ \mathsf{else}\ \mathrm{lock}(\ell)\ \{\_.\ P\}$

$P * \Box\, \mathsf{wp}\ \mathrm{lock}(\ell)\ \{\_.\ P\}$ $\mathsf{wp}\ \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ ()\ \mathsf{else}\ \mathrm{lock}(\ell)\ \{\_.\ P\}$

Which is trivial.

□

Exercise 132 To simplify working with locks, we define the following syntactic sugar:

$$\mathsf{with}(l)\ \{\ e\ \} := \mathrm{lock}(l)\ ;\ \mathsf{let}\ x := e\ \mathsf{in}\ \mathrm{unlock}(l)\ ;\ x$$

Verify the following specification:

$$(P \mathrel{-\!\!*} \mathsf{wp}\ e\ \{v.\ P * Q(v)\}) * \mathsf{lock}(\ell, P) \vdash \mathsf{wp}\ \mathsf{with}(\ell)\ \{\ e\ \}\ \{v.\ Q(v)\}$$

●

Exercise 133 In the specification of the spin lock, there is nothing that prevents us from releasing a lock twice if P is not exclusive (which it typically will be). We can enforce that the lock is released only once by incorporating an exclusive token $\mathsf{locked}(\gamma) := \boxed{\mathsf{ex}()}^{\gamma}$:

$$\{P\}\ \mathrm{mklock}()\ \{v.\ \exists \ell, \gamma.\ v = \ell * \mathsf{lock}_\gamma(\ell, P)\} \qquad \{\mathsf{lock}_\gamma(\ell, P)\}\ \mathrm{lock}(\ell)\ \{\_.\ \mathsf{locked}(\gamma) * P\}$$

$$\{\mathsf{lock}_\gamma(\ell, P) * \mathsf{locked}(\gamma) * P\}\ \mathrm{unlock}(\ell)\ \{\_.\ \mathsf{True}\}$$

Define $\mathsf{lock}_\gamma(\ell, P)$ and prove the above rules. Moreover, show that:

$$\{\mathsf{locked}(\gamma) * \mathsf{lock}_\gamma(\ell, P)\}\ \mathsf{assert}\ (!\ell == \mathsf{true})\ \{\_.\ \mathsf{True}\}$$

Hint: It is possible to reuse the specification of the lock operations for this exercise, so you do not have to duplicate their proofs. •

Exercise 134 With our lock, we can define a mutex that gives mutually exclusive access to a memory cell:

$$\begin{aligned} \text{mkmutex}(x) &:= (\text{mklock}(), \mathsf{new}(x)) \\ \text{acquire}(m) &:= \text{lock}(\pi_1(m))\ ;\ (\pi_2(m), \lambda\_.\ \text{unlock}(\pi_1(m))) \end{aligned}$$

The idea here is that acquiring a mutex hands out a reference to the memory protected by it, as well as an abstraction to release it again.

Prove the following specification, for a suitably defined predicate mutex:

$$\{P(x)\}\ \text{mkmutex}(x)\ \{m.\ \text{mutex}(m, P)\}$$

$$\{\text{mutex}(m, P)\}\ \text{acquire}(m)\ \{(\ell, r).\ \ell \mapsto: P * \{\ell \mapsto: P\}\ r()\ \{\_.\ \mathsf{True}\}\}$$

where $\ell \mapsto: P := \exists v.\ \ell \mapsto v * P\ v$. •

Exercise 135 The spin lock implementation uses a form of signaling to share the ownership of P between multiple threads. A similar strategy works also for the parallel connective $e_1 \parallel e_2$. Prove the following specification for the parallel connective $e_1 \parallel e_2$:

$$\begin{array}{l} \textsc{Parallel} \\ \mathsf{wp}\ e_1\ \{\_.\ Q_1\} * \mathsf{wp}\ e_2\ \{\_.\ Q_2\} \vdash \mathsf{wp}\ e_1 \parallel e_2\ \{\_.\ Q_1 * Q_2\} \end{array}$$

Hint: You need an invariant with *three* states: an initial state, a state where e_2 has finished executing, and a final state where the main thread has acknowledged the termination of e_2. •

Example: Parallel Counter Let us now return to the parallel increment counter from the motivating example e_{count}. We are going to discuss a slightly modified version of the example, which returns the value of r in the end and which protects the accesses of r by a lock:

$$\begin{aligned} e'_{\mathsf{count}} &:= \mathsf{let}\ r = \mathsf{new}(\overline{0})\ \mathsf{in}\ \mathsf{let}\ l = \text{mklock}()\ \mathsf{in}\ (\text{inc}(l, r) \parallel \text{inc}(l, r))\ ;\ \mathsf{with}(l)\ \{\ !\,r\ \} \\ \text{inc}(l, r) &:= \mathsf{with}(l)\ \{\ r \leftarrow !\,r + 1\ \} \end{aligned}$$

In this example, we can for the first time see the true strength of using separation logic for reasoning about concurrent programs. There are infinitely many interleavings of the expression e'_{count} differing on when which thread will acquire and release the lock l (and which thread increments first). However, there are no interesting differences in these interleavings and, hence, we can summarize the behavior of e'_{count} concisely in a single specification:

$$\{\mathsf{True}\}\ e'_{\mathsf{count}}\ \{v.\ v = \overline{2}\}$$

The specification completely hides all of the implementation details of e'_{count} that are *irrelevant* for external observers (including the fact that a new thread is spawned and that references are allocated).

Lemma 118.

$$\{\mathsf{True}\}\ e'_{\mathsf{count}}\ \{v.\, v = \overline{2}\}$$

Proof Sketch. For the lock l, we use use the following resource:

$$P := \exists n_1, n_2 : \mathbb{N}.\, r \mapsto \overline{n_1 + n_2} * \gamma_1 \overset{1/2}{\hookrightarrow} n_1 * \gamma_2 \overset{1/2}{\hookrightarrow} n_2$$

Here, the ghost variable $\gamma_i \overset{q}{\hookrightarrow} n_i$ (from the resource algebra $(((0,1], +), \mathit{Ag}(\mathbb{N}))$) denote the contributions of each thread i to the value of r. For each thread i, we keep one half of the ghost variable guarded by the lock (*i.e.*, $\gamma_i \overset{1/2}{\hookrightarrow} n_i$) and give the other half to the thread i.

After joining both threads, we own $\gamma_1 \overset{1/2}{\hookrightarrow} 1$ and $\gamma_2 \overset{1/2}{\hookrightarrow} 1$ and can, hence, show that $!\,r$ will result in $\overline{2}$. □

Exercise 136 Prove $\{\mathsf{True}\}\ e'_{\mathsf{count}}\ \{v.\, v = \overline{2}\}$ by extending the proof sketch with rigorous detail. •

10.3 A Concurrent Logical Relation

The most remarkable thing about updating our logical relation from Section 7 to concurrency is that there is nothing remarkable at all. The definitions of the type interpretations $\mathcal{V}[\![A]\!]\delta$, $\mathcal{E}[\![A]\!]\delta$, and $\mathcal{G}[\![A]\!]\delta$ stay exactly the same—except for the fact that we use the concurrent weakest precondition, of course. And, without trouble, we can reprove the semantic soundness theorem:

Theorem 119 (Semantic Soundness)**.** *If* $\Delta\,;\Gamma \vdash e : A$*, then* $\Delta\,;\Gamma \vDash e : A$.

Proof. In the cases for references, we can only open the invariant for *atomic* expressions, but the expressions in our type system are atomic, so the proof carries over unchanged. We leave the new cases for our concurrency primitives as an exercise (see Exercise 137). □

Exercise 137 Prove the following compatibility lemmas for the new connectives:

$$\frac{\Delta\,;\Gamma \vDash e : \mathbf{1}}{\Delta\,;\Gamma \vDash \mathsf{fork}\{e\} : \mathbf{1}}$$

$$\frac{\Delta\,;\Gamma \vDash e_1 : \mathsf{ref}\ A \qquad \Delta\,;\Gamma \vDash e_2 : A \qquad \Delta\,;\Gamma \vDash e_3 : A \qquad A \text{ comparable}}{\Delta\,;\Gamma \vDash \mathsf{CAS}(e_1, e_2, e_3) : \mathsf{bool}}$$

Note that in the typing rule for CAS, we need to make sure that values of type A are comparable to make sure the compare-and-set does not get stuck. Comparable types are integers, booleans, unit, references (to arbitrary types), and sums of simple types. •

More interesting than the changes to the logical relation are the implications for our semantic typing proofs. Not every data type that we have discussed so far is semantically well-typed in the presence of concurrency. The reason is that for some semantic safety proofs (*e.g.*, for the Symbol ADT), we relied on opening invariants for *multiple steps of* *computation.* Since, in the presence of concurrency, we can keep invariants only open for a single step, these proofs break down. And that is for a good reason: if we think of, for example, the Symbol ADT, then concurrent executions of the mkSym function can cause the creation of symbols that are below the counter value. (We leave it as a thought exercise to think of a concrete example.)

Fortunately, even in a concurrent setting, we can recover the proofs for our data structures and verify some interesting new ones. One central aspect here is that, if we want, we can get back to sequential reasoning by using locks. That is, if a data structure is properly locked, then after acquiring the lock, we are the only ones *in the critical section* and can hence rely on sequential reasoning. So let us now turn to interesting examples of data structures that we can verify in the presence of concurrency.

Example: Locked Data Structure We start with a simple, locked data structure: *a redundant counter*. In a redundant counter, we have two references which both store the value of the counter:

$$\begin{aligned} \text{COUNTER} &:= \exists \alpha.\, \{\text{counter} : \mathbf{1} \to \alpha, \text{get} : \alpha \to \mathsf{int}, \text{inc} : \alpha \to \mathbf{1}\} \\ \text{Counter} &:= \mathsf{pack}\{\quad \text{counter}() := (\mathsf{new}(\overline{0}), \mathsf{new}(\overline{0}), \text{mklock}()), \\ &\qquad \text{get}(c_1, c_2, l) := \mathsf{with}(l)\,\{\, \mathsf{assert}\,(!\,c_1 == !\,c_2)\,;\, !\,c_1 \,\} \\ &\qquad \text{inc}(c_1, c_2, l) := \mathsf{with}(l)\,\{\, c_1 \leftarrow !\,c_2 + 1\,;\, c_2 \leftarrow !\,c_1 \,\}\} \end{aligned}$$

Lemma 120.

$$\emptyset\,;\emptyset \vDash \text{Counter} : \text{COUNTER}$$

Proof Sketch. We allocate the two counter references and store them in the lock as:

$$P := \exists n : \mathbb{N}.\, c_1 \mapsto \overline{n} * c_2 \mapsto \overline{n}$$

The rest is straightforward using the specification of the lock. □

Exercise 138 Modify the Symbol ADT to use locks around its operations. Prove that, with locks, it is again semantically well-typed. •

Example: Channels For our next example, we look at channels again—this time with an implementation and a type system:

$$\begin{aligned} \text{CHAN}(A) &:= \exists \alpha.\, \{\text{chan} : \mathbf{1} \to \alpha, \text{send} : A \times \alpha \to \mathbf{1}, \text{receive} : \alpha \to A\} \\ \text{Chan} &:= \mathsf{pack}\{\text{chan} = \text{chan}, \text{send} = \text{send}, \text{receive} = \text{receive}\} \end{aligned}$$

where the operations on channels as follows:

$$
\begin{aligned}
\mathrm{chan}() &:= (\mathrm{mklock}(), \mathrm{mklock}(), \mathsf{new}(\mathsf{None})),\\
\mathrm{send}((s,r,c),a) &:= \mathsf{with}(s)\left\{\begin{array}{l}\mathsf{assert}\ (!c = \mathsf{None});\\ c \leftarrow \mathsf{Some}(a);\\ \mathsf{await}\{!c = \mathsf{None}\}\end{array}\right\}\\
\mathrm{receive}(s,r,c) &:= \mathsf{with}(r)\left\{\begin{array}{l}\mathsf{await}\{!c \neq \mathsf{None}\};\\ \mathsf{let}\ a = \mathrm{unwrap}(!c)\ \mathsf{in}\\ c \leftarrow \mathsf{None};\\ a\end{array}\right\}\\
\mathrm{unwrap}(o) &:= \mathsf{match}\ o\ \mathsf{with}\ \mathsf{None} \Rightarrow \mathsf{assert}\ (\mathsf{false}) \mid \mathsf{Some}(k) \Rightarrow k\ \mathsf{end}\\
\mathsf{await}\{e\} &:= \mathrm{await}(\lambda\_.\ e)\\
\mathsf{await}(f) &:= \mathsf{if}\ f()\ \mathsf{then}\ ()\ \mathsf{else}\ \mathsf{await}(f)
\end{aligned}
$$

A channel consists of two locks s and r and a reference c: the lock s is for the sender, the lock r for the receiver, and the reference c is for the exchange of the value. The sender will acquire the sender lock, send the value over the channel (by writing to c), and then wait to a receiver to take it our of the channel before returning. The receiver will acquire the receiver lock, wait for a value to appear on the channel (by reading c), and then take it out of the channel (signaling the sender the exchange is finished) and return the value.

Lemma 121.

$\emptyset\,;\emptyset \vDash \mathrm{Chan} : \mathrm{CHAN}$

As usual, our goal is to prove that our channel implementation is semantically well-typed (in Lemma 121). The proof is somewhat involved, yet it does not require any new machinery. In the following, we will sketch the high-level idea of the proof and leave completing the proof as an exercise (see Exercise 139). Let us begin by collecting a number of observations about the channel implementation and their consequences for our proof:

a) The sender and the receiver use *different locks* while working on the same underlying data. As a consequence, a sender can exclude other senders from operating concurrently on the channel data, but it will not—and should not—exclude a concurrent receiver from operating on the data. In fact, it would be plain wrong to exclude the receiver, since the sender cannot make *progress* without a corresponding receiver.

   As a consequence, we cannot store the ownership of c in either the sender or the receiver lock—the other party will always need access to c even without holding both locks. We will, thus, set up an *additional invariant* to control ownership of c (despite the locks r and s that can be used to share ownership).

b) The reference c serves two purposes: First, it carries over the data from the sending thread to the receiving thread. Second, it functions as a *two-way signal*: the sender uses the reference c to signal the receiver that there is a new value (which the receiver eagerly awaits) and the receiver uses the reference c to signal the sender that it has received the new value (which the sender eagerly awaits). It is instrumental that there is a clear order who may mutate c at any given point to ensure that the sender and the receiver do not step on each others feet.

Implementing the mere data exchange from sender to receiver is simple: we set up an invariant which contains ownership of c such that c is either None, or c is $\mathsf{Some}(a)$ such that a is semantically of type A (written $a \in \tau_A$ below). The difficult bit are the transitions between the two states. For example, consider how the value of c changes from the perspective of the receiver: Initially, c might store None and the receiver starts spinning. Eventually, c will change to $\mathsf{Some}(a)$ and the receiver exists the loop. But now the receiver reads c again and expects it to be *unchanged*, meaning it must still point to $\mathsf{Some}(a)$—no other thread may have interfered in the meantime. Finally, the receiver *mutates* c by updating it None, meaning c will not always stay $\mathsf{Some}(a)$, which rules out monotonicity arguments such as the one-shot example.

Ultimately, we want to enforce the transition system:

$$\begin{tikzcd} \boxed{c \mapsto \mathsf{None}} \arrow[r, bend right, "\textit{sender}"'] & \boxed{c \mapsto \mathsf{Some}(a)} \arrow[l, bend right, "\textit{receiver}"'] \end{tikzcd}$$

where only the sender can go from None to $\mathsf{Some}(a)$ and only the receiver can go from $\mathsf{Some}(a)$ to None. Moreover, the sender always starts and ends in the state $c \mapsto \mathsf{None}$, while the receiver can start (and end) in any of the two states.

To encode the state transition system with restrictions on the possible transitions, we use ghost state. Concretely, we use four tokens ${\color{purple}\bullet_s}$, ${\color{blue}\bullet_s}$, ${\color{purple}\bullet_r}$, and ${\color{blue}\bullet_r}$, each a separate instance of the $\mathit{Ex}(\mathbf{1})$ algebra. Each token is exclusive (meaning, for example, ${\color{purple}\bullet_s} * {\color{purple}\bullet_s} \vdash \mathsf{False}$) and we can easily allocate them in our proof. We distribute the tokens as follows:

$$\begin{aligned} P_s &:= {\color{purple}\bullet_s} \\ P_r &:= {\color{purple}\bullet_r} \\ I_c &:= \quad ({\color{blue}\bullet_s} * {\color{blue}\bullet_r} * c \mapsto \mathsf{None}) \\ &\qquad \vee ({\color{purple}\bullet_s} * {\color{blue}\bullet_r} * \exists a.\, c \mapsto \mathsf{Some}(a) * a \in \tau_A) \\ &\qquad \vee ({\color{blue}\bullet_s} * {\color{purple}\bullet_r} * \exists a.\, c \mapsto \mathsf{Some}(a) * a \in \tau_A) \\ &\qquad \vee ({\color{purple}\bullet_s} * {\color{blue}\bullet_r} * c \mapsto \mathsf{None}) \end{aligned}$$

The proposition P_s is the resource that is guarded by the sender lock—the sender token ${\color{purple}\bullet_s}$. The proposition P_r is the resource that is guarded by the receiver lock—the receiver token ${\color{purple}\bullet_r}$. The invariant I_c is the invariant that we allocate in the proof. It has four separate state and five transitions:

1. Initially, we are in the first state ${\color{blue}\bullet_s} * {\color{blue}\bullet_r} * c \mapsto \mathsf{None}$, where the receiver token ${\color{blue}\bullet_r}$, the sender token ${\color{blue}\bullet_s}$, and the ownership of c (which initially points to None) are stored in the invariant.

2. From the initial state, the sender can use its token (guarded by the lock) ${\color{purple}\bullet_s}$ to transition to the second state: ${\color{purple}\bullet_s} * {\color{blue}\bullet_r} * \exists a.\, c \mapsto \mathsf{Some}(a) * a \in \tau_A$. Note that this transition is impossible for the receiver, since it does not own the token ${\color{purple}\bullet_s}$, which is required to

transition to a $\mathsf{Some}(a)$ state. In fact, while the sender owns $\bullet_s$, the value of c must always be None, since the initial state is the only compatible state.

3. After the reference has been updated by the source, the receiver can transition to the next state: $\bullet_s * \bullet_r * \exists a.\, c \mapsto \mathsf{Some}(a) * a \in \tau_A$. This transition is impossible for the sender, since it does not have the token $\bullet_r$ required for this transition. Moreover, by taking this transition, the receiver can ensure that no-one will move from $\mathsf{Some}(a)$ to None while it closes the invariant again—until $\bullet_r$ is replaced by $\bullet_r$, the sender cannot do any state transitions.

4. Subsequently, when the receiver sets the value of c to None, we transition to the fourth state: $\bullet_s * \bullet_r * c \mapsto \mathsf{None}$ and, in the process, we swap $\bullet_r$ for $\bullet_r$. Since the receiver gets out $\bullet_r$, it can close its own lock successfully.

5. Finally, the sender is enabled again and it can, after waiting for None to appear in c, replace $\bullet_s$ with $\bullet_s$ in the invariant again, returning to the initial state $\bullet_s * \bullet_r * c \mapsto \mathsf{None}$. By taking out $\bullet_s$, the sender gets back the token that is required for its own lock.

Exercise 139 Prove Lemma 121. •

References

[1] A. Ahmed. *Semantics of types for mutable state.* PhD thesis, Princeton University, 2004.

[2] A. W. Appel and D. A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *TOPLAS*, 2001.

[3] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, 2nd revised edition, 1984.

[4] C. A. R. Hoare. An axiomatic basis for computer programming. *CACM*, 1969.

[5] R. Jung, R. Krebbers, J. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *JFP*, 2018.

[6] R. Krebbers, A. Timany, and L. Birkedal. Interactive proofs in higher-order concurrent separation logic. In *POPL*, 2017.

[7] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP*, 1983.

[8] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[9] S. Schäfer, T. Tebbi, and G. Smolka. Autosubst: Reasoning with de Bruijn terms and parallel substitutions. In *Interactive Theorem Proving*, 2015.

[10] P. Wadler. Theorems for free! In *FPCA*, 1989.