Semantics of Type Systems
Lecture Notes

Derek Dreyer MPI-SWS
Ralf Jung MPI-SWS
Jan-Oliver Kaiser MPI-SWS

February 18, 2016

Contents

1 Simply Typed Lambda Calculus

$$
\begin{array}{rrcl}
\text{Variables} & x, y & & \ldots \\
\text{Types} & A, B & ::= & a \mid A \to B \\
\text{Base Types} & a, b & ::= & \mathsf{int} \\
\text{Variable Contexts} & \Gamma & ::= & \emptyset \mid \Gamma, x : A \\
\text{Source Terms} & E & ::= & x \mid \lambda x : A.\ E \mid E_1\ E_2 \mid E_1 + E_2 \mid \overline{n} \\
\text{Runtime Terms} & e & ::= & x \mid \lambda x.\ e \mid e_1\ e_2 \mid e_1 + e_2 \mid \overline{n} \\
\text{(Runtime) Values} & v & ::= & \lambda x.\ e \mid \overline{n}
\end{array}
$$

Variables and Substitution We use Barendregt's variable convention, which means we assume that all bound variables are distinct and maintain this invariant implicitly. Another way of saying this is: we will not worry about the formal details of variable names, alpha renaming, freshness, etc., and instead just assume that all variables bound in a variable context are distinct and that we keep it that way when we add a new variable to the context. Of course, getting such details right is very important when we mechanize our reasoning, but in this part of the course, we will not be using Coq, so we can avoid worrying about it.

This may all sound very sketchy, and indeed it is if you do "funky" things with your variable contexts, like having inference rules that merge variables together. Derek did something funky in his first paper, and this led to a major flaw—the "weakening" lemma did not hold—and the paper had to be retracted (fortunately before it was actually published)! But in practice people are sloppy about variable binding all the time when doing pencil-and-paper proofs, and it is rarely a source of errors so long as you don't get funky.

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

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

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

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

1.1 Type Safety

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

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

$$\textsc{var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{lam}\ \frac{\Gamma, x : A \vdash E : B}{\Gamma \vdash \lambda x : A.\ E : A \to B} \qquad \textsc{app}\ \frac{\Gamma \vdash E_1 : A \to B \qquad \Gamma \vdash E_2 : A}{\Gamma \vdash E_1\ E_2 : B}$$

$$\textsc{plus}\ \frac{\Gamma \vdash E_1 : \mathsf{int} \qquad \Gamma \vdash E_2 : \mathsf{int}}{\Gamma \vdash E_1 + E_2 : \mathsf{int}} \qquad \textsc{int}\ \Gamma \vdash \overline{n} : \mathsf{int}$$

Curry-style typing

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

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

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

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

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 terms.

Type Erasure

$\boxed{\text{Erase}(\cdot)}$

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

Lemma 1 (Erasure)**.** *If* $\vdash E : A$*, then* $\vdash \mathit{Erase}(E) : A$*.*

Lemma 2 (Weakening)**.** *If* $\Gamma \vdash e : A$ *then* $\Gamma, x : B \vdash e : A$*.*

Proof. By induction on $\Gamma \vdash e : A$:

Case 1: VAR, $e = y$. From $y \in \Gamma$ we get $y \in \Gamma, x : B$.

Case 2: LAM, $e = \lambda y.\, e'$ and $A = A_1 \to A_2$. It suffices to show $\Gamma, x : B, y : A_1 \vdash e : A_2$. By induction, we have $\Gamma, y : A_1, x : B \vdash e : A_2$. By re-ordering, this suffices.

Case 3: APP, $e = e_1\ e_2$. It suffices to show $\Gamma, x : B \vdash e_1$ and $\Gamma, x : B \vdash e_2$, which are exactly our inductive hypotheses.

Case 4: PLUS, $e = e_1 + e_2$. (Just like APP).

Case 5: INT, $e = \overline{n}$, $A = \mathsf{int}$. By INT we have $\Gamma, x : B \vdash e : \mathsf{int}$. □

Lemma 3 (Preservation of Typing under Substitution)**.**
If $\Gamma, x : A \vdash e : B$ *and* $\Gamma \vdash v : A$*, then* $\Gamma \vdash e[v/x] : B$*.*

Proof. By induction on $\Gamma, x : A \vdash e : B$:

Case 1: VAR, $e = x$ and $B = A$. We have $x[v/x] = v$. By assumption, $\Gamma \vdash v : A$.

Case 2: VAR, $e = y \neq x$. We have $y \in \Gamma$ and $y[v/x] = y$. It suffices to show $\Gamma \vdash y : B$, which we have by VAR.

Case 3: LAM, $e = \lambda y.\, e'$ and $B = B_1 \to B_2$. Note that $(\lambda y.\, e')[v/x] = \lambda y.\, e'[v/x]$. Thus, it suffices to show $\Gamma, y : B_1 \vdash e'[v/x] : B_2$. This holds by induction.

Case 4: APP, $e = e_1\ e_2$. By induction.

Case 5: APP, $e = e_1 + e_2$. By induction.

Case 6: INT, $e = \overline{n}$ and $A = \mathsf{int}$. By INT we have $\Gamma \vdash \overline{n} : \mathsf{int}$. □

Definition 4 (Kosher Terms)**.** *A (runtime) term e is* kosher *if either it is a value or there exists e' s.t. $e \succ e'$.*

Theorem 5 (Progress)**.** *If $\vdash e : A$, then e is kosher.*

Theorem 6 (Preservation)**.** *If $\vdash e : A$ and $e \succ e'$, then $\vdash e' : A$.*

Corollary 7 (Type Safety)**.** *If $\vdash e : A$ and $e \succ^\star e'$, then e' is kosher.*

Exercise 1 (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 syntax of types:

$$
\begin{array}{rrcl}
\text{Types} & A, B & ::= & \ldots \mid A \times B \mid A + B \\
\text{Source Terms} & E & ::= & \ldots \mid \langle E_1, E_2 \rangle \mid \pi_1\ E \mid \pi_2\ E \\
& & & \mid \mathsf{inj}_1^{A+B}\ E \mid \mathsf{inj}_2^{A+B}\ E \\
& & & \mid (\mathsf{case}\ E_0\ \mathsf{of}\ \mathsf{inj}_1\ x_1.\ E_1 \mid \mathsf{inj}_2\ x_2.\ E_2\ \mathsf{end}) \\
\text{Runtime Terms} & e & ::= & \ldots \mid \langle e_1, e_2 \rangle \mid \pi_1\ e \mid \pi_2\ e \\
& & & \mid \mathsf{inj}_1\ e \mid \mathsf{inj}_2\ e \mid (\mathsf{case}\ e_0\ \mathsf{of}\ \mathsf{inj}_1\ x_1.\ e_1 \mid \mathsf{inj}_2\ x_2.\ e_2\ \mathsf{end}) \\
\text{(Runtime) Values} & v & ::= & \ldots \mid \langle v_1, v_2 \rangle \mid \mathsf{inj}_1\ v \mid \mathsf{inj}_2\ v
\end{array}
$$

The operational behavior of products and sums is defined next:

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

$$
\frac{e \succ e'}{\mathsf{inj}_i\ e \succ \mathsf{inj}_i\ e'}\ \text{INJ-STRUCT}
$$

$$
\frac{e_0 \succ e_0'}{\mathsf{case}\ e_0\ \mathsf{of}\ \mathsf{inj}_1\ x_1.\ e_1 \mid \mathsf{inj}_2\ x_2.\ e_2\ \mathsf{end} \succ \mathsf{case}\ e_0'\ \mathsf{of}\ \mathsf{inj}_1\ x_1.\ e_1 \mid \mathsf{inj}_2\ x_2.\ e_2\ \mathsf{end}}\ \text{CASE-STRUCT}
$$

$$
\mathsf{case}\ \mathsf{inj}_i\ v\ \mathsf{of}\ \mathsf{inj}_1\ x_1.\ e_1 \mid \mathsf{inj}_2\ x_2.\ e_2\ \mathsf{end} \succ e_i[v/x_i]\ \text{CASE-INJ}
$$

The new typing rules correspond to the introduction and elimination rules of conjunction and disjunction from natural deduction:

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

$$
\frac{\Gamma \vdash E_0 : B + C \qquad \Gamma, x_1 : B \vdash E_1 : A \qquad \Gamma, x_2 : C \vdash E_2 : A}{\Gamma \vdash \mathsf{case}\ E_0\ \mathsf{of}\ \mathsf{inj}_1\ x_1.\ E_1 \mid \mathsf{inj}_2\ x_2.\ E_2\ \mathsf{end} : A}\ \text{CASE}
$$

This completes the definitions you need for the following exercises:

a) Define the typing rules on untyped runtime terms.

b) Extend the proof of progress to the language with products and sums.

•

1.2 Contextual Operational Semantics

Evaluation Contexts $\quad K ::= \bullet \mid K\ e \mid v\ K \mid K + e \mid v + K$

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

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

Contextual operational semantics $\boxed{e_1 \rightsquigarrow e_2}$

CTX
$$\frac{e_1 \rightsquigarrow e_2}{K[e_1] \rightsquigarrow K[e_2]}$$

BETA
$$(\lambda x.\ e)\ v \rightsquigarrow e[v/x]$$

PLUS
$$\overline{n} + \overline{m} \rightsquigarrow \overline{n + m}$$

Exercise 2 Prove the following two statements:

a) If $e \succ e'$ then $e \rightsquigarrow e'$.

b) If $e \rightsquigarrow e'$ then $e \succ e'$.

•

Definition 8 (Kosher Terms, contextual)**.** *A (runtime) term e is* kosher *if either it is a value or there exists e' s.t. $e \rightsquigarrow e'$.*

Theorem 9 (Progress)**.** *If $\vdash e : A$, then e is kosher.*

Proof. By induction on $\vdash e : A$.

Case 1: $e = x$. By inversion on $\vdash e : A$ we have $x : A \in \emptyset$.

Case 2: $e = \lambda x.\ e'$. $\lambda x.\ e'$ is a value.

Case 3: $e = v_1\ v_2$. By inversion we have $\vdash v_1 : B \to A$ and $\vdash v_2 : B$. Thus, $v_1 = \lambda x.\ e_1$ for some x and e_1. By BETA we have $e \rightsquigarrow e_1[v_2/x]$.

Case 4: $e = v\ e_2$ where e_2 is not a value. By inversion we have $\vdash v : B \to A$ and $\vdash e_2 : B$. By induction, there exists e_2' s.t. $e_2 \rightsquigarrow e_2'$. We have $e = (v\ \bullet)[e_2]$. Thus, by CTX, we have $e \rightsquigarrow v\ e_2'$

Case 5: $e = e_1\ e_2$ where e_1 is not a value. By inversion we have $\vdash e_1 : B \to A$ and $\vdash e_2 : B$. By induction, there exists e_1' s.t. $e_1 \rightsquigarrow e_1'$. We have $e = (\bullet\ e_2)[e_1]$. Thus, by CTX, we have $e \rightsquigarrow e_1'\ e_2$.

Case 6: $e = \overline{n}$. $\overline{n}$ is a value. □

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

$$\text{HOLE}\ \vdash \bullet : A \Rightarrow A \qquad \frac{\vdash K : A \Rightarrow (C \to B) \qquad \vdash e : C}{\vdash K\ e : A \Rightarrow B}\ \text{APP-L} \qquad \frac{\vdash v : C \to B \qquad \vdash K : A \Rightarrow C}{\vdash v\ K : A \Rightarrow B}\ \text{APP-R}$$

$$\frac{\vdash K : A \Rightarrow \mathsf{int} \qquad \vdash e : \mathsf{int}}{\vdash K + e : A \Rightarrow \mathsf{int}}\ \text{PLUS-L} \qquad \frac{\vdash v : \mathsf{int} \qquad \vdash K : A \Rightarrow \mathsf{int}}{\vdash x + K : A \Rightarrow \mathsf{int}}\ \text{PLUS-R}$$

Definition 10 (Kosher Terms, contextual)**.** *A (runtime) term e is* kosher *if either it is a value or there exists e' s.t. $e \rightsquigarrow e'$.*

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

$\vdash K : B \Rightarrow A$ *and* $\vdash e : B$.

Proof. By induction on K.

Case 1: $K = \bullet$. We chose $B = A$.

Case 2: $K = K'\ e'$. By inversion of $\vdash K[e] : A$, we have $\vdash K'[e] : C \to A$ and $\vdash e' : C$. By induction, there exists B' s.t. $\vdash K' : B' \Rightarrow (C \to A)$ and $\vdash e : B'$. We choose $B = B'$. It remains to show that $\vdash K'\ e' : B' \Rightarrow A$. By APP-L, it suffices to show that $\vdash K' : B' \Rightarrow (C \to A)$ and $e' : C$.

Case 3: $K = v\ K'$. By inversion of $\vdash K[e] : A$, we have $\vdash K'[e] : C$ and $\vdash v : C \to A$. By induction, there exists B' s.t. $\vdash K' : B' \Rightarrow C$ and $\vdash e : B'$. We choose $B = B'$. It remains to show that $\vdash v\ K' : B' \Rightarrow A$. By APP-R, it suffices to show that $\vdash v : C \to A$ and $\vdash K' : B' \Rightarrow C$.

Case 4: $K = K' + e'$. By inversion of $\vdash K[e] : A$, we have $A = \mathsf{int}$, $\vdash K'[e] : \mathsf{int}$ and $\vdash e' : \mathsf{int}$. By induction, there exists B' s.t. $\vdash K' : B' \Rightarrow \mathsf{int}$ and $\vdash e : B'$. We choose $B = B'$. It remains to show that $\vdash K'[e] + e' : B' \Rightarrow \mathsf{int}$. By PLUS-L, it suffices to show that $\vdash K' : B' \Rightarrow \mathsf{int}$ and $\vdash e : \mathsf{int}$.

Case 5: $K = e' + K'$. Analogous to the previous case ☐

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

Proof. By straight-forward induction on $\vdash K : B \Rightarrow A$. (Details omitted.) ☐

Theorem 13 (Preservation)**.** *If $\vdash e : A$ and $e \rightsquigarrow e'$, then $\vdash e' : A$.*

Proof. By induction on $e \rightsquigarrow e'$.

Case 1: BETA. We have $e = (\lambda x.\ e_1)\ v$ 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 $\vdash v : A_0$. By inversion on $\vdash \lambda x.\ e_1 : A_0 \to A$ we have $x : A_0 \vdash e_1 : A$. By Lemma 3, we have $\vdash e_1[v/x] : A$.

Case 2: PLUS. We have $e = \overline{n} + \overline{m}$ and $e' = \overline{n+m}$. We establish $\vdash \overline{n+m} : \mathsf{int}$ by INT.

Case 3: CTX. We have K, e_1 and e_1' s.t. $e = K[e_1]$, $e_1 \rightsquigarrow e_1'$, and $e' = K[e_1']$. It suffices to show that $\vdash K[e_1'] : A$. By Lemma 11, there exists B s.t. $\vdash K : B \Rightarrow A$ and $\vdash e_1 : B$. By induction, we have $\vdash e_1' : B$. By Lemma 12, we have $\vdash K[e_1'] : A$. ☐

Corollary 14 (Type Safety). *If* $\vdash e : A$ *and* $e \rightsquigarrow^\star e'$*, then* e' *is kosher.*

Exercise 3 Re-do exercise 1 for the contextual operational semantics:

a) Extend the definition of evaluation contexts for products and sums.

b) Extend the definition of filling.

c) Extend the definition of the contextual semantics $\rightsquigarrow$.

d) Update the proofs of progress and preservation.

•

1.3 Termination

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

Statement 15 (Termination). *If* $\vdash e : A$*, then there exists* v *s.t.* $e \rightsquigarrow^* v$.

It will be convenient to use so-called big-step semantics. Big-step semantics relate a runtime expression to a value if and only if the expression evaluates to that value.

Big-Step Semantics

$\boxed{e \downarrow v}$

$$\frac{}{\overline{n} \downarrow \overline{n}}\ \textsc{literal} \qquad \frac{}{\lambda x.\, e \downarrow \lambda x.\, e}\ \textsc{lambda}$$

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

Exercise 4 Extend the big-step semantics to handle products and sums, as defined previously. •

Exercise 5 Prove that $e \rightsquigarrow^\star v \iff e \downarrow v$ (including products and sums).
Note that the lambda case of $e \downarrow v \Rightarrow e \rightsquigarrow^\star v$ was already done in class. •

Corollary 16. *If* $e \downarrow v$*, then* $e \rightsquigarrow^\star v$.

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

Value Relation

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

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

Expression Relation

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

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

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

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

We define the action of a substitution γ on and expression e.

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

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

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 $e \in \mathcal{E}[\![A]\!]$.*

Lemma 18 (Closure under Expansion)**.** *If $e \rightsquigarrow^\star e'$ and $e' \in \mathcal{E}[\![A]\!]$, then $e \in \mathcal{E}[\![A]\!]$.*

Theorem 19 (Semantic Soundness)**.** *If $\Gamma \vdash e : A$, then $\Gamma \vDash e : A$.*

Proof. By induction $\Gamma \vdash e : A$.

Case 1: $e = x$.
We have $e : A \in \Gamma$.
Given $\gamma \in \mathcal{G}[\![\Gamma]\!]$, we have to show $\gamma(x) \in \mathcal{E}[\![A]\!]$.
Thus, it suffices to show $\gamma(x) \in \mathcal{E}[\![A]\!]$.
From $\gamma \in \mathcal{G}[\![\Gamma]\!]$ and $x : A \in \Gamma$ we have $\gamma(x) \in \mathcal{V}[\![A]\!]$.
By Lemma 17, $\gamma(x) \in \mathcal{E}[\![A]\!]$.

Case 2: $e = \lambda x.\ e'$ and $A = B \to C$.
We have $\Gamma, x : B \vdash e' : C$. Given $\gamma \in \mathcal{G}[\![\Gamma]\!]$, we have to show $\gamma(\lambda x.\ e') \in \mathcal{E}[\![B \to C]\!]$.
Thus, by definition of γ, it suffices to show that $\lambda x.\ \gamma(e') \in \mathcal{E}[\![B \to C]\!]$.
By Lemma 17, it suffices to show that $\lambda x.\ \gamma(e') \in \mathcal{V}[\![B \to C]\!]$.
We are given $v \in \mathcal{V}[\![B]\!]$ and have to show that $\gamma(e')[v/x] \in \mathcal{E}[\![C]\!]$.
By induction, we have $\forall \gamma' \in \mathcal{G}[\![\Gamma, x : B]\!].\ \gamma'(e') \in \mathcal{E}[\![C]\!]$.
We choose $\gamma' := \gamma[x \mapsto v]$.
Thus, $\gamma'(e') = \gamma(e')[v/x] \in \mathcal{E}[\![C]\!]$.

Case 3: $e = e_1\ e_2$.
We have $\Gamma \vdash e_1 : B \to C$ and $\Gamma \vdash e_2 : B$.
Given $\gamma \in \mathcal{G}[\![\Gamma]\!]$, we have to show $\gamma(e_1\ e_2) \in \mathcal{E}[\![C]\!]$.
Thus, it suffices to show that $\gamma(e_1)\ \gamma(e_2) \in \mathcal{E}[\![C]\!]$.
By induction, $\gamma(e_1) \in \mathcal{E}[\![B \to C]\!]$ and $\gamma(e_2) \in \mathcal{E}[\![B]\!]$.

Thus, there exist e', v_2 s.t. $\gamma(e_1) \downarrow \lambda x.\, e' \in \mathcal{V}[\![B \to C]\!]$ and $\gamma(e_2) \downarrow v_2 \in \mathcal{V}[\![B]\!]$.
From $\lambda x.\, e' \in \mathcal{V}[\![B \to C]\!]$ and $v_2 \in \mathcal{V}[\![B]\!]$, we have $e'[v_2/x] \in \mathcal{E}[\![C]\!]$.
Thus, there exists v s.t. $e'[v_2/x] \downarrow v \in \mathcal{V}[\![C]\!]$ and, thus, $v \in \mathcal{E}[\![C]\!]$.
By Lemma 18, it suffices to show that $\gamma(e_1)\ \gamma(e_2) \rightsquigarrow^\star v$.
With Corollary 16, it remains to show that $\gamma(e_1)\ \gamma(e_2) \downarrow v$.
By APP, we are done.

Case 4: $e = e_1 + e_2$. (Omitted.) □

Corollary 20 (Termination)**.** *If* $\bullet \vdash e : A$*, then there exists* v *s.t.* $e \downarrow v$*.*

Proof. By Theorem 19, we have $\bullet \vDash e : A$. Pick γ to be the identity, which clearly is in $\mathcal{G}[\![\bullet]\!]$. Hence $e \in \mathcal{E}[\![A]\!]$. By definition then, $\exists v.\, e \downarrow v$. □

Exercise 6 Extend the termination proof to cover products and sums. •

2 System F: Polymorphism and Existential Types

We extend our language with polymorphism and existential types.

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

Contextual operational semantics $\boxed{e_1 \rightsquigarrow e_2}$

$$
\begin{array}{ll}
\textsc{bigBeta} & \textsc{unpack} \\
(\Lambda.\, e)\ \langle\rangle \rightsquigarrow e & \mathsf{unpack}\ (\mathsf{pack}\ v)\ \mathsf{as}\ x\ \mathsf{in}\ e \rightsquigarrow e[v/x]
\end{array}
$$

2.1 Type Safety

This section defines the typing rules for the new terms, and proves that System F (STLC with universal types) enjoys type safety. In the exercises, you will extend that proof to cover existential types. Because we now deal with type variables, we have to deal with a new kind of contexts: Type variable contexts. Well-typedness is now relative to both a type variable 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 $\boxed{\Delta\,;\Gamma \vdash E : A}$

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

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

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

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

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

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

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

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

Lemma 21 (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 would have to update the composition and decomposition lemmas to handle the new evaluation context. However, we will omit this trivial proof.

Theorem 22 (Preservation)**.** *Theorem 13 remains valid with universal types: If* $\vdash e : A$ *and* $e \rightsquigarrow e'$*, then* $\vdash e' : A$*.*

Proof. Remember we are doing induction on $e \rightsquigarrow e'$.

Case 1: bigBeta.
We have $e = (\Lambda.\, e_1)\ \langle\rangle$ and $e' = e_1$.
By inversion on $\vdash e : A$, we have $\vdash \Lambda.\, e_1 : \forall \alpha.\, B$ such that $A = B[C/\alpha]$, $\Delta \vdash C$.
By another inversion step, we obtain $\alpha\,;\emptyset \vdash e_1 : B$.
By type substitution, we have $\vdash e_1 : B[C/\alpha]$, so we are done.

□

Theorem 23 (Progress)**.** *Theorem 9 remains valid with universal types: If* $\vdash e : A$*, then* e *is kosher.*

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

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

Case 2: $e = v\ \langle\rangle$.
By inversion, we have that $\vdash v : \forall \alpha.\, B$.
Thus, $v = \Lambda.\, e'$, and $e \rightsquigarrow e'$.

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

□

Exercise 7 Extend the proofs of progress and preservation to handle existential types. •

Exercise 8 (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 9 (Existential Fun) In your first semester at UdS, when you learned SML, you saw 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 $\mathsf{rec}\ f\ (x : A) : B.\ e$. The typing rule is

$$\frac{\Gamma, f : A \to B, x : A \vdash E : B}{\Gamma \vdash \mathsf{rec}\ f\ (x : A) : B.\ e : A \to B}\ \text{REC}$$

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

$$\mathsf{rec}\ \mathit{fib}\ (x : \mathsf{int}) : \mathsf{int}.\ \mathsf{if}\ x \leq 1\ \mathsf{then}\ x\ \mathsf{else}\ \mathit{fib}\ ((x - \overline{2})) + \mathit{fib}\ (x - \overline{1})$$

This term has type $\mathsf{int} \to \mathsf{int}$.

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

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

b) Define an instance 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 adds an equality test to the interface. Define a function of type $A_{\mathsf{ISET}} \to A_{\mathsf{ISETE}}$ that implements the equality test for an arbitrary implementation of A_{ISET}.

•

2.2 Termination

We extend the semantic model to handle universal and existential types. The naïve approach, i.e. quantifying over arbitrary syntatic 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 into our model.

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

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

Semantic Type Relation $\boxed{\delta \in \mathcal{D}[\![\Delta]\!]}$

$$\mathcal{D}[\![\Delta]\!] = \{\delta \mid \forall \alpha \in \Delta.\ \delta(\alpha) \in \mathit{SemType}\}$$

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

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

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

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

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

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

Theorem 24 (Semantic Soundness)**.** *Theorem 19 remains valid with universal types: If* $\Delta\,;\Gamma \vdash e : A$*, then* $\Delta\,;\Gamma \vDash e : A$*.*

Proof. The existing cases need to be adapted to the extended model with type variable substitutions. This is a straight-forward exercise. We present the cases for universal types.

Case 1: $\Lambda.\ e : \forall\alpha.\ B$.
By inversion, $\Delta, \alpha\,;\Gamma \vdash e : B$.
Suppose $\delta \in \mathcal{D}[\![\Delta]\!], \gamma \in \mathcal{G}[\![\Gamma]\!]\delta$.
It remains to show that $\gamma(\Lambda.\ e) \in \mathcal{E}[\![\forall\alpha.\ B]\!]\delta$.
Thus, it suffices to show $\Lambda.\ \gamma(e) \in \mathcal{V}[\![\forall\alpha.\ B]\!]\delta$.
Suppose $S \in \mathit{SemType}$.
It suffices to show $\gamma(e) \in \mathcal{E}[\![B]\!](\delta, \alpha \mapsto S)$.
By indunction, $\Delta, \alpha \mapsto S\,;\Gamma \vDash e : B$.
We have $(\delta, \alpha \mapsto S) \in \mathcal{D}[\![\Delta, \alpha]\!]$.
It remains to show that $\gamma \in \mathcal{G}[\![\Gamma]\!](\delta, \alpha \mapsto S)$.
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*

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

Case 2: $e\ \langle\rangle$.
By inversion: $\Delta\,;\Gamma \vdash e : \forall\alpha.\ B$ and $\Delta \vdash A$.
By induction, $\Delta\,;\Gamma \vDash e : \forall\alpha.\ B$.
Suppose $\delta \in \mathcal{D}[\![\Delta]\!], \gamma \in \mathcal{G}[\![\Gamma]\!]\delta$.
It remains to show that $\gamma(e\ \langle\rangle) \in \mathcal{E}[\![B[A/\alpha]]\!]\delta$.
Thus, it suffices to show $\gamma(e)\ \langle\rangle \in \mathcal{E}[\![B[A/\alpha]]\!]\delta$.
We have $\gamma(e) \in \mathcal{E}[\![\forall\alpha.\ B]\!]\delta$, i.e. there exists $v \in \mathcal{V}[\![\forall\alpha.\ B]\!]\delta$.
We have $(\gamma(e))\ \langle\rangle \rightsquigarrow^\star v\ \langle\rangle$.
It remains to show that $v\ \langle\rangle \in \mathcal{E}[\![B[A/\alpha]]\!]\delta$.
We pick $S = \mathcal{V}[\![A]\!]\delta$.
Pick $\delta' = (\delta, \alpha \mapsto S)$.
By the definition of $\mathcal{V}[\![\cdot]\!]$, we have: $v = \Lambda.\ e'$, and $e' \in \mathcal{E}[\![B]\!]\delta'$.
It suffices to show that $v\ \langle\rangle \in \mathcal{E}[\![B]\!]\delta'$.
Again, to finish this case of the proof we rely on an auxiliary lemma:

Lemma (Boring Lemma 2)**.**

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

□

Exercise 10 Extend the proof of semantic soundness (*i.e.*, termination) with existential types. •

2.3 Free Theorems

Our model allows us to prove several theorems about specific universal types. This class of theorems was coined "free theorems" by Philip Wadler in "Theorems for Free!" (1989).

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

Proof.
Suppose, by way of contradiction, $\vdash e : \forall\alpha.\ \alpha$.
By Theorem 24, $e \in \mathcal{E}[\![\forall\alpha.\ \alpha]\!]$.
We have $e \downarrow v \in \mathcal{V}[\![\forall\alpha.\ \alpha]\!]$.
We pick $S = \emptyset$.
Then $v = \Lambda.\ e'$ and $e' \in \mathcal{E}[\![\alpha]\!](\alpha \mapsto \emptyset)$.
Thus, $e' \downarrow v' \in \mathcal{V}[\![\alpha]\!](\alpha \mapsto \emptyset)$.
Hence, $v' \in \emptyset$. □

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.
Suppose $\vdash f : \forall\alpha.\ \alpha \to \alpha$.
By Theorem 24, $f \downarrow f_v \in \mathcal{V}[\![\forall\alpha.\ \alpha \to \alpha]\!]$.
Pick $S = \{v\}$.
We have $f_v = \Lambda.\ e'$ and $e' \in \mathcal{E}[\![\alpha \to \alpha]\!](\alpha \mapsto S)$. (In the future, we will sometimes write this as $\mathcal{E}[\![S \to S]\!]$.)
Because $v \in S$, we have $e'\ v \in \mathcal{E}[\![S]\!]$ (this is essentially semantic soundness of application) and thus $e'\ v \downarrow v$.
Putting it all together, we have $f\ \langle\rangle\ v \rightsquigarrow^*\ f_v\ \langle\rangle\ v = (\Lambda.\ e')\ \langle\rangle\ v \rightsquigarrow e'\ v \downarrow v$. □

Exercise 11 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 12 Prove the following: Given two closed types A_1, A_2 and a closed term f of type $\forall\alpha.\ (A_1 \to A_2 \to \alpha) \to \alpha$, and given a closed type A and a closed term g of type $A_1 \to A_2 \to A$, there exist values v, v_1, v_2 such that

(i) $f\ \langle\rangle\ g \downarrow v$

(ii) $g\ v_1\ v_2 \downarrow v$

Essentially, this proves that f can do nothing but call g with some arguments. •

2.4 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 $\boxed{e_1 \rightsquigarrow e_2}$

$$\ldots \qquad \frac{}{\mathsf{assert}\ \mathsf{true} \rightsquigarrow ()}\ \textsc{assert-true}$$

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

Consider the following signature.

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

We can implement this signature as follows.

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

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 24) 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.

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

We encode the sequential composition $e_1\ ;\ e_2$ as $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ 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 pick $S := \{\overline{0}, \overline{1}\}$.
We now have $v \in S \Rightarrow (v == \overline{0} \vee v == \overline{1}) \downarrow \mathsf{true}$.
It suffices to show that

$$\begin{aligned}&\{\text{bit} := \overline{0},\ \text{flip} := \lambda x.\ \mathsf{assert}\ (x == \overline{0} \vee x == \overline{1})\ ;\ \overline{1} - x,\ \text{get} := \lambda x.\ \mathsf{assert}\ (x == \overline{0} \vee x == \overline{1})\ ;\ x > \overline{0}\}\\ &\in \mathcal{V}[\![\{\text{bit} : \alpha,\ \text{flip} : \alpha \to \alpha,\ \text{get} : \alpha \to \mathsf{bool}\}]\!](\alpha \mapsto S)\end{aligned}$$

Thus, we are left with three cases.

Case 1: bit.
To show: $\overline{0} \in \mathcal{V}[\![\alpha]\!](\alpha \mapsto S)$. This is true, since $\overline{0} \in \{\overline{0}, \overline{1}\}$

Case 2: flip.
To show: $(\lambda x.\ \mathsf{assert}\ (x == \overline{0} \vee x == \overline{1})\ ;\ \overline{1} - x) \in \mathcal{V}[\![\alpha \to \alpha]\!](\alpha \mapsto S)$.
Suppose $v \in S$.
To show: $(\mathsf{assert}\ (v == \overline{0} \vee v == \overline{1})\ ;\ \overline{1} - v) \in \mathcal{E}[\![\alpha]\!](\alpha \mapsto S)$.
Since $v \in S$, we have $(\mathsf{assert}\ (v == \overline{0} \vee v == \overline{1})\ ;\ \overline{1} - v) \rightsquigarrow ((\)\ ;\ \overline{1} - v) \rightsquigarrow \overline{1} - v$.
We conclude $1 - v \in S$.

Case 3: 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 24). 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 13 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

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

(For simplicity, we are using named records, which can easily be compiled down to pairs.)

This exercise is about proving that toint will only ever yield even numbers.

a) Define a function $\mathit{even} : \mathsf{int} \to \mathsf{bool}$ that tests whether a number is even. As usually, you can use addition, subtraction, and the comparison operators – as well as recursive functions.

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

c) 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 14 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 $\mathrm{MySum}(A,B) \in \mathcal{V}[\![\mathrm{SUM}(A,B)]\!]$. •

3 Recursive Types

We extend our language with recursive types. These types allow us to encode recursive data structures that we are used to program with. A typical example is that of the list type:

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

Since list mentions itself, it is a recursive type. We use $\approx$ here, because we will not make list be defined 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".

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

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

$$\ldots \qquad \frac{\Delta\,;\Gamma \vdash E : A[\mu\alpha.\ A/\alpha]}{\Delta\,;\Gamma \vdash \mathsf{roll}_{\mu\alpha.\ A}\ E : \mu\alpha.\ A}\ \textsc{roll} \qquad \frac{\Delta\,;\Gamma \vdash E : \mu\alpha.\ A}{\Delta\,;\Gamma \vdash \mathsf{unroll}_{\mu\alpha.\ A}\ E : A[\mu\alpha.\ A/\alpha]}\ \textsc{unroll}$$

Contextual operational semantics $\boxed{e_1 \rightsquigarrow e_2}$

$$\ldots \qquad \mathsf{unroll}\ \mathsf{roll}\ v \rightsquigarrow v\ \textsc{unroll}$$

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.

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

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 15 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.

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

$$\frac{e_1 \rightsquigarrow e_1'}{\mathsf{app}(e_1, e_2) \rightsquigarrow \mathsf{app}(e_1', e_2)} \qquad \frac{\vdash v_1 : D \qquad e_2 \rightsquigarrow e_2'}{\mathsf{app}(v_1, e_2) \rightsquigarrow \mathsf{app}(v_1, e_2')} \qquad \mathsf{app}(\mathsf{lam}\ x.\ e, v) \rightsquigarrow^\star e[v/x]$$

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 the beta reduction.

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

We can now show that the λ-calculus with recursive types has a well-typed divergent term. To do this, we define a term $\bot$ that reduces to itself.

$$\begin{aligned} \omega &:= \mathsf{lam}\ x.\ \mathsf{app}(x, x) = \mathsf{roll}\ \lambda x.\ (\mathsf{unroll}\ x)\ x \\ \bot &:= \mathsf{app}(\omega, \omega) = (\mathsf{unroll}\ \omega)\ \omega \rightsquigarrow^\star (\lambda x.\ (\mathsf{unroll}\ x)\ x)\ \omega \rightsquigarrow (\mathsf{unroll}\ \omega)\ \omega = \bot \end{aligned}$$

Exercise 16 (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. Let $C := A \to B$.

Define a value form $\mathsf{fix}_C\ f\ x.\ e$ satisfying the following:

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

and

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

the latter assuming C is closed. •

3.2 Girard's Typecast Operator ("J")

We will now show that we can obtain divergence – and even 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 type and semantics. Technically, we have to change the type of runtime terms and the reduction rules to have types in them. We will not spell out that change here.

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

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

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

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.

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

It remains to check the reduction rule for $\mathsf{unroll}\ (\mathsf{roll}\ v)$.

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

Exercise 17 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_A and unroll_A, 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,

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

•

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 2.4, 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 have to resort to the technique of *step-indexing*. This technique has been originally developed by Appel and McAllester in 2001, and by Ahmed in 2004.

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 $n \in \mathbb{N}$ and a closed value v. Intuitively, $(m, v) \in \mathcal{V}[\![A]\!]\delta$ means that no well-typed program using v at type A will "go wrong" in m 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 $(m, v) \in \mathcal{V}[\![A]\!]\delta$, then $\forall j < m.\ (j, v) \in \mathcal{V}[\![A]\!]\delta$.

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

Step-indexed evaluation

$\boxed{e \searrow^m e'}$

$$\frac{\forall e'.\ e \not\rightarrow e'}{e \searrow^0 e} \qquad\qquad \frac{e \rightarrow e' \qquad e' \searrow^m e''}{e \searrow^{m+1} e''}$$

Notice that, unlike the $e \downarrow v$ "evaluates to"-relation, e' does *not have to be a value*. All $e \searrow^m e'$ says is that e will stop computing after m steps, and it will end up in term e'. It could either be stuck (*i.e.*, have crashed), or arrived at a value.

Exercise 18 When showing semantic soundness of step-indexing, we will rely on a few lemmas stating basic properties of the reduction relations.

Prove the following statements.

Lemma 25. *If $K[e]$ is a value, then so is e.*

Lemma 26. *If e is not a value, and $K[e] \rightarrow e'$, then there exists an e'' such that $e' = K[e'']$ and $e \rightarrow e''$.*

Note: It is okay if you only work out the cases for the empty context, and the two application cases.
Hint: The proof of the previous lemma proceeds by induction over K. It is worth first proving the following lemma: If $e_1\ e_2 \rightarrow e'$, then one of the following cases holds:

(i) There exist e_1' s.t. $e_1 = \lambda x.\ e_1'$ and e_2 is a value and $e' = e_1'[e_2/x]$.

(ii) There exist e_1' s.t. $e' = e_1'\ e_2$ and $e_1 \rightarrow e_1'$.

(iii) There exist e_2' s.t. e_1 is a value and $e' = e_1\ e_2'$ and $e_2 \rightarrow e_2'$.

Lemma 27. *If $K[e] \searrow^m e'$, then there exists $j \leq m$ and an e'' such that $e \searrow^j e''$ and $K[e''] \searrow^{m-j} e'$.*

•

Now, we can re-define the semantic model.

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

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

Semantic Type Relation $\boxed{\delta \in \mathcal{D}[\![\Delta]\!]}$

$$\mathcal{D}[\![\Delta]\!] = \{\delta \mid \forall \alpha \in \Delta.\, \delta(\alpha) \in \mathit{SemType}\}$$

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

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

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

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

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

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

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

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

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 m steps on *any* non-deterministic execution,

then it cannot be in the expression relation at step-index $m+1$. 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 28 (Safety)**.** *A program e is* safe *if it does not get stuck, i.e., if for all m and e' such that $e \searrow^m e'$, 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 2.4, 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 19 Prove that the value relation $\mathcal{V}[\![A]\!]\delta$ and the expression relation $\mathcal{E}[\![A]\!]\delta$ are monotone with respect to the step-index:

If $(m, v) \in \mathcal{V}[\![A]\!]\delta$, then $\forall j < m.\ (j, v) \in \mathcal{V}[\![A]\!]\delta$. Furthermore, if $(m, v) \in \mathcal{E}[\![A]\!]\delta$, then $\forall j < m.\ (j, v) \in \mathcal{E}[\![A]\!]\delta$. •

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

Lemma 29 (Bind)**.** *If $(m, e) \in \mathcal{E}[\![A]\!]\delta$,*
and $\forall j \leq m.\ \forall v.\ (j, v) \in \mathcal{V}[\![A]\!]\delta \Rightarrow (j, K[v]) \in \mathcal{E}[\![B]\!]\delta$,
then $(m, K[e]) \in \mathcal{E}[\![B]\!]\delta$.

Proof. Suppose $j < m$, $K[e] \searrow^j e'$.
To show: $(m - j, e') \in \mathcal{V}[\![B]\!]$.
By Lemma 27, 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 our first assumption, $(m - j_1, e_1) \in \mathcal{V}[\![A]\!]\delta$.
By our second assumption, $(m - j_1, K[e_1]) \in \mathcal{E}[\![B]\!]\delta$.
From $(m - j_1, K[e_1]) \in \mathcal{E}[\![B]\!]\delta$ and $K[e_1] \searrow^{j-j_1} e'$, we get $(m - j_1 - (j - j_1), e') \in \mathcal{V}[\![B]\!]\delta$.
With a bit of arithmetic, this is equivalent to $(m - j, e') \in \mathcal{V}[\![B]\!]\delta$, so we are done. □

Lemma 29 lets us compute parts of expressions down to values, if we know that these parts 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 of determinism of the reduction relation.

Lemma 30 (Closure under Expansion)**.**
If e reduces deterministically for j steps, and if $e \rightsquigarrow^j e'$ and $(m, e') \in \mathcal{E}[\![A]\!]\delta$,
then $(m + j, e) \in \mathcal{E}[\![A]\!]\delta$.

Proof. Suppose $i < m + j$ and $e \searrow^i e''$.
To show: $(m + j - i, e'') \in \mathcal{V}[\![A]\!]\delta$.
By our first two assumptions, $e' \searrow^{i-j} e''$.
We have $i - j < n$.
Thus, by our third assumption, $(m - (i - j), e'') \in \mathcal{V}[\![A]\!]\delta$.
This is the same as $((m + j) - i, e'') \in \mathcal{V}[\![A]\!]\delta$, which is what we had to show. □

Lemma 31 (Value Inclusion)**.** *If* $(m, e) \in \mathcal{V}[\![A]\!]\delta$*, then* $(m, e) \in \mathcal{E}[\![A]\!]\delta$*.*

These lemmas will be extremely helpful in our next theorem.

Theorem 32 (Semantic Soundess)**.** *If* $\Delta\,;\Gamma \vdash e : A$*, then* $\Delta\,;\Gamma \vDash e : A$*.*

Proof. Suppose $\delta \in \mathcal{D}[\![\Delta]\!]$, $(m, \gamma) \in \mathcal{G}[\![\Gamma]\!]\delta$.

Case 1: $e = \lambda x.\ e'$ and $A = B \to C$.
To show: $(m, \gamma(\lambda x.\ e')) \in \mathcal{E}[\![B \to C]\!]\delta$.
By definition of substitution, it suffices to show
$(m, \lambda x.\ \gamma e') \in \mathcal{E}[\![B \to C]\!]\delta$.
Since this is already a value, we apply Lemma 31.
It remains to show that $(m, \lambda x.\ \gamma e') \in \mathcal{V}[\![B \to C]\!]\delta$.
Suppose $j \le m$ and $(j, v) \in \mathcal{V}[\![B]\!]\delta$.
To show: $(j, \gamma e[v/x]) \in \mathcal{E}[\![C]\!]\delta$.
We define $\gamma' := \gamma[x \mapsto v]$.
By induction, it suffices to show that
$(j, \gamma') \in \mathcal{G}[\![\Gamma, x : B]\!]\delta$.
Suppose $y : D \in \Gamma$.
To show: $(j, \gamma'(y)) \in \mathcal{V}[\![D]\!]\delta$.

Case 1: $y \in \mathrm{dom}(\gamma')$.
We have $\gamma'(y) = \gamma(y)$ and, by assumption, $(m, \gamma(y)) \in \mathcal{V}[\![D]\!]\delta$.
By monotonicity, $(j, \gamma(y) \in \mathcal{V}[\![D]\!]\delta$.

Case 2: $y = x$ and $D = B$.
We have $\gamma'(x) = v$ and, by assumption, $(j, v) \in \mathcal{V}[\![B]\!]\delta$.

Case 2: $e = e_1\ e_2$ and $A = C$.
To show: $(m, (\gamma\ e_1)\ (\gamma\ e_2))$.
We define $K_1 := \bullet\ (\gamma\ e_2)$.
We have $(\gamma\ e_1)\ (\gamma\ e_2) = K_1[\gamma\ e_1]$.
We apply Lemma 29 with K_1.
(1) To show: $(m, \gamma\ e_1) \in \mathcal{E}[\![B \to C]\!]\delta$.
By induction.
(2) Suppose $j \le m$, $(j, v_1) \in \mathcal{V}[\![B \to C]\!]\delta$.
To show: $(j, K_1[v_1]) \in \mathcal{E}[\![C]\!]\delta$.
We define $K_2 := v_1\ \bullet$.
Thus, we have $K_1[v_1] = v_1\ (\gamma\ e_2) = K_2[\gamma\ e_2]$.
We apply Lemma 29 with K_2.
(1) To show: $(j, \gamma\ e_2) \in \mathcal{E}[\![B]\!]\delta$.
By induction, we have $(m, \gamma\ e_2) \in \mathcal{E}[\![B]\!]\delta$.
By monotonicity, $(j, \gamma\ e_2) \in \mathcal{E}[\![B]\!]\delta$.
(2) Suppose $i \le j$, $(i, v_2) \in \mathcal{V}[\![B]\!]\delta$.
To show: $(i, K_2[v_2]) \in \mathcal{E}[\![C]\!]\delta$, *i.e.*, $(i, v_1\ v_2) \in \mathcal{E}[\![C]\!]\delta$.
We have $v_1 = \lambda x.\ e_1'$ for some e_1, and $(i, e_1'[v_2/x]) \in \mathcal{E}[\![C]\!]\delta$.
By Lemma 30 and beta-reduction being deterministic, $(i + 1, v_1\ v_2) \in \mathcal{E}[\![C]\!]\delta$.
Thus, by monotonicity, $(i, v_1\ v_2) \in \mathcal{E}[\![C]\!]\delta$.

Case 3: $e = \mathsf{roll}\ e_0$ and $A = \mu\alpha.\ B$.
To show: $(m, \mathsf{roll}\ (\gamma\ e_0)) \in \mathcal{E}[\![\mu\alpha.\ B]\!]\delta$.

Define $K := \mathsf{roll}\ \bullet$.
We apply Lemma 29.
(1) To show: $(m, \gamma e_0) \in \mathcal{E}[\![B[\mu\alpha.\ B/\alpha]]\!]\delta$.
By induction.
(2) Suppose $j \leq m$, $(j, v) \in \mathcal{V}[\![B[\mu\alpha.\ B/\alpha]]\!]\delta$.
To show: $(j, K[v]) \in \mathcal{E}[\![\mu\alpha.\ B]\!]\delta$, *i.e.*, $(j, \mathsf{roll}\ v) \in \mathcal{E}[\![\mu\alpha.\ B]\!]\delta$.
By Lemma 31, it suffices to show $(j, \mathsf{roll}\ v) \in \mathcal{V}[\![\mu\alpha.\ B]\!]\delta$.
Suppose $i \leq j$.
To show: $(i, v) \in \mathcal{V}[\![B[\mu\alpha.\ B/\alpha]]\!]\delta$.
By monotonicity.

Case 4: $e = \mathsf{unroll}\ e_0$ and $A = B[\mu\alpha.\ B/\alpha]$.
To show: $(m, \mathsf{unroll}\ (\gamma\ e_0)) \in \mathcal{E}[\![B[\mu\alpha.\ B/\alpha]]\!]\delta$.
Define $K := \mathsf{unroll}\ \bullet$.
We apply Lemma 29.
(1) To show: $(m, \gamma e_0) \in \mathcal{E}[\![\mu\alpha.\ B]\!]\delta$.
By induction.
(2) Suppose $j \leq m$, $(j, v) \in \mathcal{V}[\![\mu\alpha.\ B]\!]\delta$.
To show: $(j, K[v]) \in \mathcal{E}[\![B[\mu\alpha.\ B/\alpha]]\!]\delta$, *i.e.*, $(j, \mathsf{unroll}\ v) \in \mathcal{E}[\![B[\mu\alpha.\ B/\alpha]]\!]\delta$.
We have $v = \mathsf{roll}\ v'$ for some v', and $\forall i < j.\ (i, v') \in \mathcal{V}[\![B[\mu\alpha.\ B/\alpha]]\!]\delta$.
It suffices to show $(j, \mathsf{unroll}\ (\mathsf{roll}\ v')) \in \mathcal{E}[\![B[\mu\alpha.\ B/\alpha]]\!]\delta$.

Case 1: $j = 0$. Trivial by definition of $\mathcal{E}[\![]\!]$.

Case 2: $j > 0$. We execute one step. (Reduction of unroll is deterministic.) Thus, it suffices to show $(j-1, v') \in \mathcal{E}[\![B[\mu\alpha.\ B/\alpha]]\!]\delta$.
We apply $\forall i < j.\ (i, v') \in \mathcal{V}[\![B[\mu\alpha.\ B/\alpha]]\!]\delta$ 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 20 The value relation for the sum type is – unsurprisingly – defined as follows:

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

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

Exercise 21 Consider the following existential type describing an interface for lists:

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

The formal definition of the equality test for integers looks as follows:

$$\overline{n} == \overline{n} \rightsquigarrow \mathsf{true} \qquad\qquad \frac{n \neq m}{\overline{n} == \overline{m} \rightsquigarrow \mathsf{false}}$$

One possible implementation of this interface is to store a list $[v_1, v_2, \ldots, v_n]$ and the length of the list 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

$$\begin{aligned} \mathsf{mylistcase}\ \langle\rangle\ \mathsf{mynil}\ f_1\ f_2 &\rightsquigarrow^*\ f_1 \\ \mathsf{mylistcase}\ \langle\rangle\ (\mathsf{mycons}\ a\ l)\ f_1\ f_2 &\rightsquigarrow^*\ f_2\ a\ l \end{aligned}$$

where a, l, f_1, f_2 are values.

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 MyList(A) is semantically well-typed:

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

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

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

•

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

We have seen the well-typed fixpoint combinator $\mathsf{fix}_C\ f\ x.\ e$. In this section, we will look at a closely-related combinator called z.

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

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

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 expressions 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. The following theorem shows that this reasoning is sound in our model.

Theorem 33 (Löb Induction)**.**
If $\forall m.\ (m-1, e) \in \mathcal{E}[\![A]\!]\delta \Rightarrow (m, e) \in \mathcal{E}[\![A]\!]\delta$*,*
then $\forall m.\ (m, e) \in \mathcal{E}[\![A]\!]\delta$*.*

Proof. By induction on m.

Case 1: $m = 0$. Trivial by definition of $\mathcal{E}[\![A]\!]$.

Case 2: $m > 0$.
By induction, $(m-1, e) \in \mathcal{E}[\![A]\!]\delta$.
To show: $(m, e) \in \mathcal{E}[\![A]\!]\delta$.
This is exactly our assumption. □

Corollary 34.
If $\forall m.\ (\forall j < m.\ (j, e) \in \mathcal{E}[\![A]\!]\delta) \Rightarrow (m, e) \in \mathcal{E}[\![A]\!]\delta$*,*
then $\forall m.\ (m, e) \in \mathcal{E}[\![A]\!]\delta$*.*

To harness the full power of Löb induction, we will eventually apply it on expressions that are functions. In these cases, our goal will often be $(m, e) \in \mathcal{V}[\![A \to B]\!]$. Our Löb induction hypothesis, however, will be $(m', e) \in \mathcal{E}[\![A \to B]\!]$. 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 22 Prove the following lemma:

Lemma 35.
If $(m, \lambda x.\ e) \in \mathcal{E}[\![A \to B]\!]\delta$*,*
then $(m, \lambda x.\ e) \in \mathcal{V}[\![A \to B]\!]\delta$*.*

•

Before we get to the safety proof for Z, we first introduce yet another helpful lemma that will make our life easier. This lemma is extracting the core of the semantic soundness proof of application.

Lemma 36 (Semantic Application)**.** *If* $(m, e_1) \in \mathcal{E}[\![A \to B]\!]$ *and* $(m, e_2) : \mathcal{E}[\![A]\!]$*, then* $(m, e_1\ e_2) \in \mathcal{E}[\![B]\!]$

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

Lemma 37 (Z is safe)**.**

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

Proof. Suppose $\delta \in \mathcal{D}[\![\Sigma]\!]$, $(m, \gamma) \in \mathcal{G}[\![\Gamma]\!]\delta$.
To show: $(m, \gamma\ z) \in \mathcal{E}[\![A \to B]\!]\delta$.
We define $g' := \lambda r.\ \mathsf{let}\ f = \lambda x.\ r\ r\ x\ \mathsf{in}\ \lambda x.\ \gamma\ e$.
Thus, it suffices to show $(m, \lambda x.\ g'\ g'\ x) \in \mathcal{E}[\![A \to B]\!]\delta$.
We apply Theorem 34.
To show: $(\forall m' < m.\ (m', \lambda x.\ g'\ g'\ x) \in \mathcal{E}[\![A \to B]\!]\delta) \Rightarrow (m, \lambda x.\ g'\ g'\ x) \in \mathcal{E}[\![A \to B]\!]\delta$.
Suppose $(\forall m' < m.\ (m', \lambda x.\ g'\ g'\ x) \in \mathcal{E}[\![A \to B]\!]\delta)$.
To show: $(m, \lambda x.\ g'\ g'\ x) \in \mathcal{E}[\![A \to B]\!]\delta$.

As $(\lambda x.\ g'\ g'\ x)$ is already a value, we apply Lemma 31.
To show: $(m, \lambda x.\ g'\ g'\ x) \in \mathcal{V}[\![A \to B]\!]\delta$.
Suppose $j \leq m$ and $(j, v_1) \in \mathcal{V}[\![A]\!]\delta$.
To show: $(j, g'\ g'\ v_1) \in \mathcal{E}[\![B]\!]\delta$.
We apply Lemma 36 and handle the hypotheses in reverse order.
(2) To show: $(j, v_1) \in \mathcal{E}[\![A]\!]\delta$.
By assumption.
(1) To show: $(j, g'\ g') \in \mathcal{E}[\![A \to B]\!]\delta$.
We have $g'\ g' \rightsquigarrow \mathsf{let}\ f = \lambda x.\ g'\ g'\ x\ \mathsf{in}\ \lambda x.\ \gamma\ e \rightsquigarrow \lambda x.\ \gamma\ e[\lambda x.\ g'\ g'\ x/f]$.
Thus, it suffices to show $(j-2, \lambda x.\ \gamma\ e[\lambda x.\ g'\ g'\ x/f]) \in \mathcal{E}[\![A \to B]\!]\delta$.
As this is a value, we apply Lemma 31.
To show: $(j-2, \lambda x.\ \gamma\ e[\lambda x.\ g'\ g'\ x/f]) \in \mathcal{V}[\![A \to B]\!]\delta$.
Suppose $i \leq j$, $(i, v_2) \in \mathcal{V}[\![A]\!]$.
To show: $(i, \gamma\ e[\lambda x.\ g'\ g'\ x/f][v/x]_2) \in \mathcal{E}[\![B]\!]\delta$.
We define $\gamma' = \gamma[f \mapsto \lambda x.\ g'\ g'\ x, x \mapsto v_2]$.
We apply our semantic judgment on e.
It suffices to show $(i, \gamma') \in \mathcal{G}[\![\Gamma, f : A \to B, x : A]\!]$.
(1) Suppose $y : C \in \Gamma$.
We have $\gamma'(y) = \gamma(y)$ and, by monotonicity, $(i, \gamma(y)) \in \mathcal{V}[\![C]\!]\delta$.
(2) We have $\gamma(x) = v_2$ and $(i, v_2) \in \mathcal{V}[\![A]\!]\delta$.
(3) Finally, we have to show $(i, \lambda x.\ g'\ g'\ x) \in \mathcal{V}[\![A \to B]\!]\delta$.
We apply Lemma 35.
To show: $(i, \lambda x.\ g'\ g'\ x) \in \mathcal{E}[\![A \to B]\!]\delta$.
We apply our (Löb) induction hypothesis with $m' := i \leq j - 2 < j \leq m$. □

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.

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

To give operational semantics to the newly-introduced operations, we need to extend our reduction relation with heaps (also called stores in the literature). We use $\emptyset$ to refer to the empty heap. All the existing reduction rules are lifted in the expected way: They work for any heap, and do not change it.

Contextual operational semantics

$$\boxed{h_1 \,;\, e_1 \rightsquigarrow h_2 \,;\, e_2}$$

$$
\ldots \qquad \frac{h \,;\, e \rightsquigarrow h' \,;\, e'}{h \,;\, K[e] \rightsquigarrow h' \,;\, K[e']}\ \textsc{ctx} \qquad \frac{\ell \notin \mathrm{dom}(h)}{h \,;\, \mathsf{new}\ v \rightsquigarrow h[\ell \mapsto v] \,;\, \ell}\ \textsc{new} \qquad \frac{h(\ell) = v}{h \,;\, {*\ell} \rightsquigarrow h \,;\, v}\ \textsc{deref}
$$

$$
\frac{h(\ell) = v'}{h \,;\, \ell \leftarrow v \rightsquigarrow h[\ell \mapsto v] \,;\, ()}\ \textsc{assign}
$$

Notice that if we say $h(\ell) = v$, this implicitly also asserts that $\ell \in \mathrm{dom}(h)$. 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. This is due to there being no location case. In the runtime term typing rules, we are faced with the problem of assigning a type to a location. This is very similar to the VAR rule: we have to look up what the type should be in the variable context. Thus, we introduce a heap typing context, denoted Σ, and assign a type to the location by querying that context.

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

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

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

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

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 $() \to \overline{n}$, 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.

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

To illustrate the operational semantics of the newly introduced operations, we investigate the execution of this program under an empty heap.

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

Exercise 23 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:

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

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 $\forall\alpha.\ \text{STACK}(\alpha)$, and behaves like the Java class (*i.e.*, like an imperative stack):

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

•

Exercise 24 (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 25 (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 Type Safety

We continue with the proof of type safety. As usual, we adapt the proofs of progress and preservation. With this particular extension to the language, we also have to re-state composition and decomposition of contexts, as the typing of contexts now takes into account the heap typing.

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

$$\ldots \qquad \frac{\Sigma \vdash K : A \Rightarrow B}{\Sigma \vdash \mathsf{new}\ K : A \Rightarrow \mathsf{ref}\ B}\ \text{NEW} \qquad \frac{\Sigma \vdash K : A \Rightarrow \mathsf{ref}\ B}{\Sigma \vdash {}^*K : A \Rightarrow B}\ \text{DEREF}$$

$$\frac{\Sigma \vdash K : A \Rightarrow \mathsf{ref}\ B \qquad \Sigma\,;\emptyset\,;\emptyset \vdash e : B}{\Sigma \vdash K \leftarrow e : A \Rightarrow \mathbf{1}}\ \text{ASSIGN-L} \qquad \frac{\Sigma\,;\emptyset\,;\emptyset \vdash v : \mathsf{ref}\ B \qquad \Sigma \vdash K : A \Rightarrow B}{\Sigma \vdash v \leftarrow K : A \Rightarrow \mathbf{1}}\ \text{ASSIGN-R}$$

Lemma 38 (Composition)**.**
If $\Sigma\,;\emptyset\,;\emptyset \vdash e : B$ *and* $\Sigma \vdash K : B \Rightarrow A$*, then* $\Sigma\,;\emptyset\,;\emptyset \vdash K[e] : A$*.*

Lemma 39 (Decomposition)**.**
If $\Sigma\,;\emptyset\,;\emptyset \vdash K[e] : A$*, then* $\Sigma\,;\emptyset\,;\emptyset \vdash e : B$ *and* $\Sigma \vdash K : B \Rightarrow A$*.*

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

Lemma 40 (Σ-Weakening)**.** *If* $\Sigma\,;\Delta\,;\Gamma \vdash e : A$ *and* $\Sigma' \supseteq \Sigma$*, then* $\Sigma'\,;\Delta\,;\Gamma \vdash e : A$*.*

Lemma 41 (Contextual Σ-Weakening)**.** *If* $\Sigma \vdash K : A$ *and* $\Sigma' \supseteq \Sigma$*, then* $\Sigma' \vdash K : A$*.*

We continue with the proof of progress. The statement of progress used to be (and still is) about well-typed expressions. With the addition of heaps, we also need to make sure that the heap matches the same heap typing used for typing the expression: We need a notion of a heap being well-typed.

To this end, we introduce an new judgment which connects heaps and heap typing contexts. 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.

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

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

Additionally, we now 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 42 (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: $e = \mathsf{new}\ e'$. By induction, we have

Case 1:. There exist e'', h' s.t. $h\,; e' \rightsquigarrow h'\,; e''$.
We pick $K = \mathsf{new}\ \bullet$ and apply CTX.
Thus, $h\,; K[e'] = h\,; \mathsf{new}\ e' \rightsquigarrow h'\,; \mathsf{new}\ e'' = h\,; K[e'']$.

Case 2:. e' is a value.
By NEW, $h\,; \mathsf{new}\ e' \rightsquigarrow h[\ell \mapsto e']\,; \ell$ s.t. $\ell \notin \mathrm{dom}(h)$.

Case 3: $e = *e'$. By induction, we have

Case 1:. There exist e'', h' s.t. $h\,; e' \rightsquigarrow h'\,; e''$.
We pick $K = *\bullet$ and apply CTX.
Thus, $h\,; K[e'] = h\,; *e' \rightsquigarrow h'\,; *e'' = h\,; K[e'']$.

Case 2:. e' is a value.
By inversion on the typing derivation of e', there exists ℓ s.t. $e' = \ell \wedge \ell : \mathsf{ref}\ A \in \Sigma$.
By $h : \Sigma$, we have $\Sigma\,;\emptyset\,;\emptyset \vdash h(\ell) : A$ and, thus, that $h(\ell)$ is defined.
By DEREF, $h\,; *e' \rightsquigarrow h\,; h(\ell)$. □

Exercise 26 Do the remaining case of the proof of progress for the latest version of our language: $e = e_1 \leftarrow e_2$. •

We proceed with the proof of preservation. Again, changes have to be made to take into account the heap. We would like to treat the heap and its typing derivation in a similar way to how we treat the expression. Thus, we require the heap on the right hand side of the reduction relation to be well-typed. However, it will become apparent that we can not use the same heap typing under which the original heap was well-typed: We have to allow new locations to be allocated at any type. At the same time, we have to ensure that *existing* locations remain in the heap typing, and keep their type. Thus, we existentially quantify over a larger heap typing.

Lemma 43 (Preservation)**.**
If $\Sigma\,;\emptyset\,;\emptyset \vdash e : A$*,* $h : \Sigma$*, and* $h\,; e \rightsquigarrow h'\,; e'$*,*
then there exists $\Sigma' \supseteq \Sigma$ *s.t.* $h' : \Sigma'$ *and* $\Sigma'\,;\emptyset\,;\emptyset \vdash e' : A$*.*

Proof. By induction on the reduction of $h\,; e'$. Existing cases (except for CTX) remain mostly unchanged. (We have $h' = h$ and pick $\Sigma' = \Sigma$.)

Case 1: CTX.
We have $h\,; K[e_1] \rightsquigarrow h'\,; K[e_1']$ and $h\,; e_1 \rightsquigarrow h'\,; e_1'$.
By Lemma 39, there exists B s.t. $\Sigma\,;\emptyset\,;\emptyset \vdash e : B$ and $\Sigma \vdash K : B \Rightarrow A$.
By induction, there exists $\Sigma' \supseteq \Sigma$ s.t. $h' : \Sigma'$ and $\Sigma'\,;\emptyset\,;\emptyset \vdash e_1' : B$.
It remains to show $\Sigma'\,;\emptyset\,;\emptyset \vdash K[e_1'] : A$.
By Lemma 41, $\Sigma' \vdash K : B \Rightarrow A$.
Thus, by Lemma 38, $\Sigma'\,;\emptyset\,;\emptyset \vdash K[e_1] : A$.

□

Exercise 27 Do the remaining cases of the proof of preservation: new , *, $\leftarrow$. •

4.3 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).

let val x = ref nil	$x : \forall\alpha.\ \alpha$ list ref
in $x := [5, 6]$;	x : int list ref
(hd (!x)) (7)	x : (int → int) list ref

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 was coined "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.

let $x = \Lambda$. ref nil	$x : \forall\alpha.$ ref (list α)
in $x\ \langle\rangle \leftarrow [5, 6]$;	$x\ \langle\rangle$: ref (list int)
(hd (!($x\ \langle\rangle$))) (7)	$x\ \langle\rangle$: ref (list int → int)

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.

4.4 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.

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

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.

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

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.

4.5 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.

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

We update our definition of an expression terminating in m steps with some final term.

Step-indexed evaluation $\boxed{e \searrow^m e'}$

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

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

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

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 a restriction on our language. From here on, our language has only first-order reference. Put differently, the heap is not allowed to contain functions, polymorphic or existential types, or references.

First-Order Types $\boxed{a}$

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

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

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

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

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

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 44. $\vdash v : a \Leftrightarrow (m, W, v) \in \mathcal{V}[\![a]\!]\delta$.

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:

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

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

Theorem 45.

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

Proof. Suppose $j < m$, $h : W$, $h\,;\text{MyMutBit} \searrow^j h'\,;e'$.
We have $j = 2$, $h' = h[\ell \mapsto \overline{0}]$ (for some $\ell \notin \mathrm{dom}(h)$), and

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

We pick $W' := W +\!\!+ \{[\ell \mapsto \overline{0}],\ [\ell \mapsto \overline{1}]\}$. Let $n := |W|$.

a) To show: $W' \sqsupseteq W$. (Trivial.)

b) To show: $h' : W'$.
From $h : W$ we have $h = h_1 \uplus \ldots \uplus h_n$.
Since $\ell \notin \mathrm{dom}(h)$, we have $\forall i \in 1 \ldots n.\ \ell \notin \mathrm{dom}(h_i)$.
Thus, $h' = h_1 \uplus \ldots \uplus h_n \uplus h_{n+1}$ where $h_{n+1} := [\ell \mapsto \overline{0}]$.
It remains to show $\forall i \in 1 \ldots |W'|.\ h_i \in W'[i]$.
With $h : W$, this reduces to $h_{|W|+1} \in W'[|W| + 1]$, *i.e.*, $[\ell \mapsto \overline{0}] \in \{[\ell \mapsto \overline{0}],\ [\ell \mapsto \overline{1}]\}$.

c) To show: $(m - 2, W', e') \in \mathcal{V}[\![(\mathbf{1} \to \mathbf{1}) \times (\mathbf{1} \to \mathsf{bool})]\!]$.
(We only do the case for flip here.)
To show: $(m - 2, W', \lambda y.\ \mathsf{assert}\ (*\ell == 0 \vee *\ell == 1)\ ;\ \ell \leftarrow \overline{1} - *\ell) \in \mathcal{V}[\![\mathbf{1} \to \mathbf{1}]\!]$.
Suppose $j \leq m - 2$, and $W'' \sqsupseteq W'$.
To show: $(j, W'', \mathsf{assert}\ (*\ell == 0 \vee *\ell == 1)\ ;\ \ell \leftarrow \overline{1} - *\ell) \in \mathcal{E}[\![\mathbf{1}]\!]$.
Suppose $j'' < j$, $h'' : W''$, and $h''\,;e'' \searrow^{j''} h'''\,;e'''$.
We have $h'' : W''$, and, thus, $h'' \subseteq h_1 \uplus \ldots \uplus h_n \uplus h_{n+1} \uplus h_{n+2} \uplus \ldots h_{n'}$
with $\forall i \in 1 \ldots n'.\ h_i \in W''[i]$.
Since $W'' \sqsupseteq W'$ we have $h_{n+1} = [\ell \mapsto \overline{0}]$ or $h_{n+1} = [\ell \mapsto \overline{1}]$.
Thus, $h(\ell) = \overline{0}$ or $h(\ell) = \overline{1}$.
We have $h''' = h''[\ell \mapsto \overline{1 - h''(\ell)}]$, $e''' = ()$.
We pick $W''' = W''$.
It remains to show:

i) $(j'', W''', ()) \in \mathcal{V}[\![\mathbf{1}]\!]$. By definition of $\mathcal{V}[\![\mathbf{1}]\!]$.

ii) $h''' : W'''$. We have $[\ell \mapsto h''(\ell)] \in \{[\ell \mapsto \overline{0}],\ [\ell \mapsto \overline{1}]\}$, and, thus, $[\ell \mapsto \overline{1 - h''(\ell)}] \in \{[\ell \mapsto \overline{0}],\ [\ell \mapsto \overline{1}]\}$. □

Lemma 46 (Context decomposition of step-indexed evaluation)**.**
If $h\,;K[e] \searrow^{m} h'\,;e'$,
then $\exists j \leq m, e'', h''.\ h\,;e \searrow^{j} h''\,;e'' \wedge h''\,;K[e''] \searrow^{m-j} h'\,;e'$.

Exercise 28 Prove the following lemma. You may use context decomposition of step-indexed evaluation.

Lemma 47 (Bind)**.** *If* $(m, W, e) \in \mathcal{E}[\![A]\!]\delta$,
and $\forall j \leq m.\ \forall W' \sqsupseteq W.\ \forall v.\ (j, W', v) \in \mathcal{V}[\![A]\!]\delta \Rightarrow (j, W', K[v]) \in \mathcal{E}[\![B]\!]\delta$,
then $(m, W, K[e]) \in \mathcal{E}[\![B]\!]\delta$.

•

Exercise 29 Prove the following lemma:

Lemma 48 (Closure under Expansion)**.**
If e *reduces deterministically for* j *steps under any heap,*
and if $\forall h.\ h\,;e \rightsquigarrow^{j} h\,;e'$ *and* $(m, W, e') \in \mathcal{E}[\![A]\!]\delta$,
then $(m + j, W, e) \in \mathcal{E}[\![A]\!]\delta$.

•

Exercise 30 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 $\forall m, W.\ (m, W, \text{SafeCounter}) \in \mathcal{V}[\![\mathbf{1} \to \text{COUNTER}]\!]$. •

Proof outlines. We use this opportunity to introduce a notation for *proof outlines*, that omits some of the nitty-gritty details of these proofs. In particular, we omit all step-indices. Furthermore, we use disjoint union to express heaps, which makes reasoning about world satisfaction much easier.

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 $\qquad (\_, W_0, \lambda\_.\ e_{\text{ctr}}) \in \mathcal{V}[\![\mathbf{1} \to \text{COUNTER}]\!]$

$W_1 \sqsupseteq W_0$

$(\_, W_1, v_1) \in \mathcal{V}[\![\mathbf{1}]\!]$ $\qquad (\_, W_1, e_{\text{ctr}}) \in \mathcal{E}[\![\text{COUNTER}]\!]$

$h_1 : W_1$

$h_1 ; e_{\text{ctr}} \searrow^{-} h_2 ; e_2$

$h_2 = h_1 \uplus \overbrace{[\ell_1 \mapsto \overline{0}, \ell_2 \mapsto \overline{0}]}^{h_{\text{ctr}}}$

$e_2 = \{\mathsf{inc} = \lambda\_.\ e_{\text{inc}}, \mathsf{get} = \lambda\_.\ e_{\text{get}}\}$

$e_{\text{inc}} := \ell_1 \leftarrow {*\ell_1} + \overline{1} ; \ell_2 \leftarrow {*\ell_2} + \overline{1}$

$e_{\text{get}} := \mathsf{let}\ v1 = {*\ell_1}\ \mathsf{in}\ \mathsf{let}\ v2 = {*\ell_2}\ \mathsf{in}\ \mathsf{assert}\ (v1 == v2) ; v1$

Pick W_2 with new invariant i:

$W_2[i] := H_{\text{ctr}} := \{h \mid \exists n.\ h(\ell_1) = h(\ell_2) = \overline{n}\}$

$h_2 : W_2$ (done by $h_{\text{ctr}} \in H_{\text{ctr}}$)

$(\_, W_2, e_2) \in \mathcal{V}[\![\text{COUNTER}]\!]$

Case inc**:** $\qquad (\_, W_2, \lambda\_.\ e_{\text{inc}}) \in \mathcal{V}[\![\mathbf{1} \to \mathbf{1}]\!]$

$W_3 \sqsupseteq W_2$ $\qquad (\_, W_3, e_{\text{inc}}) \in \mathcal{E}[\![\mathbf{1}]\!]$

$h_3 : W_3$

$h_3 ; e_{\text{inc}} \searrow^{-} h_4 ; e_4$

By world satisfaction: $W_3[i] = H_{\text{ctr}}$

$h_3 = h_3' \uplus \underbrace{[\ell_1 \mapsto \overline{n}, \ell_2 \mapsto \overline{n}]}_{\in H_{\text{ctr}}}$

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 $W_4 := W_3$

$h_4 : W_4$ (done by definition of H_{ctr})

$(\_, W_2, e_4) \in \mathcal{V}[\![\text{COUNTER}]\!]$ (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 $\qquad \boxed{\mathcal{G}[\![\Gamma]\!]\delta}$

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

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

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

Now we can prove the core theorem.

Theorem 49 (Semantic Soundess)**.** *If* $\Delta ; \Gamma \vdash e : A$*, then* $\Delta ; \Gamma \vDash e : A$*.*

Proof.
As before, we proceed by induction on the typing judgment of e.

Case 1: $e = \mathsf{new}\ e'$.
Suppose $\delta \in \mathcal{D}[\![\Delta]\!]$, m, W, γ with $(m, W, \gamma) \in \mathcal{G}[\![\Gamma]\!]\delta$.
By induction, we have $(m, W, \gamma(e')) \in \mathcal{E}[\![A]\!]\delta$.
To show: $(m, W, \mathsf{new}\ (\gamma(e'))) \in \mathcal{E}[\![\mathsf{ref}\ A]\!]\delta$.
We apply Lemma 47 with $K = \mathsf{new}\ \bullet$.
Suppose $j \le m$, $W' \sqsupseteq W$, $(j, W', v) \in \mathcal{V}[\![a]\!]\delta$.
To show: $(j, W', \mathsf{new}\ v) \in \mathcal{E}[\![\mathsf{ref}\ a]\!]\delta$.
Suppose $j' < j$, $h : W'$, $h\,;\mathsf{new}\ v \searrow^{j'} h'\,; e''$.
We have $h' = h[\ell \mapsto v]$, $e'' = \ell$ for some $\ell \notin \mathrm{dom}(h)$.
To show $\exists W'' \sqsupseteq W'.\ h' : W' \wedge (j - j', W'', \ell) \in \mathcal{V}[\![\mathsf{ref}\ a]\!]\delta$.
We pick $W'' = W' +\!\!+ \{[\ell \mapsto \hat{v}] \mid \vdash \hat{v} : a\}$.

a)To show: $h' : W''$. This follows from $h : W'$, our assumption on v, and Lemma 44.

b)To show: $W'' \sqsupseteq W'$. (Trivial)

c)To show: $(m - j', W'', \ell) \in \mathcal{V}[\![\mathsf{ref}\ a]\!]\delta$. By definition of $\mathcal{V}[\![\mathsf{ref}\ a]\!]$. □

4.6 Protocols

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

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

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

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

Now consider the following program.

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

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.

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

Lemma 50. *$\sqsupseteq$ is reflexive and transitive.*

Value Relation

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

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

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 31 (In F^μ* + STSs) This exercise operates in F^μ*, 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 following cases of semantic soundness: $^*e$, $e_1 \leftarrow e_2$, $\lambda x.\ e$, $e_1\ e_2$ •

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

Lemma 51. *Recall the example program from above which we used to motivate the introduction of state transition systems.*

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

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

Proof.

We have: To show:
$(\_, W, e) \in \mathcal{E}[\![(\mathbf{1} \to \mathbf{1}) \to \mathbf{1}]\!]$

$h : W$
$h\,;e \searrow^{-} h'\,;e'$
$\ell \notin \mathrm{dom}(h)$
$h' = h \uplus [\ell \mapsto \overline{3}]$
$e' = \lambda f.\ \ell \leftarrow \overline{4}\,;f\ ()\,;\mathsf{assert}\ (*\ell == \overline{4})$
Pick $W_0[i] := \{\ \Phi := \{(s_3, s_4)\}^\star, \mathrm{c} := s_3, \mathrm{H} := \lambda s_n.\ \{[\ell \mapsto n]\}\ \}$
$h' : W_0$ (done by $[\ell \mapsto \overline{3}] \in W_0[i].\mathrm{H}(s_3)$)
$(\_, W_0, e') \in \mathcal{V}[\![(\mathbf{1} \to \mathbf{1}) \to \mathbf{1}]\!]$

$W_1 \sqsupseteq W_0$
$(\_, W_1, v) \in \mathcal{V}[\![\mathbf{1} \to \mathbf{1}]\!]$
Define $e_1 := \ell \leftarrow \overline{4}\,;v\ ()\,;\mathsf{assert}\ (*\ell == \overline{4})$ $(\_, W_1, e_1) \in \mathcal{E}[\![\mathbf{1}]\!]$

$h_1 : W_1$
$h_1\,;e_1 \searrow^{-} h_2\,;e_2$
$W_1[i] = W_0[i]$ except that $W_1[i].\mathrm{c}$ could be s_4
$h_1[\ell \mapsto \overline{4}]\,;(v\ ()\,;\mathsf{assert}\ (*\ell == \overline{4})) \searrow^{-} h_2\,;e_2$
Define $W_2[i] = W_1[i]$ with $\mathrm{c} := s_4$
$h_1[\ell \mapsto \overline{4}] : W_2$
By Lemma 52
with $h_1[\ell \mapsto \overline{4}] : W_2$
and $h_1[\ell \mapsto \overline{4}]\,;(v\ ()\,;\mathsf{assert}\ (*\ell == \overline{4}))\,; \searrow^{-} h_2\,;e_2$
we get $\exists W_3 \sqsupseteq W_2.\ h_2 : W_3 \wedge (\_, W_3, e_2) \in \mathcal{V}[\![\mathbf{1}]\!]$
Pick W_3 as given
By transitivity of $\sqsupseteq$, $W_3 \sqsupseteq W_1$ □

Lemma 52 (Auxiliary "Lemma")**.** $(\_, W_2, (v\ ()\,;\mathsf{assert}\ (*\ell == \overline{4}))) \in \mathcal{E}[\![\mathbf{1}]\!]$

Proof.

We have: To show:
$(\_, W_2, (v\ ()\,;\mathsf{assert}\ (*\ell == \overline{4}))) \in \mathcal{E}[\![\mathbf{1}]\!]$

By assumption, and def. of $\mathcal{V}[\![\mathbf{1} \to \mathbf{1}]\!]$,
$(\_, W_2, v\ ()) \in \mathcal{E}[\![\mathbf{1}]\!]$
We apply Lemma 47 (the bind lemma) with $K := \bullet\,;\mathsf{assert}\ (*\ell == \overline{4})$
$\forall W_4 \sqsupseteq W_2.\ (\_, W_4, \mathsf{assert}\ *\ell == \overline{4}) \in \mathcal{E}[\![\mathbf{1}]\!]$

$W_4 \sqsupseteq W_2$
$h_4 : W_4$
$h_4\,;\mathsf{assert}\ (*\ell == \overline{4}) \searrow^{-} h_5\,;e_5$
$h_5(\ell) = \overline{4}$ because $W_4 \sqsupseteq W_2$ and $W_2[i].\mathrm{c} = s_4$
$h_5 = h_4 \wedge e_5 = ()$
Pick $W_5 := W_4$ $(\_, W_5, ()) \in \mathcal{V}[\![\mathbf{1}]\!]$
□

4.7 State & Existentials

We present several examples that combine references and existential types to encode stateful abstract data types.

4.7.1 Symbol ADT

Consider the following signature and the corresponding (stateful) implementation.

$$\begin{aligned}
\text{SYMBOL} &:= \exists\alpha.\ \{\ \mathsf{mkSym} : \mathbf{1} \to \alpha,\\
&\qquad\qquad\ \mathsf{check} : \alpha \to \mathbf{1}\ \}\\
\text{Symbol} &:= \mathsf{let}\ c = \mathsf{new}\ \overline{0}\ \mathsf{in}\\
&\qquad \mathsf{pack}\ \{\ \mathsf{mkSym} := \lambda_.\ \mathsf{let}\ x = {*c}\ \mathsf{in}\ c \leftarrow x+1\,;x,\\
&\qquad\qquad\ \mathsf{check} := \lambda x.\ \mathsf{assert}\ (x < c)\ \}
\end{aligned}$$

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.

$$\begin{aligned}
W[i] &:= \{\ \Phi := \{(s_{n_1}, s_{n_2}) \mid (n_2 \geq n_1)\}\,,\\
&\qquad\ \mathrm{c} := s_0,\\
&\qquad\ \mathrm{H} := \lambda s_n.\ \{[\ell_c \mapsto n]\}\ \}\\
\alpha &\mapsto \{(_, W, \overline{n}) \mid 0 \leq n \wedge W[i].\mathrm{c} = s_m \wedge n < m\}
\end{aligned}$$

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.

4.7.2 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.

$$\begin{aligned}
\text{TWIN} &:= \exists\alpha.\, \exists\beta.\, \{\ \text{mkRed} : \mathbf{1} \to \alpha,\\
&\qquad\qquad \text{mkBlue} : \mathbf{1} \to \beta,\\
&\qquad\qquad \text{check} : (\alpha, \beta) \to \mathbf{1}\ \}\\
\text{Twin} &:= \mathsf{let}\ c = \mathsf{new}\ \overline{0}\ \mathsf{in}\\
&\qquad \mathsf{pack}\ \mathsf{pack}\ \{\ \text{mkRed} := \lambda_.\ \mathsf{let}\ x = {*c}\ \mathsf{in}\ c \leftarrow x + 1\,;\, x,\\
&\qquad\qquad\qquad \text{mkBlue} := \lambda_.\ \mathsf{let}\ x = {*c}\ \mathsf{in}\ c \leftarrow x + 1\,;\, x,\\
&\qquad\qquad\qquad \text{check} := \lambda(x, y).\ \mathsf{assert}\ (x \neq y)\ \}
\end{aligned}$$

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 auxiallary 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.

$$\begin{aligned}
W[i] := \Bigg\{ \ & \Phi := \left\{ (s_{n,R,B}, s_{n',R',B'}) \;\middle|\; \begin{array}{l} R \uplus B = \{0 \ldots n-1\} \\ R' \uplus B' = \{0 \ldots n'-1\} \\ n \leq n',\ R \subseteq R',\ B \subseteq B' \end{array} \right\},\\
& \mathrm{c} := s_{0,\emptyset,\emptyset},\\
& \mathrm{H} := \lambda s_{n,R,B}.\ \{[\ell_c \mapsto n]\}\ \Bigg\}\\
\alpha \mapsto\ & \{(_, W, \overline{n}) \mid W[i].\mathrm{c} = s_{m,R,B} \wedge n \in R\}\\
\beta \mapsto\ & \{(_, W, \overline{n}) \mid W[i].\mathrm{c} = s_{m,R,B} \wedge n \in B\}
\end{aligned}$$

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 32 (In F^{μ}* + STSs) Consider the following stateful abstract data type. We

assume we are given some *first-order* types a and b.

$$\begin{aligned}
\text{SUM} := \exists\beta.\ \{\ & \mathsf{setA} : a \to \beta,\\
& \mathsf{setB} : b \to \beta,\\
& \mathsf{getA} : \beta \to \mathbf{1} + a,\\
& \mathsf{getB} : \beta \to \mathbf{1} + b\ \}
\end{aligned}$$

$$\begin{aligned}
\text{MySum} := \lambda_.\ & \mathsf{let}\ x = \mathsf{new}\ \langle \overline{1}, \overline{1}\rangle\ \mathsf{in}\\
& \mathsf{pack}\ \{\ \mathsf{setA} := \lambda y.\ x \leftarrow \langle \overline{1}, y\rangle,\\
& \qquad \mathsf{setB} := \lambda y.\ x \leftarrow \langle \overline{2}, y\rangle,\\
& \qquad \mathsf{getA} := \lambda_.\ \mathsf{let}\ (c, d) = {*x}\ \mathsf{in}\\
& \qquad\qquad \mathsf{if}\ c == \overline{1}\ \mathsf{then}\ \mathsf{inj}_2\ d\ \mathsf{else}\ \mathsf{inj}_1\ (),\\
& \qquad \mathsf{getB} := \lambda_.\ \mathsf{let}\ (c, d) = {*x}\ \mathsf{in}\\
& \qquad\qquad \mathsf{if}\ c == \overline{2}\ \mathsf{then}\ \mathsf{inj}_2\ d\ \mathsf{else}\ \mathsf{inj}_1\ ()\ \}
\end{aligned}$$

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:

$$\forall m, W.\ (m, W, \text{MySum}) \in \mathcal{V}[\![\mathbf{1} \to \text{SUM}]\!]$$

Give only a proof *sketch* (*i.e.*, give only the invariants, the semantic type picked for β, and how the STSs are updated). •

Exercise 33 (In F^{μ}* + STSs) Consider the following code:

$$\begin{aligned}
e_{\text{ctr}} := \ & \mathsf{let}\ c = \mathsf{new}\ \overline{0}\ \mathsf{in}\\
& \lambda n.\ \mathsf{if}\ n > {*c}\ \mathsf{then}\ c \leftarrow n\ \mathsf{else}\ ();\\
& \lambda_.\ \mathsf{assert}\ ({*c} \geq n)
\end{aligned}$$

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:

$$\forall m, W.\ (m, W, e_{\text{ctr}}) \in \mathcal{E}[\![\mathsf{int} \to (\mathbf{1} \to \mathbf{1})]\!]$$

Give a proof *outline* (following the conventions described previously, and as usual ignoring step-indices). •

Exercise 34 (In F^{μ}* + Inv) This exercise operates in F^{μ}* 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:

$e_{\text{id}} := \lambda f.\ \mathsf{let}\ x = \mathsf{new}\ f\ \mathsf{in}\ \lambda y.\ {*x}\ y$

Show that e_{id} is semantically well-typed: For any closed types A, B, prove

$$\forall m, W.\ (m, W, e_{\text{id}}) \in \mathcal{V}[\![(A \to B) \to (A \to B)]\!]$$

Give a proof *outline* (following the conventions described previously, and as usually ignoring step-indices). •