Semantics of Type Systems
Lecture Notes

Derek Dreyer MPI-SWS
Ralf Jung MPI-SWS
Jan-Oliver Kaiser MPI-SWS
Hoang-Hai Dang MPI-SWS
David Swasey MPI-SWS

February 12, 2018

Contents

1 Simply Typed Lambda Calculus

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

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 source 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).

$$\textsc{var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{lam}\ \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.\, e : A \to B} \qquad \textsc{app}\ \frac{\Gamma \vdash e_1 : A \to B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\ e_2 : B}$$

$$\textsc{plus}\ \frac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{int}} \qquad \textsc{int}\ \Gamma \vdash \overline{n} : \mathsf{int}$$

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

Type Erasure

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

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

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

Exercise 1 Prove the Erasure lemma. •

Lemma 2 (Exchange for Curry-style typing)**.** *If* $\Gamma_1, x_1 : A_1, \Gamma_2, x_2 : A_2, \Gamma_3 \vdash e : A$*, then* $\Gamma_1, x_2 : A_2, \Gamma_2, x_1 : A_1, \Gamma_3 \vdash e : A$*.*

Proof. An easy induction on the given derivation. □

Lemma 3 (Weakening for Curry-style typing)**.** *If* $\Gamma \vdash e : A$*, then* $\Gamma, x : B \vdash e : A$*.*

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

Case 1: VAR, $e = y$ and $y : A \in \Gamma$. It suffices to show $y : A \in (\Gamma, x : B)$ by VAR. This is immediate.

Case 2: LAM, $e = \lambda y.\, e'$ and $A = A_1 \to A_2$ and $\Gamma, y : A_1 \vdash e' : A_2$. It suffices to show $\Gamma, x : B, y : A_1 \vdash e' : A_2$ by LAM. By Lemma 2 and our induction hypothesis.

Case 3: APP, $e = e_1\ e_2$. It suffices to show $\Gamma, x : B \vdash e_1$ and $\Gamma, x : B \vdash e_2$ by APP, 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 4 (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, $e = y \neq x$ and $y : B \in (\Gamma, x : A)$. It suffices to show $\Gamma \vdash y : B$ since $y[v/x] = y$. It suffices to show $y : B \in \Gamma$ by VAR. This is immediate.

Case 3: LAM, $e = \lambda y.\ e'$ and $B = B_1 \to B_2$ and $\Gamma, x : A, y : B_1 \vdash e' : B_2$. Note that $(\lambda y.\ e')[v/x] = \lambda y.\ e'[v/x]$. Thus, it suffices to show $\Gamma, y : B_1 \vdash e'[v/x] : B_2$ by LAM. This holds by Lemma 2 and 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 5 (Kosher Terms)**.** *A (runtime) term e is* kosher *if either it is a value or there exists e' s.t. $e \succ e'$.*

Lemma 6 (Canonical forms)**.** *If $\vdash v : A$, then:*

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

Proof. By inversion. □

Theorem 7 (Progress)**.** *If $\vdash e : A$, then e is kosher.*

Theorem 8 (Preservation)**.** *If $\vdash e : A$ and $e \succ e'$, then $\vdash e' : A$.*

Corollary 9 (Type Safety)**.** *If $\vdash e : A$ and $e \succ^\star e'$, then e' is kosher.*

1.2 Contextual Operational Semantics

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

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

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

Primitive reduction $\boxed{e_1 \rightsquigarrow_{\mathrm{p}} e_2}$

BETA
$(\lambda x.\ e)\ v \rightsquigarrow_{\mathrm{p}} e[v/x]$

PLUS
$\overline{n} + \overline{m} \rightsquigarrow_{\mathrm{p}} \overline{n + m}$

Reduction

$\boxed{e_1 \rightsquigarrow e_2}$

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

Exercise 2 Prove that the structural semantics and the contextual semantics are equivalent:

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

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

•

Exercise 3 Prove that both $\succ$ and $\rightsquigarrow$ are deterministic. •

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

Theorem 11 (Progress)**.** *If $\vdash e : A$, then e is kosher.*

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

Case 1: VAR, $e = x$ and $x : A \in \emptyset$. Absurd.

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

Case 3: APP, $e = v_1\ v_2$ and $\vdash v_1 : B \to A$ and $\vdash v_2 : B$. We have $v_1 = \lambda x.\, e_1$ for some x and e_1 by Lemma 6. Thus, $e \rightsquigarrow e_1[v_2/x]$ by BETA.

Case 4: APP, $e = v\ e_2$ where e_2 is not a value and $\vdash e_2 : B$. By induction, there exists e_2' s.t. $e_2 \rightsquigarrow e_2'$. By inversion, we have $e_2 = K[e_3]$ and $e_2' = K[e_3']$ and $e_3 \rightsquigarrow_{\text{p}} e_3'$ for some K, e_3, and e_3'. Thus, $e = (v\ K)[e_3'] \rightsquigarrow (v\ K)[e_3']$ by CTX.

Case 5: APP, $e = e_1\ e_2$ where e_1 is not a value and $\vdash e_1 : B \to A$. By induction, there exists e_1' s.t. $e_1 \rightsquigarrow e_1'$. By inversion, we have $e_1 = K[e_3]$ and $e_1' = K[e_3']$ and $e_3 \rightsquigarrow_{\text{p}} e_3'$ for some K, e_3, and e_3'. Thus, $e = (K\ e_2)[e_3] \rightsquigarrow (K\ e_2)[e_3']$ by CTX.

Case 6: PLUS. Analogous to the cases for application.

Case 7: INT, $e = \overline{n}$. $\overline{n}$ is a value. □

Contextual typing

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

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

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

Lemma 12 (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 $\vdash e' : C$.

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 13 (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.) □

Lemma 14 (Primitive preservation)**.** *If* $\vdash e : A$ *and* $e \rightsquigarrow_{\mathrm{p}} e'$*, then* $\vdash e' : A$*.*

Proof. By cases on $e \rightsquigarrow_{\mathrm{p}} e'$:

Case 1: BETA, $e = (\lambda x.\ e_1)\ v$ and $e' = e_1[v/x]$. It remains to show that $\vdash e_1[v/x] : A$. By inversion on $\vdash e : A$ we have $\vdash \lambda x.\ e_1 : A_0 \to A$ and $\vdash v : A_0$ for some A_0. By inversion on $\vdash \lambda x.\ e_1 : A_0 \to A$ we have $x : A_0 \vdash e_1 : A$. By Lemma 4, we have $\vdash e_1[v/x] : A$.

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

Theorem 15 (Preservation)**.** *If* $\vdash e : A$ *and* $e \rightsquigarrow e'$*, then* $\vdash e' : A$*.*

Proof. Invert $e \rightsquigarrow e'$ to obtain K, e_1, e_1' s.t. $e = K[e_1]$ and $e' = K[e_1']$ and $e_1 \rightsquigarrow_{\mathrm{p}} e_1'$.
By Lemma 12, there exists B s.t. $\vdash K : B \Rightarrow A$ and $\vdash e_1 : B$.
By Lemma 14, from $\vdash e_1 : B$ and $e_1 \rightsquigarrow_{\mathrm{p}} e_1'$, we have $\vdash e_1' : B$.
By Lemma 13, from $\vdash e_1' : B$ and $\vdash K : B \Rightarrow A$, we have $\vdash K[e_1'] : A$, so we are done. □

Corollary 16 (Type Safety)**.** *If* $\vdash e : A$ *and* $e \rightsquigarrow^\star e'$*, then* e' *is kosher.*

Exercise 4 We call an expression e *safe* if for any expression e' s.t. $e \rightsquigarrow^\star e'$, e' is kosher. If e is closed and well-typed, by Type Safety we know that e must be safe. Is there a closed expression that is safe, but *not* well-typed? In other words, is there a closed expression e that is safe but there is no type A s.t. $\vdash e : A$? Give one example if there is such an expression, otherwise prove their non-existence. •

Exercise 5 (Products and Sums) From logic, you know the connectives conjunction ($\wedge$) and disjunction ($\vee$). The corresponding constructs in programming are *products* and

sums. In the following, we will extend our lambda calculus with support for products and sums. Let us start by extending the syntax of the language and the type system:

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

The structural operational semantics of products and sums is defined next:

$$
\ldots \qquad \frac{e_1 \succ e_1'}{\langle e_1, e_2 \rangle \succ \langle e_1', e_2 \rangle}\ \textsc{prod-struct-l} \qquad \frac{e_2 \succ e_2'}{\langle v_1, e_2 \rangle \succ \langle v_1, e_2' \rangle}\ \textsc{prod-struct-r} \qquad \frac{e \succ e'}{\pi_i\ e \succ \pi_i\ e'}\ \textsc{proj-struct} \qquad \pi_i\ \langle v_1, v_2 \rangle \succ v_i\ \textsc{proj}
$$

$$
\frac{e \succ e'}{\mathsf{inj}_i\ e \succ \mathsf{inj}_i\ e'}\ \textsc{inj-struct}
$$

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

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

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

$$
\ldots \qquad \frac{\Gamma \vdash E_1 : A \qquad \Gamma \vdash E_2 : B}{\Gamma \vdash \langle E_1, E_2 \rangle : A \times B}\ \textsc{prod} \qquad \frac{\Gamma \vdash E : A_1 \times A_2}{\Gamma \vdash \pi_i\ E : A_i}\ \textsc{proj} \qquad \frac{\Gamma \vdash E : A_i}{\Gamma \vdash \mathsf{inj}_i^{A_1+A_2}\ E : A_1 + A_2}\ \textsc{inj}
$$

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

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

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

b) Extend the statement of the canonical forms lemma.

c) Extend the definition of evaluation contexts K.

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

e) Extend the definition of the primitive reduction $\rightsquigarrow_{\mathrm{p}}$.

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

g) Extend the contextual typing judgment $\vdash K : A \Rightarrow B$.

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

i) Extend the proof of composition for the new cases.

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

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

1.3 Termination

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

Statement 17 (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 6 Extend the big-step semantics to handle products and sums, as defined previously. •

Exercise 7 Prove that $e \rightsquigarrow^\star v \iff e \downarrow v$ (including products and sums). •

Corollary 18. *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 an 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 19 (Value Inclusion)**.** *If* $e \in \mathcal{V}[\![A]\!]$*, then* $e \in \mathcal{E}[\![A]\!]$*.*

Lemma 20 (Closure under Expansion)**.** *If* $e \rightsquigarrow^\star e'$ *and* $e' \in \mathcal{E}[\![A]\!]$*, then* $e \in \mathcal{E}[\![A]\!]$*.*

Theorem 21 (Semantic Soundness)**.** *If* $\Gamma \vdash e : A$*, then* $\Gamma \vDash e : A$*.*

Proof. By induction on $\Gamma \vdash e : A$, and then using the *compatibility* lemmas of the semantic typing (see Lemma 22, Lemma 23, and Lemma 24). The compatibility lemmas state that we can derive the rules of syntactic typing for semantic typing. As such, the semantic typing is *compatible* with the syntactic typing. In this semantic soundness proofs, for each syntactic rule, the inductive hypotheses give us the semantic premises of the rule. We then only need to apply the corresponding compatibility lemma to close the case.

Compatibility lemmas for VAR, LAM, and APP are Lemma 22, Lemma 23, and Lemma 24, respectively. We omit the compatibility lemmas for INT and PLUS. □

Lemma 22 (Compatibility with VAR)**.** *If* $x : A \in \Gamma$ *then* $\Gamma \vDash x : A$*.*

Proof.

We have:	To show:
$x : A \in \Gamma$	$\Gamma \vDash x : A$
Suppose $\gamma \in \mathcal{G}[\![\Gamma]\!]$	$\gamma(x) \in \mathcal{E}[\![A]\!]$
$\gamma(x) \in \mathcal{V}[\![A]\!]$	
We conclude by value inclusion (Lemma 19).	

□

Lemma 23 (Compatibility with LAM)**.** *If* $\Gamma, x : A \vDash e : B$ *then* $\Gamma \vDash \lambda x.\, e : A \to B$*.*

Proof.

We have:	To show:
$\Gamma, x : A \vDash e : B$	$\Gamma \vDash \lambda x.\ e : A \to B$
Suppose $\gamma \in \mathcal{G}[\![\Gamma]\!]$	$\gamma(\lambda x.\ e) \in \mathcal{E}[\![A \to B]\!]$
	$\lambda x.\ \gamma(e) \in \mathcal{E}[\![A \to B]\!]$
	By Lemma 19, to show $\lambda x.\ \gamma(e) \in \mathcal{V}[\![A \to B]\!]$
Suppose $v \in \mathcal{V}[\![A]\!]$	$\gamma(e)[v/x] \in \mathcal{E}[\![B]\!]$
Let $\gamma' := \gamma[x \mapsto v]$, so $\gamma'(e) = \gamma(e)[v/x]$	
	$\gamma'(e) \in \mathcal{E}[\![B]\!]$
From $\gamma \in \mathcal{G}[\![\Gamma]\!]$ and $v \in \mathcal{V}[\![A]\!]$, have $\gamma' \in \mathcal{G}[\![\Gamma, x : A]\!]$	
We are done by apply $\Gamma, x : A \vDash e : B$.	

□

Lemma 24 (Compatibility with APP)**.** *If* $\Gamma \vDash e_1 : A \to B$ *and* $\Gamma \vDash e_2 : A$ *then* $\Gamma \vDash e_1\ e_2 : B$.

Proof.

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

□

Exercise 8 In the value relation, we define the set of "good" function values as:

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

If we instead define the set as:

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

does the proof of semantic soundness still go through? •

Corollary 25 (Termination)**.** *If* $\emptyset \vdash e : A$*, then there exists* v *s.t.* $e \downarrow v$.

Proof. By Theorem 21, we have $\emptyset \vDash e : A$. Pick γ to be the identity, which clearly is in $\mathcal{G}[\![\emptyset]\!]$. Hence $e \in \mathcal{E}[\![A]\!]$. By definition then, $\exists v.\ e \downarrow v$. □

Exercise 9 Extend the termination proof, which requires extending the semantic soundness proof, to cover products and sums. •

2 System F: Polymorphism and Existential Types

We extend our language with polymorphism and existential types.

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

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

BIGBETA
$$(\Lambda.\, e)\,\langle\rangle \rightsquigarrow_{\mathrm{p}} e$$

UNPACK
$$\mathsf{unpack}\,(\mathsf{pack}\,v)\,\mathsf{as}\,x\,\mathsf{in}\,e \rightsquigarrow_{\mathrm{p}} e[v/x]$$

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 $\boxed{\Delta\,;\Gamma \vdash E : A}$

$$\ldots \qquad \text{LAM}\ \frac{\Delta \vdash A \qquad \Delta\,;\Gamma, x : A \vdash E : B}{\Delta\,;\Gamma \vdash \lambda x : A.\, E : A \to B} \qquad \text{BIGLAM}\ \frac{\Delta, \alpha\,;\Gamma \vdash E : A}{\Delta\,;\Gamma \vdash \Lambda\alpha.\, E : \forall\alpha.\, A}$$

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

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

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

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

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

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

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

Theorem 26 (Progress)**.** *Theorem 11 remains valid: If* $\vdash e : A$*, then* e *is kosher.*

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

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

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

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

Case 4: pack, unpack. See Exercise 10. □

Lemma 27 (Type Substitution)**.**

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

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

Theorem 28 (Preservation)**.**

Theorem 15 remains valid: If $\vdash e : A$ *and* $e \rightsquigarrow e'$*, then* $\vdash e' : A$.

Proof. The proof of preservation remains the same. We only need to update the proof for *primitive* preservation (Lemma 14).

We have:	To show:
Case: bigBeta	
Have $e = (\Lambda.\, e_1)\ \langle\rangle$ and $e' = e_1$	$\vdash e_1 : B[C/\alpha]$
By inversion on $\vdash e : A$, have $\vdash \Lambda.\, e_1 : \forall \alpha.\, B$ s.t. $A = B[C/\alpha]$, $\vdash C$.	
By another inversion, have $\alpha\,;\emptyset \vdash e_1 : B$.	
We are done by type substitution.	
Case: unpack	
See Exercise 10.	

□

Exercise 10 Extend the proofs of progress and preservation to handle existential types.
•

Exercise 11 (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 12 (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 $\mathsf{int} \to \mathsf{int}$. Finally, assume that the language has records types. Records are, syntactically, a bit of a mouthful:

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

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

$$\textsc{record}\quad \frac{\Delta\,;\Gamma \vdash e_1 : A_1 \qquad \cdots \qquad \Delta\,;\Gamma \vdash e_n : A_n}{\Delta\,;\Gamma \vdash \{lab_1 := e_1, \ldots, lab_n := e_n\} : \{lab_1 : A_1, \ldots, lab_n : A_n\}}$$

$$\textsc{project}\quad \{lab_1 := v_1, \ldots, lab_n := v_n\}.lab_i \rightsquigarrow_{\mathrm{p}} v_i \text{ when } 1 \le i \le n$$

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

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

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

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

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

•

2.2 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.

Big-Step Semantics

$\boxed{e \downarrow v}$

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

Semantic Types

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

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

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 29 (Semantic Soundness)**.**

Theorem 21 remains valid: If $\Delta\,;\Gamma \vdash e : A$*, then* $\Delta\,;\Gamma \vDash e : A$*.*

Proof. By induction on $\Delta\,;\Gamma \vdash e : A$ and then using the compatibility lemmas. The existing cases need to be adapted to the extended model with type variable substitutions. This is a straightforward exercise. We present the cases for universal types in Lemma 30 and Lemma 31. You will finish the cases for existential types in Exercise 13. □

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

Lemma 30 (Compatibility for type abstraction; cf. BIGLAM)**.**

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

Proof.

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

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

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

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

□

Lemma 31 (Compatibility for type application; cf. BIGAPP)**.**

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

Proof.

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

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

Lemma (Boring Lemma 2)**.**

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

Lemma (Boring Lemma 3)**.** *If $\delta \in \mathcal{D}[\![\Delta]\!]$ and $\Delta \vdash A$, then $\mathcal{V}[\![A]\!]\delta \in \mathit{SemType}$.* □

Exercise 13 Prove the compatibility lemmas for the cases of 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.

We have:	To show:
Suppose, by way of contradiction, $\vdash e : \forall\alpha.\ \alpha$	$\bot$
By Theorem 29, $e \in \mathcal{E}[\![\forall\alpha.\ \alpha]\!]$, so $e \downarrow v \in \mathcal{V}[\![\forall\alpha.\ \alpha]\!]$	
Pick $S = \emptyset$, then $v = \Lambda.\ e'$ and $e' \in \mathcal{E}[\![\alpha]\!](\alpha \mapsto \emptyset)$	
Hence $e' \downarrow v' \in \emptyset$ for some v'	

□

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

Proof.

We have:	To show:
Suppose $\vdash f : \forall\alpha.\ \alpha \to \alpha$	$f\ \langle\rangle\ v \downarrow v$
By Theorem 29, $f \downarrow f_v \in \mathcal{V}[\![\forall\alpha.\ \alpha \to \alpha]\!]$	
Pick $S = \{v\}$	
We have $f_v = \Lambda.\ e'$ and $e' \in \mathcal{E}[\![\alpha \to \alpha]\!](\alpha \mapsto S)$.	
(We sometimes write this as $\mathcal{E}[\![S \to S]\!]$.)	
From $v \in S$, we have $e'\ v \in \mathcal{E}[\![S]\!]$ and thus $e'\ v \downarrow v$.	
So, $f\ \langle\rangle\ v \rightsquigarrow^* f_v\ \langle\rangle\ v = (\Lambda.\ e')\ \langle\rangle\ v \rightsquigarrow e'\ v \downarrow v$	

□

Exercise 14 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 15 Suppose A_1, A_2, and A are closed types and f is a closed term of type $\forall\alpha.\ (A_1 \to A_2 \to \alpha) \to \alpha$ and g is a closed term of type $A_1 \to A_2 \to A$. Prove that if $f\ \langle\rangle\ g \downarrow v$, then $\exists v_1, v_2.\ g\ v_1\ v_2 \downarrow v$. (Essentially, this means that f can do nothing but call g with some arguments.) •

2.4 Church encodings

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

The empty type

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

The unit type

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

Booleans

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

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

Statement 32. $\mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 \rightsquigarrow^\star e_2$.

Proof.

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

□

Product types

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

Church numerals

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

Statement 33. $\mathsf{iter}_C\ n\ z\ s \rightsquigarrow^\star$ *result of* $s^n(z)$.

Proof. By induction on n. □

Exercise 16 Define a Church encoding for sum types in System F.

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

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

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

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

•

Exercise 17 Lists in System F can be Church encoded as

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

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

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

c) Define a function head of type $\mathsf{list}\ A \to A + \mathbf{1}$. $\mathsf{head}\ l$ should evaluate to $\mathsf{inj}_1\ v$ if l evaluates to a list whose head is v, or $\mathsf{inj}_2\ ()$ if l evaluates to nil.

d) Define a function tail of type $\mathsf{list}\ A \to \mathsf{list}\ A$, which computes the tail of the list.

•

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

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

Statement 34 (Free Theorem for bool values)**.** $\forall v_1, v_2.\ v\ \langle\rangle\ v_1\ v_2 \downarrow v' \in \{v_1, v_2\}$.

Statement 34 allows us to say that $(\mathsf{if}\ v\ \mathsf{then}\ v_1\ \mathsf{else}\ v_2) \downarrow v' \in \{v_1, v_2\}$. This result is a bit weak: it does not guarantee that two executions of $\mathsf{if}\ v\ \mathsf{then}\ v_1\ \mathsf{else}\ v_2$ will evaluate to the same value, because one execution can evaluate to v_1, while the other to v_2. Thus the theorem does not allow us to distinguish true and false values. We actually want the following stronger lemma, which encapsulates the expected behaviors of bool values.

Statement 35 (Expected behaviors of bool values)**.**

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

The lemma states that the application of v either always produces the first value (the then branch), or always produces the second value (the else branch). We show that our semantic model, which is strong enough to show Statement 34, is not strong enough to prove Statement 35. We do this by adding an extension to our language, with which we can build a bool value that satisfies Statement 34 but violates Statement 35.

We extend the language with an expression $\mathsf{type\ is\ bool}\ \langle A\rangle$ which checks if A is bool and returns the check result as a bool value:

$$\frac{A = \mathsf{bool}}{\mathsf{type\ is\ bool}\ \langle A\rangle \rightsquigarrow_{\mathrm{p}} \mathsf{true}} \qquad \frac{A \neq \mathsf{bool}}{\mathsf{type\ is\ bool}\ \langle A\rangle \rightsquigarrow_{\mathrm{p}} \mathsf{false}}$$

With this extension we can build the following value:

$v_{\mathsf{bad}} := \Lambda\alpha.\ \lambda x, y : \alpha.\ \mathsf{if\ type\ is\ bool}\ \langle\alpha\rangle\ \mathsf{then}\ x\ \mathsf{else}\ y$

We can see that v_{bad} has type bool and satisfies Statement 34 but not Statement 35. When instantiated with bool, v_{bad} takes the then branch, but when instantiated with a type that is not bool, it takes the else branch, as is shown below:

$$v_{\mathsf{bad}}\ \langle\mathsf{bool}\rangle\ \mathsf{true}\ \mathsf{false} \rightsquigarrow^{\star} \mathsf{true}$$
$$v_{\mathsf{bad}}\ \langle\mathsf{nat}\rangle\ \overline{0}\ \overline{1} \rightsquigarrow^{\star} \overline{1}$$

Thus, our semantic model does not distinguish different bool values. In order to prove that our Church encodings are faithful encodings, we would need to extend our model to reasoning about *relational parametricity* (see John Reynolds. *Types, Abstraction, and Parametric Polymorphism,* IFIP Congress, 1983).

2.5 Existential types and invariants

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

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

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

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

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

Consider the following signature.

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

We can implement this signature as follows.

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

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 29) 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.

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

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: $\vDash \mathrm{MyBit} : \mathrm{BIT}$.

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

Proof.

We have:	To show:
	$\mathrm{MyBit} \in \mathcal{V}[\![\mathrm{BIT}]\!]$

Pick $S := \{\overline{0}, \overline{1}\}$

It suffices to show

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

Thus, we are left with three cases.

Case: bit $\quad \overline{0} \in \mathcal{V}[\![\alpha]\!](\alpha \mapsto S) = S$

This is true, since $\overline{0} \in \{\overline{0}, \overline{1}\}$

Case: flip $\quad (\lambda x.\ \mathsf{assert}\ (x == \overline{0} \vee x == \overline{1})\ ;\ \overline{1} - x) \in \mathcal{V}[\![\alpha \to \alpha]\!](\alpha \mapsto S)$

Suppose $v \in S$

$$(\mathsf{assert}\ (v == \overline{0} \vee v == \overline{1})\ ;\ \overline{1} - v) \in \mathcal{E}[\![\alpha]\!](\alpha \mapsto S)$$

Since $v \in S$,

have $(\mathsf{assert}\ (v == \overline{0} \vee v == \overline{1})\ ;\ \overline{1} - v) \rightsquigarrow ((\)\ ;\ \overline{1} - v) \rightsquigarrow \overline{1} - v \rightsquigarrow \overline{1 - n}$

where $v = \overline{n}$

$$\overline{1 - n} \in \mathcal{V}[\![\alpha]\!](\alpha \mapsto S) = S$$

We conclude since $\overline{1 - n} \in S$.

Case: get

With similar reasoning as above, we show that the assert succeeds.

□

Note that all our structural syntactic safety rules extend to semantic safety. In other words, if some syntactically well-typed piece of code *uses* MyBit, then the entire program will be semantically safe because every syntactic typing rule preserves the semantic safety. (The entire program cannot be syntactically well-typed, since it contains MyBit.) This is essentially what we prove when we prove Semantic Soundess (Theorem 29). 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 18 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

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

This exercise is about proving that toint will only ever yield even numbers. The function $\mathit{even} : \mathsf{int} \to \mathsf{bool}$ tests whether a number is even.

$$\mathit{even} := \mathsf{rec}\ f\ (x : \mathsf{int}) : \mathsf{int}.$$
$$\mathsf{if}\ x = \overline{0}\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ \mathsf{if}\ x = 1\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ \mathsf{if}\ x < \overline{0}\ \mathsf{then}\ f\ (x + \overline{2})\ \mathsf{else}\ f\ (x - \overline{2})$$

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

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

•

Exercise 19 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 familiar recursive data structures, as well as the recursive functions needed to program with them. A familiar example is the type of integer lists:

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

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

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

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

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

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

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

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

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

$$\cdots \qquad \mathsf{unroll}\ (\mathsf{roll}\ v) \rightsquigarrow_{\mathrm{p}} v\ \textsc{unroll}$$

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 20 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 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 our lanuage with recursive types has a well-typed divergent term. To do this, we define a term $\bot$ that reduces to itself.

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

Exercise 21 (Keep Rollin') Use the roll and unroll primitives introduced in class to encode *typed* fixpoints. Specifically: Suppose you are given types A and B well-formed in Δ.

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

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

and (when A, B closed)

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

•

3.2 Girard's Typecast Operator ("J")

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

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

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

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

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

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

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

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

Exercise 22 Encode the roll and unroll primitives using Girard's cast operator. Specifically: Suppose you are given a type A s.t. $\Delta, \alpha \vdash A$ for some Δ.

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

$\Delta \vdash R_A$

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

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

and, if A is closed,

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

•

3.3 Semantic model: Step-indexing

In this section, we want to develop a semantic model of our latest type system, including recursive types. Clearly, we will no longer be able to use this model to show termination, since we saw that we can now write diverging well-typed terms. However, as we saw in in the discussion about semantic existential types in section 2.5, a semantic model can be helpful even if it does not prove termination: We can use it to show that ill-typed code, like MyBit and MySum, is actually semantically well-typed and hence safe to use from well-typed code.

However, when we try to naively define the value relation for recursive types, it becomes immediately clear that we have a problem: The type in the recursive occurrence of $\mathcal{V}[\![A]\!]\delta$ does not become smaller. To mitigate this, we resort to the technique of *step-indexing,* 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 $k \in \mathbb{N}$ and a closed value v. Intuitively, $(k, v) \in \mathcal{V}[\![A]\!]\delta$ means that no well-typed program using v at type A will "go wrong" in k steps (or less). This intuition also explains why we want these relations to be *monotone* or *downwards-closed* with respect to the step-index: If $(k, v) \in \mathcal{V}[\![A]\!]\delta$, then $\forall j \leq k.\ (j, v) \in \mathcal{V}[\![A]\!]\delta$.

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

Step-indexed termination

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

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

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

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

Exercise 23 Prove the following statements.

Lemma 36. *If $K[e]$ is a value, then so is e.*

Lemma 37. *If e is not a value, and $K[e] \rightsquigarrow e'$, then there exists an e'' such that $e' = K[e'']$ and $e \rightsquigarrow e''$.*

Lemma 38. *If $K[e] \searrow^k e'$, then there exists $j \leq k$ and an e'' such that $e \searrow^j e''$ and $K[e''] \searrow^{k-j} e'$.*

•

Now, we can define our semantic model.

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

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

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

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

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

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

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

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

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

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

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

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

Definition 39 (Safety)**.** *A program e is* safe *if it does not get stuck, i.e., if for all k and e' such that $e \searrow^{k} e'$, e' is a value.*

By this definition, clearly, all semantically well-typed programs are safe. Now that the model no longer proves termination, safety is the primary motivation for even having a semantic model: As we saw in section 2.5, 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 24 (Monotonicity) Prove that the value relation $\mathcal{V}[\![A]\!]\delta$ and the expression relation $\mathcal{E}[\![A]\!]\delta$ are monotone with respect to step-indices:

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

•

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

Lemma 40 (Bind)**.**

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

Proof.

We have:	To show:
(i) $(k, e) \in \mathcal{E}[\![A]\!]\delta$	
(ii) $\forall j \leq k.\ \forall v.\ (j, v) \in \mathcal{V}[\![A]\!]\delta \Rightarrow (j, K[v]) \in \mathcal{E}[\![B]\!]\delta$	$(k, K[e]) \in \mathcal{E}[\![B]\!]\delta$
Suppose $j < k$, $K[e] \searrow^{j} e'$	$(k - j, e') \in \mathcal{V}[\![B]\!]\delta$
By Lemma 38,	
there exist e_1 and $j_1 \leq j$ s.t. $e \searrow^{j_1} e_1$ and $K[e_1] \searrow^{j - j_1} e'$.	
By (i), $(k - j_1, e_1) \in \mathcal{V}[\![A]\!]\delta$.	
By (ii), $(k - j_1, K[e_1]) \in \mathcal{E}[\![B]\!]\delta$.	
With $K[e_1] \searrow^{j - j_1} e'$,	
we get $(k - j_1 - (j - j_1), e') \in \mathcal{V}[\![B]\!]\delta$, so we are done.	

□

Lemma 40 lets us zap subexpressions down to values, if we know that those subexpressions are in the relation. This is extremely helpful when proving that composite terms are semantically well-typed.

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

Lemma 41 (Closure under Expansion)**.**
If e *reduces deterministically for* j *steps, and if* $e \rightsquigarrow^{j} e'$ *and* $(k, e') \in \mathcal{E}[\![A]\!]\delta$,
then $(k + j, e) \in \mathcal{E}[\![A]\!]\delta$.

Proof.

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

□

Lemma 42 (Value Inclusion)**.** *If* $(k, e) \in \mathcal{V}[\![A]\!]\delta$*, then* $(k, e) \in \mathcal{E}[\![A]\!]\delta$*.*

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

These lemmas will be extremely helpful in our next theorem.

Theorem 43 (Semantic Soundess)**.** *If* $\Delta\,;\Gamma \vdash e : A$*, then* $\Delta\,;\Gamma \vDash e : A$*.*

Proof. Again, we do induction on $\Delta\,;\Gamma \vdash e : A$ and then use the compatibility lemmas. We prove a few compatibility lemmas in Lemma 44, Lemma 45, Lemma 46, and Lemma 47. □

Lemma 44 (Compatibility for lambda abstraction; cf. LAM)**.**

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

Proof.

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

□

Lemma 45 (Compatibility for function application; cf. APP)**.**

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

Proof.

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

□

Lemma 46 (Compatibility for roll; cf. ROLL)**.**

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

Proof.

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

□

Lemma 47 (Compatibility for unroll; cf. UNROLL)**.**

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

Proof.

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

□

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

Exercise 25 The value relation for the sum type is—unsurprisingly—defined as follows:

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

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

Exercise 26 Consider the following existential type describing an interface for lists:

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

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

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

a) Implement mylistcase such that

$$\mathsf{mylistcase}\ \langle\rangle\ \mathsf{mynil}\ v\ f \rightsquigarrow^* v$$
$$\mathsf{mylistcase}\ \langle\rangle\ (\mathsf{mycons}\ a\ l)\ v\ f \rightsquigarrow^* f\ a\ l$$

where a, l, v, f are values.

You may assume an operation for testing integer equality:

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

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

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

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

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

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

•

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

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

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

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

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

The way we will prove Z safe is most peculiar. At one point in the proof, we will arrive at what seems to be a circularity: In the process of proving a certain expression safe, the proof obligation reduces to showing that same expression to be safe. It seems like we made no progress at all. However, the step-indices are not the same. In fact, during the proof, we will take steps that decrease the step index of the final goal. The way out of this conundrum is to simply assume the expression to be safe at all lower step indices. The following theorem shows that this reasoning is sound in our model.

Theorem 48 (Löb Induction)**.**
If $\forall j.\ (j-1, e) \in \mathcal{E}[\![A]\!]\delta \Rightarrow (j, e) \in \mathcal{E}[\![A]\!]\delta$,
then $\forall k.\ (k, e) \in \mathcal{E}[\![A]\!]\delta$.

Proof. By induction on k.

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

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

Corollary 49.
If $\forall j.\ (\forall i < j.\ (i, e) \in \mathcal{E}[\![A]\!]\delta) \Rightarrow (j, e) \in \mathcal{E}[\![A]\!]\delta$,
then $\forall k.\ (k, e) \in \mathcal{E}[\![A]\!]\delta$.

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

Exercise 27 Prove the following lemma:

Lemma 50.
If $(k, \lambda x.\ e) \in \mathcal{E}[\![A \to B]\!]\delta$,
then $(k, \lambda x.\ e) \in \mathcal{V}[\![A \to B]\!]\delta$.

•

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

Lemma 51 (Semantic Application)**.** *If* $(k, e_1) \in \mathcal{E}[\![A \to B]\!]\delta$ *and* $(k, e_2) \in \mathcal{E}[\![A]\!]\delta$, *then* $(k, e_1\ e_2) \in \mathcal{E}[\![B]\!]\delta$.

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

Lemma 52 (Z is safe)**.**

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

Proof.

We have:	To show:
(i) $\Delta\,;\Gamma, f : A \to B, x : A \vDash e : B$	$\Delta\,;\Gamma \vDash Z : A \to B$
Suppose $\delta \in \mathcal{D}[\![\Sigma]\!], (k, \gamma) \in \mathcal{G}[\![\Gamma]\!]\delta$	$(k, \gamma\ Z) \in \mathcal{E}[\![A \to B]\!]\delta$
Set $g' := \lambda r.\ \mathsf{let}\ f = \lambda x.\ r\ r\ x\ \mathsf{in}\ \lambda x.\ \gamma\ e$	$(k, \lambda x.\ g'\ g'\ x) \in \mathcal{E}[\![A \to B]\!]\delta$
(ii) $\forall k' < k.\ (k', \lambda x.\ g'\ g'\ x) \in \mathcal{E}[\![A \to B]\!]\delta)$	
by Corollary 49 (Löb induction).	
	By value inclusion, $(k, \lambda x.\ g'\ g'\ x) \in \mathcal{V}[\![A \to B]\!]\delta$
Suppose $j \le k$ and $(j, v_1) \in \mathcal{V}[\![A]\!]\delta$	$(j, g'\ g'\ v_1) \in \mathcal{E}[\![B]\!]\delta$
We apply Lemma 51 and handle the goals in reverse order.	
(2)	$(j, v_1) \in \mathcal{E}[\![A]\!]\delta$
By value inclusion and assumption.	
(1)	$(j, g'\ g') \in \mathcal{E}[\![A \to B]\!]\delta$
Have $g'\ g' \rightsquigarrow \mathsf{let}\ f = \lambda x.\ g'\ g'\ x\ \mathsf{in}\ \lambda x.\ \gamma\ e \rightsquigarrow \lambda x.\ \gamma\ e[\lambda x.\ g'\ g'\ x/f]$.	
	By closure under expansion, $(j-2, \lambda x.\ \gamma\ e[\lambda x.\ g'\ g'\ x/f]) \in \mathcal{E}[\![A \to B]\!]\delta$
	By value inclusion, $(j-2, \lambda x.\ \gamma\ e[\lambda x.\ g'\ g'\ x/f]) \in \mathcal{V}[\![A \to B]\!]\delta$
Suppose $i \le j-2, (i, v_2) \in \mathcal{V}[\![A]\!]$	$(i, \gamma\ e[\lambda x.\ g'\ g'\ x/f][v_2/x]) \in \mathcal{E}[\![B]\!]\delta$
Set $\gamma' = \gamma[f \mapsto \lambda x.\ g'\ g'\ x, x \mapsto v_2]$	$(i, \gamma' e) \in \mathcal{E}[\![B]\!]\delta$
	By applying (i), $(i, \gamma') \in \mathcal{G}[\![\Gamma, f : A \to B, x : A]\!]$
Suppose $y : C \in \Gamma, f : A \to B, x : A$	$(i, \gamma'(y)) \in \mathcal{V}[\![C]\!]\delta$
Case: $y : C \in \Gamma$	
Have $\gamma'(y) = \gamma(y)$	$(i, \gamma(y)) \in \mathcal{V}[\![C]\!]\delta$
From $(k, \gamma) \in \mathcal{G}[\![\Gamma]\!]\delta$, we have $(k, \gamma(y)) \in \mathcal{V}[\![C]\!]\delta$.	
By monotonicity, we are done.	
Case: $y = x, C = A$	$(i, \gamma'(x)) \in \mathcal{V}[\![A]\!]\delta$
Have $\gamma'(x) = v_2$	$(i, v_2) \in \mathcal{V}[\![A]\!]\delta$
By assumption.	
Case: $y = f, C = A \to B$	$(i, \gamma'(f)) \in \mathcal{V}[\![A \to B]\!]\delta$
	$(i, \lambda x.\ g'\ g'\ x) \in \mathcal{V}[\![A \to B]\!]\delta$
	By Lemma 50, $(i, \lambda x.\ g'\ g'\ x) \in \mathcal{E}[\![A \to B]\!]\delta$
We apply our Löb induction hypothesis (ii), with $k' := i \le j-2 < j \le k$.	

□

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

4 Mutable State

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

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

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

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

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

Primitive reduction $\boxed{h_1\,; e_1 \rightsquigarrow_{\mathrm{p}} h_2\,; e_2}$

$$
\ldots \qquad \frac{\ell \notin \mathrm{dom}(h)}{h\,;\mathsf{new}\ v \rightsquigarrow_{\mathrm{p}} h[\ell \mapsto v]\,;\ell}\ \textsc{new} \qquad \frac{h(\ell) = v}{h\,; {*\ell} \rightsquigarrow_{\mathrm{p}} h\,; v}\ \textsc{deref} \qquad \frac{\ell \in \mathrm{dom}(h)}{h\,;\ell \leftarrow v \rightsquigarrow_{\mathrm{p}} h[\ell \mapsto v]\,;()}\ \textsc{assign}
$$

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

$$
\frac{h\,; e \rightsquigarrow_{\mathrm{p}} h'\,; e'}{h\,; K[e] \rightsquigarrow h'\,; K[e']}\ \textsc{ctx}
$$

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, as locations do not arise in the source language.

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

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

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

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

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

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

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 arbitrary heap h:

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

Exercise 28 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 29 (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 30 (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 need to do a little work to extend our proof of syntactic type safety for state.

Our contextual typing judgment must now track the types of locations. In addition to adding a heap context Σ, we have a few new rules:

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

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

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

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

Lemma 53 (Composition)**.**
If $\Sigma\,;\emptyset\,;\emptyset \vdash e : B$ *and* $\Sigma \vdash K : B \Rightarrow A$*, then* $\Sigma\,;\emptyset\,;\emptyset \vdash K[e] : A$*.*

Lemma 54 (Decomposition)**.**
If $\Sigma\,;\emptyset\,;\emptyset \vdash K[e] : A$*, then* $\Sigma\,;\emptyset\,;\emptyset \vdash e : B$ *and* $\Sigma \vdash K : B \Rightarrow A$ *for some* B*.*

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

Lemma 55 (Σ-Weakening)**.** *If* $\Sigma\,;\Delta\,;\Gamma \vdash e : A$ *and* $\Sigma' \supseteq \Sigma$*, then* $\Sigma'\,;\Delta\,;\Gamma \vdash e : A$*.*

Lemma 56 (Contextual Σ-Weakening)**.**
If $\Sigma \vdash K : A \Rightarrow B$ *and* $\Sigma' \supseteq \Sigma$*, then* $\Sigma' \vdash K : A \Rightarrow B$*.*

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

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

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

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

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

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

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

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

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

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

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

Lemma 57 (Progress)**.**
If $\Sigma\,;\emptyset\,;\emptyset \vdash e : A$ *and* $h : \Sigma$*,*
then e *is a value or there exist* e', h' *s.t.* $h\,; e \rightsquigarrow h'\,; e'$*.*

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

Case 1: $e = \ell$. ℓ is a value.

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

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

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

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

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

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

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

Subcase 1: $h\,; e_1 \rightsquigarrow h'\,; e'_1$.
$h\,; e_3 \rightsquigarrow_{\mathrm{p}} h'\,; e'_3$ and $e_1 = K[e_3]$ and $e'_1 = K[e'_3]$ by inversion.
Thus $e = (K \leftarrow e_2)[e_3]$ and we have $h\,; e \rightsquigarrow h'\,; (K \leftarrow e_2)[e'_3]$ by CTX.

Subcase 2: e_1 is a value and $h\,; e_2 \rightsquigarrow h'\,; e'_2$.
$h\,; e_3 \rightsquigarrow_{\mathrm{p}} h'\,; e'_3$ and $e_2 = K[e_3]$ and $e'_2 = K[e'_3]$ by inversion.
Thus $e = (e_1 \leftarrow K)[e_3]$ and we have $h\,; e \rightsquigarrow h'\,; (e_1 \leftarrow K)[e'_3]$ by CTX.

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

Lemma 58 (Primitive preservation)**.**
If $\Sigma\,;\emptyset\,;\emptyset \vdash e : A$ *and* $h : \Sigma$ *and* $h\,; e \rightsquigarrow_{\mathrm{p}} h'\,; e'$,
then there exists $\Sigma' \supseteq \Sigma$ *s.t.* $h' : \Sigma'$ *and* $\Sigma'\,;\emptyset\,;\emptyset \vdash e' : A$.

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

Exercise 31 Prove primitive preservation for new, *, and $\leftarrow$. •

Lemma 59 (Preservation)**.**
If $\Sigma\,;\emptyset\,;\emptyset \vdash e : A$ *and* $h : \Sigma$ *and* $h\,; e \rightsquigarrow h'\,; e'$,
then there exists $\Sigma' \supseteq \Sigma$ *s.t.* $h' : \Sigma'$ *and* $\Sigma'\,;\emptyset\,;\emptyset \vdash e' : A$.

Proof.

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

□

4.3 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 is called the *value restriction* as it restricts implicit let polymorphism to values. To see why this solves the problem, consider the translation of the example above into System F with references.

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 step-indexed termination judgment for state.

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

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

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

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

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 an important restriction on our language. From here on, our language has only first-order references. Put differently, the heap is not allowed to contain functions, polymorphic or existential types, or references. If the heap contains these higher-order types, the semantic model presented below will not work.

First-Order Types $\boxed{a}$

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

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

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

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

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

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

Lemma 60 (First-order types are simple)**.** $\vdash v : a \Leftrightarrow (k, W, v) \in \mathcal{V}[\![a]\!]\delta$.

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

Example Recall our implementation of the MUTBIT signature:

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

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

Theorem 61.

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

Proof.

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

We have:	To show:
	$(k, W, \text{MyMutBit}) \in \mathcal{E}[\![\text{MUTBIT}]\!]$
Suppose $j < k$, $h : W$, $h\,;\text{MyMutBit} \searrow^{j} h'\,;e'$.	$\exists W' \sqsupseteq W.\ h' : W' \wedge (k-j, W', e') \in \mathcal{V}[\![\text{MUTBIT}]\!]$
We have $j = 2$, $h' = h[\ell \mapsto \overline{0}]$ (for some $\ell \notin \text{dom}(h)$), and $e' = e_0(\ell)$.	$\exists W' \sqsupseteq W.\ h' : W' \wedge (k-2, W', e_0(\ell)) \in \mathcal{V}[\![\text{MUTBIT}]\!]$
Let $n := \lvert W\rvert$. Pick $W' := W +\!\!+ \{[\ell \mapsto \overline{0}],\ [\ell \mapsto \overline{1}]\}$.	
	$W' \sqsupseteq W$
Trivial.	
	$h' : W'$
From $h : W$ we have $h = h_1 \uplus \ldots \uplus h_n$.	
Since $\ell \notin \text{dom}(h)$, we have $\forall i \in 1 \ldots n.\ \ell \notin \text{dom}(h_i)$.	
Thus, $h' = h \uplus h_{n+1} \supseteq h_1 \uplus \ldots \uplus h_n \uplus h_{n+1}$ where $h_{n+1} := [\ell \mapsto \overline{0}]$.	
	$\forall i \in 1 \ldots \lvert W'\rvert.\ h_i \in W'[i]$
With $h : W$	$h_{n+1} \in W'[n+1]$
	$[\ell \mapsto \overline{0}] \in \{[\ell \mapsto \overline{0}],\ [\ell \mapsto \overline{1}]\}$
Trivial.	
	$(k-2, W', e_0(\ell)) \in \mathcal{V}[\![(\mathbf{1} \to \mathbf{1}) \times (\mathbf{1} \to \mathsf{bool})]\!]$
We only do the case for flip here.	
	$(k-2, W', \lambda y.\ \mathsf{assert}\ (*\ell == 0 \vee *\ell == 1)\ ;\ \ell \leftarrow \overline{1} - *\ell) \in \mathcal{V}[\![\mathbf{1} \to \mathbf{1}]\!]$
Suppose $j \le k-2$, and $W'' \sqsupseteq W'$.	
	$(j, W'', \mathsf{assert}\ (*\ell == 0 \vee *\ell == 1)\ ;\ \ell \leftarrow \overline{1} - *\ell) \in \mathcal{E}[\![\mathbf{1}]\!]$
Suppose $j'' < j$, $h'' : W''$, and $h''\,;e'' \searrow^{j''} h'''\,;e'''$.	
	$\exists W''' \sqsupseteq W''.\ h''' : W''' \wedge (j - j'', W''', e''') \in \mathcal{V}[\![\mathbf{1}]\!]$
From $h'' : W''$, $h'' \supseteq h_1 \uplus \ldots \uplus h_n \uplus h_{n+1} \uplus h_{n+2} \uplus \ldots h_{n'}$	
with $\forall i \in 1 \ldots n'.\ h_i \in W''[i]$.	
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 $h''(\ell) = \overline{1}$.	
So $h''' = h''[\ell \mapsto \overline{1 - h''(\ell)}]$, $e''' = ()$.	
	$\exists W''' \sqsupseteq W''.\ h''' : W''' \wedge (j - j'', W''', ()) \in \mathcal{V}[\![\mathbf{1}]\!]$
We pick $W''' = W'' \sqsupseteq W''$.	
	$h''' : W''' \wedge (j - j'', W''', ()) \in \mathcal{V}[\![\mathbf{1}]\!]$
	$(j-2, W''', ()) \in \mathcal{V}[\![\mathbf{1}]\!]$
Trivial.	
	$h''' : W'''$
	From $h'' : W''$, it suffices to show $[\ell \mapsto \overline{1 - h''(\ell)}] \in W''[n+1]$
	$[\ell \mapsto \overline{1 - h''(\ell)}] \in \{[\ell \mapsto \overline{0}],\ [\ell \mapsto \overline{1}]\}$
This follows from $h''(\ell) = \overline{0}$ or $h''(\ell) = \overline{1}$.	

□

Lemma 62 (Decomposition of step-indexed termination)**.**
If $h\,;K[e] \searrow^{k} h'\,;e'$,
then $\exists j \leq k, e'', h''.\ h\,;e \searrow^{j} h''\,;e'' \wedge h''\,;K[e''] \searrow^{k-j} h'\,;e'$.

Exercise 32 Prove the bind lemma. You may use the decomposition of step-indexed termination lemma.

Lemma 63 (Bind)**.**
If $(k, W, e) \in \mathcal{E}[\![A]\!]\delta$,
and $\forall j \leq k.\ \forall W' \sqsupseteq W.\ \forall v.\ (j, W', v) \in \mathcal{V}[\![A]\!]\delta \Rightarrow (j, W', K[v]) \in \mathcal{E}[\![B]\!]\delta$,
then $(k, W, K[e]) \in \mathcal{E}[\![B]\!]\delta$.

•

Exercise 33 Prove closure under expansion.

Lemma 64 (Closure under Expansion)**.**
If e *reduces deterministically for* j *steps under any heap,*
and if $\forall h.\ h\,;e \rightsquigarrow^{j} h\,;e'$ *and* $(k, W, e') \in \mathcal{E}[\![A]\!]\delta$,
then $(k + j, W, e) \in \mathcal{E}[\![A]\!]\delta$.

•

Exercise 34 Consider this interface for a counter

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

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

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

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

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

Proof.

We have:	To show:
$e_{\text{ctr}} = \mathsf{let}\ c1 = \mathsf{new}\ \overline{0}\ \mathsf{in}\ \mathsf{let}\ c2 = \mathsf{new}\ \overline{0}\ \mathsf{in}\ \ldots$	
W_0	$(\_, W_0, \lambda\_.\ e_{\text{ctr}}) \in \mathcal{V}[\![\mathbf{1} \to \text{COUNTER}]\!]$
$W_1 \sqsupseteq W_0$	
$(\_, W_1, v_1) \in \mathcal{V}[\![\mathbf{1}]\!]$	$(\_, W_1, e_{\text{ctr}}) \in \mathcal{E}[\![\text{COUNTER}]\!]$
$h_1 : W_1$	
$h_1 ; e_{\text{ctr}} \searrow^{-} h_2 ; e_2$	$\exists W_2 \sqsupseteq W_1.\ h_2 : W_2 \wedge (\_, W_2, e_2) \in \mathcal{V}[\![\text{COUNTER}]\!]$
$h_2 = h_1 \uplus \overbrace{[\ell_1 \mapsto \overline{0}, \ell_2 \mapsto \overline{0}]}^{h_{\text{ctr}}}$	
$e_2 = \{\mathsf{inc} = \lambda\_.\ e_{\text{inc}}, \mathsf{get} = \lambda\_.\ e_{\text{get}}\}$	
$e_{\text{inc}} := \ell_1 \leftarrow {*\ell_1} + \overline{1} ; \ell_2 \leftarrow {*\ell_2} + \overline{1}$	
$e_{\text{get}} := \mathsf{let}\ v1 = {*\ell_1}\ \mathsf{in}\ \mathsf{let}\ v2 = {*\ell_2}\ \mathsf{in}\ \mathsf{assert}\ (v1 == v2) ; v1$	
Pick W_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**:**	$(\_, W_2, \lambda\_.\ e_{\text{inc}}) \in \mathcal{V}[\![\mathbf{1} \to \mathbf{1}]\!]$
$W_3 \sqsupseteq W_2$	$(\_, W_3, e_{\text{inc}}) \in \mathcal{E}[\![\mathbf{1}]\!]$
$h_3 : W_3$	
$h_3 ; e_{\text{inc}} \searrow^{-} h_4 ; e_4$	
	$\exists W_4 \sqsupseteq W_3.\ h_4 : W_4 \wedge (\_, W_4, e_4) \in \mathcal{V}[\![\mathbf{1}]\!]$
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})
	$(\_, W_4, ()) \in \mathcal{V}[\![\mathbf{1}]\!]$ (trivial)

□

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

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

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

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

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

Now we can prove the core theorem.

Theorem 65 (Semantic Soundess)**.** *If* $\Delta ; \Gamma \vdash e : A$*, then* $\Delta ; \Gamma \vDash e : A$*.*

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

Lemma 66 (Compatibility for first-order allocation (cf. NEW))**.**

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

Proof.

We have:	To show:
(i) $\Delta\,;\Gamma \vDash e : a$	$\Delta\,;\Gamma \vDash \mathsf{new}\ e : \mathsf{ref}\ a$
$\delta \in \mathcal{D}[\![\Delta]\!], (k, W, \gamma) \in \mathcal{G}[\![\Gamma]\!]\delta$	$(k, W, \gamma(\mathsf{new}\ e)) \in \mathcal{E}[\![\mathsf{ref}\ a]\!]\delta$
	$(k, W, \mathsf{new}\ \gamma(e)) \in \mathcal{E}[\![\mathsf{ref}\ a]\!]\delta$
$(k, W, \gamma(e)) \in \mathcal{E}[\![a]\!]\delta$ by (i)	
$j \le k, W' \sqsupseteq W, (j, W', v) \in \mathcal{V}[\![a]\!]\delta$	$(j, W', \mathsf{new}\ v) \in \mathcal{E}[\![\mathsf{ref}\ a]\!]\delta$
by bind with $K = \mathsf{new}\ \bullet$	
$j' < j, h : W', h\,;\mathsf{new}\ v \searrow^{j'} h'\,;e''$	
	$\exists W'' \sqsupseteq W'.\ h' : W'' \wedge (j - j', W'', e'') \in \mathcal{V}[\![\mathsf{ref}\ a]\!]\delta$
$j' = 1, h' = h[\ell \mapsto v], e'' = \ell, \ell \notin \mathrm{dom}(h)$ by inversion	
	$\exists W'' \sqsupseteq W'.\ h' : W'' \wedge (j - 1, W'', \ell) \in \mathcal{V}[\![\mathsf{ref}\ a]\!]\delta$
Pick $W'' = W' +\!\!+ \{[\ell \mapsto \hat{v}] \mid \vdash \hat{v} : a\}$.	
	$W'' \sqsupseteq W'$
Trivial.	
	$h' : W''$
This follows from $h : W'$, our assumption on v, and Lemma 60.	
	$(j - 1, W'', \ell) \in \mathcal{V}[\![\mathsf{ref}\ a]\!]\delta$
By definition of $\mathcal{V}[\![\mathsf{ref}\ a]\!]$.	

□

Exercise 35 Prove compatibility for first-order dereferencing.

Lemma 67 (Compatibility for first-order dereferencing)**.**

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

•

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.

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

Lemma 68 (World Extension is a pre-order)**.** $\sqsupseteq$ *is reflexive and transitive.*

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

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

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

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

Exercise 36 (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 compatibility lemmas for the cases for first-order references: $\mathsf{new}\ e$, $^*e$, $e_1 \leftarrow e_2$.

•

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

Lemma 69. *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:
	$(\_, W, e) \in \mathcal{E}[\![(\mathbf{1} \to \mathbf{1}) \to \mathbf{1}]\!]$
$h : W$	
$h\,;e \searrow^{-} h_0\,;e_0$	
	$\exists W_0 \sqsupseteq W.\ h_0 : W_0 \wedge (\_, W_0, e_0) \in \mathcal{V}[\![(\mathbf{1} \to \mathbf{1}) \to \mathbf{1}]\!]$
$\ell \notin \mathrm{dom}(h)$	
$h_0 = h \uplus [\ell \mapsto \overline{3}]$	
$e_0 = \lambda f.\ \ell \leftarrow \overline{4}\,;f\ ()\,;\mathsf{assert}\ (*\ell == \overline{4})$	
Pick W_0 that extends W with a new invariant i:	
$W_0[i] := \{\ \Phi := \{(s_3, s_4)\}^\star, \mathrm{c} := s_3, \mathrm{H} := \lambda s_n.\ \{[\ell \mapsto n]\}\ \}$	
	$W_0 \sqsupseteq W$ (trivial)
	$h_0 : W_0$ (done by $[\ell \mapsto \overline{3}] \in W_0[i].\mathrm{H}(s_3)$)
	$(\_, W_0, e_0) \in \mathcal{V}[\![(\mathbf{1} \to \mathbf{1}) \to \mathbf{1}]\!]$
$W_1 \sqsupseteq W_0$	
$(\_, W_1, v) \in \mathcal{V}[\![\mathbf{1} \to \mathbf{1}]\!]$	
Let $e_1 := \ell \leftarrow \overline{4}\,;v\ ()\,;\mathsf{assert}\ (*\ell == \overline{4})$	$(\_, W_1, e_1) \in \mathcal{E}[\![\mathbf{1}]\!]$
$h_1 : W_1$	
$h_1\,;e_1 \searrow^{-} h_2\,;e_2$	
	$\exists W_f \sqsupseteq W_1.\ h_2 : W_f \wedge (\_, W_f, e_2) \in \mathcal{V}[\![\mathbf{1}]\!]$
In order to pick W_f, we need to know what h_2 and e_2 can be,	
so we need to symbolically execute e_1.	
From $W_1 \sqsupseteq W_0$, $W_1[i] = W_0[i]$ except that $W_1[i].\mathrm{c}$ could be s_4.	
From $h_1 : W_1$ and the invariant i, $\ell \in \mathrm{dom}(h_1)$.	
So $h_1\,;e_1 \rightsquigarrow^\star h_1[\ell \mapsto \overline{4}]\,;(v\ ()\,;\mathsf{assert}\ (*\ell == \overline{4}))$	
and (i) $h_1[\ell \mapsto \overline{4}]\,;(v\ ()\,;\mathsf{assert}\ (*\ell == \overline{4})) \searrow^{-} h_2\,;e_2$.	
Now, we need to execute $v\ ()\,;\mathsf{assert}\ (*\ell == \overline{4})$.	
For that, we need to come up with a new world.	
Let $W_2[i] = W_1[i]$ with $\mathrm{c} := s_4$	
(ii) $h_1[\ell \mapsto \overline{4}] : W_2$	
By Lemma 70, along with (i) and (ii),	
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 $\sqsupseteq$, $W_3 \sqsupseteq W_1$.	$\square$

Lemma 70 (Auxiliary "Lemma")**.** $(\_, W_2, (v\ ()\,;\mathsf{assert}\ (*\ell == \overline{4}))) \in \mathcal{E}[\![\mathbf{1}]\!]$

Proof.

We have:	To show:
From $(\_, W_1, v) \in \mathcal{V}[\![\mathbf{1} \to \mathbf{1}]\!]$, by monotonicity $(\_, W_2, v) \in \mathcal{V}[\![\mathbf{1} \to \mathbf{1}]\!]$	$(\_, W_2, (v\ ()\ ;\ \mathsf{assert}\ (*\ell == \overline{4}))) \in \mathcal{E}[\![\mathbf{1}]\!]$
By assumption, and def. of $\mathcal{V}[\![\mathbf{1} \to \mathbf{1}]\!]$, $(\_, W_2, v\ ()) \in \mathcal{E}[\![\mathbf{1}]\!]$ We apply Lemma 63 (the bind lemma) with $K := \bullet\ ;\ \mathsf{assert}\ (*\ell == \overline{4})$	$\forall W_4 \sqsupseteq W_2.\ (\_, W_4, \mathsf{assert}\ *\ell == \overline{4}) \in \mathcal{E}[\![\mathbf{1}]\!]$
$W_4 \sqsupseteq W_2$ $h_4 : W_4$ $h_4\ ;\ \mathsf{assert}\ (*\ell == \overline{4}) \searrow^{-} h_5\ ;\ e_5$	$\exists W_5 \sqsupseteq W_4.\ h_5 : W_5 \wedge (\_, W_5, e_5) \in \mathcal{V}[\![\mathbf{1}]\!]$
$h_5(\ell) = \overline{4}$ because $W_4 \sqsupseteq W_2$ and $W_2[i].\mathrm{c} = s_4$ $h_5 = h_4 \wedge e_5 = ()$	$\exists W_5 \sqsupseteq W_4.\ h_4 : W_5 \wedge (\_, W_5, ()) \in \mathcal{V}[\![\mathbf{1}]\!]$
Trivial by picking $W_5 := W_4$.	$\square$

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.

$$\begin{aligned}
\mathrm{SYMBOL} &:= \exists \alpha.\ \{\ \mathsf{mkSym} : \mathbf{1} \to \alpha,\\
&\qquad\qquad\quad \mathsf{check} : \alpha \to \mathbf{1}\ \}\\
\mathrm{Symbol} &:= \mathsf{let}\ c = \mathsf{new}\ \overline{0}\ \mathsf{in}\\
&\qquad \mathsf{pack}\ \left\langle \mathsf{int}, \begin{array}{r}\{\ \mathsf{mkSym} := \lambda_.\ \mathsf{let}\ x = *c\ \mathsf{in}\ c \leftarrow x + 1\ ;\ x,\\ \mathsf{check} := \lambda x.\ \mathsf{assert}\ (x < *c)\ \}\end{array} \right\rangle \mathsf{as}\ \mathrm{SYMBOL}
\end{aligned}$$

Intuitively, the function check guarantees that only mkSym can generate values of type α.

The proof of safety for Symbol showcases how semantic types and invariants can work together in interesting ways. Instead of going through the proof, we give the invariant that will be associated with the location ℓ_c correspding to the reference c, and the semantic type for the type variable α.

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

Note that the semantic type assigned to α is defined in terms of the transition system that governs ℓ_c. Consequently, the semantic type will grow over time as the value in ℓ_c increases. This is possible because when we define the interpretation of α, we already picked the new

world, and hence we know which index our island will have. This is similar to how, when we define the island, we already know the actual location ℓ_c picked by the program.

In the proof of safety of Symbol, we know that both mkSym and check will only be called in worlds future to the one in which we established our invariant. Consequently, we can rely on the existence of the transition we set up. The transitions we take mirror the change to c in the program.

Proof Sketches. We refer to the above as a *proof sketch*. In such a sketch, we only give the invariants that are picked, the interpretations of semantic types, and how we update the state of STSs.

4.7.2 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.

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

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 37 (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:

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

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

Exercise 38 (In F^{μ}* + STSs) Consider the following code:

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

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:

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

Give a proof *outline*, not a proof sketch (following the conventions described previously, in which step-indices are ignored). •

Exercise 39 (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

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

Give a proof *outline*, not a proof sketch (following the conventions described previously, in which step-indices are ignored). •

4.8 Semantically well-typed expressions are safe

The following theorem tells us that, in order to to prove that a closed expression is safe—or *kosher* in our old tongue, it is *adequate* to show that the expression is semantically well-typed.

Theorem 71 (Adequacy)**.**

If $\vDash e : A$ *then*
for all h, e', h' *such that* $h\,;e \rightsquigarrow^\star e'\,;h'$*, either* e' *is a value or* $h'\,;e'$ *can make a step.*

Proof.

We have:	To show:
(i) $\vDash e : A$	
$h\,;e \rightsquigarrow^\star e'\,;h'$	e' is a value or $h'\,;e'$ can make a step
Suppose $h'\,;e'$ cannot take a step any more.	
	e' is a value
(Here we exploit the fact that the reduction relation $\rightsquigarrow$ is decidable.)	
So $h\,;e \searrow^j h'\,;e'$.	
From (i), have $\forall k, W.\ (k, W, e) \in \mathcal{E}[\![A]\!]$.	
We instantiate this with $k := j + 1$ and $W := [\,]$.	
It is trivial to see that $h : W$.	
So we have some W' s.t. $W' \sqsupseteq W \wedge h' : W' \wedge (1, W', e') \in \mathcal{V}[\![A]\!]$.	
Thus e' is a value.	□

5 Contextual Equivalence and Relational Parametricity

We revisit the problem in Section 2.4 of showing that our Church encodings are *full* and *faithful* encodings. They are full in the sense that the encodings include all the canonical forms we expect as elements of the type, and they are faithful in the sense that the encodings do not include those that don't behave like one of the canonical forms.

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

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

First, we introduce program contexts, which only concern with composing expressions and not with the evaluation strategy like evaluation contexts.

$$\text{Program Context} \quad C ::= \bullet \mid C\ e \mid e\ C \mid \lambda x.\ C \mid \Lambda\alpha.\ C \mid C\ \langle A \rangle$$

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

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

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

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

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

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

The judgement $C : (\Delta\,;\Gamma \vdash A) \rightsquigarrow (\Delta'\,;\Gamma' \vdash A')$ essentially says that if $\Delta\,;\Gamma \vdash e : A$, then $\Delta'\,;\Gamma' \vdash C[e] : A'$.

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

$$\begin{aligned}\Delta\,;\Gamma \vdash & e_1 \equiv_{\text{ctx}} e_2 : A \\ & := \Delta\,;\Gamma \vdash e_1 : A \wedge \Delta\,;\Gamma \vdash e_2 : A \\ & \quad \wedge \forall C : (\Delta\,;\Gamma \vdash A) \rightsquigarrow (\emptyset\,;\emptyset \vdash \mathsf{bool}).\ C[e_1] \Downarrow \mathsf{true} \iff C[e_2] \Downarrow \mathsf{true}\end{aligned}$$

where

$$e \Downarrow b := \exists v.\ e \downarrow v \wedge \eta(v) \downarrow b$$
$$\eta(v) := \mathsf{if}\ v\ \mathsf{then\ true\ else\ false}$$

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

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

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

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

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

$$\begin{aligned}
\mathrm{VRel} &:= \mathbb{P}(CVal \times CVal)\\
\mathcal{V}[\![\alpha]\!]\delta &:= \delta(\alpha)\\
\mathcal{V}[\![\mathsf{int}]\!]\delta &:= \{(\overline{n},\overline{n})\}\\
\mathcal{V}[\![A \to B]\!]\delta &:= \left\{(v_1,v_2) \;\middle|\; \forall(v_1',v_2') \in \mathcal{V}[\![A]\!]\delta.\ (v_1\ v_1', v_2\ v_2') \in \mathcal{E}[\![B]\!]\delta\right\}\\
\mathcal{V}[\![\forall\alpha.\ A]\!]\delta &:= \{(v_1,v_2) \mid \forall R \in \mathrm{VRel}.\ (v_1\ \langle\rangle, v_2\ \langle\rangle) \in \mathcal{E}[\![A]\!](\delta, \alpha \mapsto R)\}\\
\mathcal{V}[\![\exists\alpha.\ A]\!]\delta &:= \{(\mathsf{pack}\ v_1, \mathsf{pack}\ v_2) \mid \exists R \in \mathrm{VRel}.\ (v_1,v_2) \in \mathcal{V}[\![A]\!](\delta, \alpha \mapsto R)\}
\end{aligned}$$

Theorem 72 (Soundness of $\approx$ w.r.t. $\equiv_{\mathrm{ctx}}$, or $\approx\ \subseteq\ \equiv_{\mathrm{ctx}}$)**.**

If $\Delta\,;\Gamma \vdash e_1 : A \wedge \Delta\,;\Gamma \vdash e_2 : A \wedge \Delta\,;\Gamma \vdash e_1 \approx e_2 : A$,
then $\Delta\,;\Gamma \vdash e_1 \equiv_{\mathrm{ctx}} e_2 : A$.

Proof. Suppose $C : (\Delta\,;\Gamma \vdash A) \rightsquigarrow (\emptyset\,;\emptyset \vdash \mathsf{bool})$.
By compatibility (Lemma 73), $\emptyset\,;\emptyset \vdash C[e_1] \approx C[e_2] : \mathsf{bool}$.
By adequacy (Lemma 74), we are done. □

Lemma 73 (Compatibility)**.**

If $\Delta\,;\Gamma \vdash e_1 : A$ *and* $\Delta\,;\Gamma \vdash e_2 : A$
and $\Delta\,;\Gamma \vdash e_1 \approx e_2 : A$
and $C : (\Delta\,;\Gamma \vdash A) \rightsquigarrow (\Delta'\,;\Gamma' \vdash A')$,

then $\Delta'\,;\Gamma' \vdash C[e_1] \approx C[e_2] : A'$
and $\Delta'\,;\Gamma' \vdash C[e_1] : A'$
and $\Delta'\,;\Gamma' \vdash C[e_2] : A'$.

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

Lemma 74 (Adequacy)**.**

If $e_1, e_2 : \mathsf{bool}$ *and* $(e_1,e_2) \in \mathcal{E}[\![\mathsf{bool}]\!]$,
then $e_1 \Downarrow \mathsf{true} \iff e_2 \Downarrow \mathsf{true}$.

Proof. $e_1 \downarrow v_1, e_2 \downarrow v_2, (v_1,v_2) \in \mathcal{V}[\![\forall\alpha.\ \alpha \to \alpha \to \alpha]\!]$.
To show $(v_1\ \langle\rangle\ \mathsf{true}\ \mathsf{false}, v_2\ \langle\rangle\ \mathsf{true}\ \mathsf{false}) \in \mathcal{E}[\![\alpha]\!](\alpha \mapsto R)$.
Pick $R := \{(\mathsf{true},\mathsf{true}),(\mathsf{false},\mathsf{false})\} \in \mathrm{VRel}$. □

Example 75 (Representation Independence)**.**

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

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

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

Theorem 76 (Fundamental Property of the Logical Relations)**.**

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

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

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

Theorem 77. *If* $\vdash e : \mathsf{bool}$*, then* $\vdash e \equiv_{\text{ctx}} \eta(e) : \mathsf{bool}$.

Proof.

We have:	To show:
$\vdash e : \mathsf{bool}$	$\vdash e \equiv_{\text{ctx}} \eta(e) : \mathsf{bool}$
	By soundness (Theorem 72) $\vdash e \approx \eta(e) : \mathsf{bool}$
	$(e, \mathsf{if}\ e\ \mathsf{then\ true\ else\ false} \in \mathcal{E}[\![\mathsf{bool}]\!])$
By the fundamental property (Theorem 76) and our assumption,	
$(e, e) \in \mathcal{E}[\![\mathsf{bool}]\!]$	
So $e \downarrow v$, and (i) $(v, v) \in \mathcal{V}[\![\mathsf{bool}]\!]$.	
Also $\mathsf{if}\ e\ \mathsf{then\ true\ else\ false} \downarrow b \in \{\mathsf{true}, \mathsf{false}\}$.	
	$(v, b) \in \mathcal{V}[\![\mathsf{bool}]\!]$
	$(v, b) \in \mathcal{V}[\![\forall\alpha.\ \alpha \to \alpha \to \alpha]\!]$
Suppose $R \in \text{VRel}, (v_1, v_2) \in R, (v_1', v_2') \in R$.	
	$(v\ \langle\rangle\ v_1\ v_1', b\ \langle\rangle\ v_2\ v_2') \in \mathcal{E}[\![R]\!]$
	It suffices to show $(v\ \langle\rangle\ v_1\ v_1', (\mathsf{if}\ v\ \mathsf{then\ true\ else\ false})\ \langle\rangle\ v_2\ v_2') \in \mathcal{E}[\![R]\!]$
	$(v\ \langle\rangle\ v_1\ v_1', (v\ \langle\rangle\ \mathsf{true}\ \mathsf{false})\ \langle\rangle\ v_2\ v_2') \in \mathcal{E}[\![R]\!]$
Pick $S := \{(v_a, v_b) \mid (v_a, v_b\ \langle\rangle\ v_2\ v_2') \in \mathcal{E}[\![R]\!]\} \in \text{VRel}$.	
	It suffices to show $(v\ \langle\rangle\ v_1\ v_1', v\ \langle\rangle\ \mathsf{true}\ \mathsf{false}) \in \mathcal{E}[\![S]\!]$
We apply (i).	
	$(v_1, \mathsf{true}) \in \mathcal{V}[\![S]\!]$
Follows from the definition of S and that $(v_1, v_2) \in R$.	
	$(v_2, \mathsf{false}) \in \mathcal{V}[\![S]\!]$
Follows from the definition of S and that $(v_1', v_2') \in R$.	

□

Exercise 40 Prove that our encodings of natural numbers in Section 2.4 are full and faithful. That is, show that if $\vdash n : \mathsf{nat}$, then $\vdash n \equiv_{\text{ctx}} n\ \langle\mathsf{nat}\rangle\ \mathsf{zero}\ \mathsf{succ} : \mathsf{nat}$. •