

Formal Foundations of Programming Languages

Lecture Notes

Ralf Jung Max Vistруп

ETH Zurich

based on
Semantics of Type Systems
Derek Dreyer, Simon Spies, Lennard Gäher, Ralf Jung,
Jan-Oliver Kaiser, Hoang-Hai Dang, David Swasey, Jan Menz

This work is licensed under a [Creative Commons “Attribution 4.0 International”](https://creativecommons.org/licenses/by/4.0/) license.



Contents

1	Simply Typed Lambda Calculus	2
1.1	Operational Semantics	3
1.2	The Untyped λ -calculus	5
1.3	Typing	7
1.4	Type Safety	9
1.5	Termination	14
2	De Bruijn representation	17
2.1	Substitution	18
2.2	Operational Semantics	20
2.3	Type System	20
2.4	Type Safety	20
2.5	Termination	21
3	System F: Polymorphism and Existential Types	23
3.1	System F	23
3.2	Encoding Data Types	24
3.3	Data Abstraction with Existential Types	27
3.4	System F with De Bruijn representation	29
3.5	Type Safety	30
3.6	Termination	32
3.7	Type Casts Break Termination	35
3.8	Free Theorems	36
3.9	Semantic Type Safety of Unsafe Code with Existential Types	38
4	Mutable State	41
4.1	Data Abstraction via Local State	42
4.2	Type Safety	43
4.3	Recursion via state	46

1 Simply Typed Lambda Calculus

The terms and values of the simply-typed lambda calculus are defined as follows:

Variables	$x, y \quad \dots$
Runtime Terms	$e ::= x \mid \lambda x. e \mid e_1 e_2 \mid e_1 + e_2 \mid \bar{n}$
(Runtime) Values	$v ::= \lambda x. e \mid \bar{n}$

We call these “runtime” terms since they are the terms on which we will define the runtime semantics, *i.e.*, the operational semantics.

Some notes on operator precedence and syntactic conventions: λ binds very weakly, so *e.g.*, $\lambda x. \lambda y. x y$ is interpreted as $\lambda x. \lambda y. (x y)$. Application is left-associative, so $x y z$ is interpreted as $(x y) z$. We use $\lambda x, y. e$ as a short-hand for $\lambda x. \lambda y. e$.

Variables and Substitution In the simply typed λ -calculus, variables and substitution are instrumental to define how terms are evaluated (called the *operational semantics*). During evaluation, if we apply a λ -abstraction $\lambda x. e$ to a value v , then the variable x is *substituted* with the value v in e . For example, the term $(\lambda x. x + \bar{1}) \bar{4}\bar{1}$ proceeds with $\bar{4}\bar{1} + \bar{1}$ in the next step of the evaluation. This step is commonly called “ β -reduction” (“beta-reduction”).

We write $e[e'/x]$ for the substitution operation that replaces x with e' in e , and we pronounce this “ e with e' for x ”. We define it recursively by:

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

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

$$\begin{aligned}
 \text{fv}(x) &:= \{x\} \\
 \text{fv}(\lambda x. e) &:= \text{fv}(e) \setminus \{x\} \\
 \text{fv}(e_1 e_2) &:= \text{fv}(e_1) \cup \text{fv}(e_2) \\
 \text{fv}(e_1 + e_2) &:= \text{fv}(e_1) \cup \text{fv}(e_2) \\
 \text{fv}(\bar{n}) &:= \emptyset
 \end{aligned}$$

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

To avoid variable capturing, a correct substitution renames bound variables where conflicts arise. For example, $(\lambda x. y + x)[\bar{2} + x/y]$ should result in $\lambda z. (\bar{2} + x) + z$, such that x remains free. Unfortunately, defining (and reasoning about) a substitution operation that properly renames bound variables is

oftentimes tedious, especially in proof assistants. Thus, on paper, it is standard to assume and follow *Barendregt’s variable convention* [1]: all bound and free variables are distinct and this invariant is maintained implicitly.

Since we *mechanize* our proofs in Coq, we cannot assume Barendregt’s variable convention. Instead, we use the (slightly broken) substitution operation above. In our use cases, the substitution will only insert *closed* terms (*i.e.*, terms without any free variables: $\text{fv}(e) = \emptyset$), which avoids the problem of variable capture entirely. (In later sections, we will discuss the more complicated DeBruijn representation of terms with binders, which simplifies defining a correct, capture-avoiding substitution.)

1.1 Operational Semantics

In order to reason about programs, we have to assign a semantics to them. In the following, we assign an *operational semantics* to programs, meaning we describe how runtime terms are evaluated. We distinguish three different operational semantics: *big-step semantics*, *structural semantics*, and *contextual semantics*. They are all equivalent but distinct ways of expressing how our terms compute.

Big-Step Semantics

$$\boxed{e \Downarrow v}$$

The big-step semantics are arguably the “most natural” semantics: they directly define when a term e computes to a value v .

$$\begin{array}{lll} \text{LITERAL} & \text{LAMBDA} & \text{APP} \\ \hline \bar{n} \Downarrow \bar{n} & \lambda x. e \Downarrow \lambda x. e & \frac{e_1 \Downarrow \lambda x. e \quad e_2 \Downarrow v_2 \quad e[v_2/x] \Downarrow v}{e_1 e_2 \Downarrow v} \\ & & \text{PLUS} \\ & & \frac{e_1 \Downarrow \bar{n}_1 \quad e_2 \Downarrow \bar{n}_2}{e_1 + e_2 \Downarrow \bar{n}_1 + \bar{n}_2} \end{array}$$

Small-step semantics

Whereas a big-step semantics takes a single “big step” directly from a term to its final value, a small-step semantics defines a step-by-step process whereby the term is slowly reduced further and further, until it eventually reaches a value. Small-step semantics look more complicated at first, but they are needed when considering infinite executions, or to model concurrency (where the individual steps of multiple computations can interleave).

Small-step semantics come in two flavors, “structural” and “contextual”. We will discuss structural semantics (often called SOS, “Structured Operational Semantics”) first.

Structural Semantics

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

We define a structural *call-by-value*, *weak*, *right-to-left* operational semantics on our runtime terms:

$$\begin{array}{lll} \text{APP-STRUCT-R} & \text{APP-STRUCT-L} & \text{BETA} \\ \hline \frac{e_2 \succ e'_2}{e_1 e_2 \succ e_1 e'_2} & \frac{e_1 \succ e'_1}{e_1 v_2 \succ e'_1 v_2} & (\lambda x. e) v \succ e[v/x] \\ \\ \text{PLUS-STRUCT-R} & \text{PLUS-STRUCT-L} & \text{PLUS} \\ \hline \frac{e_2 \succ e'_2}{e_1 + e_2 \succ e_1 + e'_2} & \frac{e_1 \succ e'_1}{e_1 + v_2 \succ e'_1 + v_2} & \bar{n} + \bar{m} \succ \bar{n} + \bar{m} \end{array}$$

The three adjectives have the following meaning:

- *Right-to-left*: in a term like $e_1 + e_2$, we evaluate e_2 to a value before we even begin evaluating e_1 . This is reflected in **PLUS-STRUCT-L** by requiring the right-hand operand to be a value for the rule to apply.

- *Call-by-value*: we evaluate the argument of a function to a value before doing substitution. This is reflected in **BETA** requiring the argument to be a value, and in **APP-STRUCT-R** letting us evaluate the argument *before* doing β -reduction.
- *Weak*: we do *not* allow reduction below λ abstractions. This is reflected by *not* having a structural rule for $\lambda x. e$. In contrast, a *strong* semantics would let us reduce $\lambda x. (\lambda y. y) x$ to $\lambda x. x$. However, given that our substitution operation is not capture-avoiding, we cannot allow strong reduction in our calculus.

Contextual Semantics

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

For the contextual semantics, we first define evaluation contexts:

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

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

Context Filling

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

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

Base reduction

$$\boxed{e_1 \rightsquigarrow_b e_2}$$

$$\begin{array}{ll} \text{BETA} & \text{PLUS} \\ (\lambda x. e) v \rightsquigarrow_b e[v/x] & \overline{n} + \overline{m} \rightsquigarrow_b \overline{n + m} \end{array}$$

Contextual reduction

$$\boxed{e_1 \rightsquigarrow e_2}$$

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

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

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

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

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

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

Proof Sketch. By induction on K_1 .

One example for a reasoning principle that can be stated more elegantly in the contextual semantics than the structural semantics is context lifting on \rightsquigarrow^* : if $e_1 \rightsquigarrow^* e_2$, then $K[e_1] \rightsquigarrow^* K[e_2]$. To even express the same result in the structural semantics, we'd have to state four lemmas: one for each structural rule. Having a notion of “evaluation contexts” gives us some extra vocabulary that makes these kinds of properties a lot easier to work with.

1.2 The Untyped λ -calculus

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

To get started, let us write a term that causes an infinite reduction.

$$\omega := \lambda x. x x \qquad \Omega := \omega \omega$$

ω applies its argument to itself, and Ω applies ω to itself. Let us try out what happens when we evaluate Ω .

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

As it turns out, Ω reduces to itself! Thus, there are terms in the untyped lambda calculus such as Ω which *diverge* (*i.e.*, their reduction chains do not terminate). Note that we had to use the small-step operational semantics for this; big-step semantics can only talk about terminating executions.

Scott encodings We can not only write diverging terms, but we can also encode inductive data types in the untyped lambda calculus. For example, we can encode natural numbers in their Peano representation (*i.e.*, with the constructors 0 and S) as lambda terms. The basic idea is to interpret natural numbers as “case distinctions”. That is, each number will be an abstraction with two arguments, z and s , the “zero” and “successor” cases. If the number is 0, then it will return the argument z . If the number is $S n$, then it will return $s n$, *i.e.*, it will call x applied to the predecessor n . You should think of $n z s$ as being equivalent to the following Coq code:

```

match n with
| 0 => z
| S n => s n
end

```

Now we can define our first numbers. We will write $\tilde{0}$ for the Scott encoding of 0, and so on:

$$\tilde{0} := \lambda z, s. z \qquad \tilde{1} := \lambda z, s. s \tilde{0} \qquad \tilde{2} := \lambda z, s. s \tilde{1}$$

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

$$\begin{aligned}\tilde{0} &:= \lambda z, s. z \\ \tilde{S}n &:= \lambda z, s. s \tilde{n}\end{aligned}$$

Note that \tilde{n} is a value of each n , *i.e.*, it does not reduce any further as a lambda term.

We can also define the successor operation as a function in the calculus:

$$\tilde{S} := \lambda n. \lambda z, s. s n$$

Convince yourself that $\tilde{S} \tilde{1}$ reduces to $\tilde{2}$, and in general $\tilde{S} \tilde{n}$ reduces to $\tilde{S}n$.

We can use this representation, known as the Scott encoding, to compute with natural numbers. That is, since we can do a case analysis on the numbers *by definition*, we can distinguish them and compute different results depending on the case. For example, the function $\mathbf{pred} := \lambda n. n \tilde{0} (\lambda n'. n')$ computes the predecessor of a Scott encoded natural number.

If we want to define slightly more interesting functions on our numbers such as addition $\mathbf{add} \ n \ m$, we run into a problem. The naive definition of multiplication by 2 would be:

$$\mathbf{mul2} := \lambda n. n \tilde{0} (\lambda n'. \tilde{S}(\tilde{S}(\mathbf{mul2} \ n')))$$

Remember that to read (and write) the definitions, it is helpful to turn them into Coq `match` statements as explained above.

Unfortunately, this definition is broken! The term that we want to define, $\mathbf{mul2}$, occurs in its own definition on the right hand side. To fix this problem, we are now going to develop a mechanism to add recursion to the untyped lambda calculus.

Recursion To enable recursion, we define a recursion operator \mathbf{fix} (sometimes called fixpoint combinator or Y -combinator). The idea of \mathbf{fix} is that given a template $F := \lambda rec, x. e$ of the recursive function that we want to define, where rec is a placeholder for recursive calls, the expression $\mathbf{fix} \ F \ v$ reduces to $F (\mathbf{fix} \ F) \ v$. In other words, we end up calling F such that rec becomes $\mathbf{fix} \ F$ (and the value v is just forwarded).

In other words, the desired reduction behavior of \mathbf{fix} is

$$\mathbf{fix} \ F \ v \succ^* F (\mathbf{fix} \ F) \ v$$

We will define \mathbf{fix} below. Beforehand, let us explore how we can define recursive functions such as $\mathbf{mul2}$ with \mathbf{fix} . We first define the template $\mathbf{MUL2}$, and then we take the fixpoint to obtain $\mathbf{mul2}$:

$$\begin{aligned}\mathbf{MUL2} &:= \lambda rec, \lambda n. n \tilde{0} (\lambda n'. \tilde{S}(\tilde{S}(rec \ n')))) \\ \mathbf{mul2} &:= \mathbf{fix} \ \mathbf{MUL2}\end{aligned}$$

Why does this make sense? Let us consider what happens when reducing $\mathbf{mul2} \ \tilde{1}$:

$$\begin{aligned}\mathbf{mul2} \ \tilde{1} &= \mathbf{fix} \ \mathbf{MUL2} \ \tilde{1} && \text{by the definition of } \mathbf{mul2} \\ \gamma^* \mathbf{MUL2} (\mathbf{fix} \ \mathbf{MUL2}) \ \tilde{1} &&& \text{by the fix reduction rule} \\ &= \mathbf{MUL2} \ \mathbf{mul2} \ \tilde{1} && \text{by the definition of } \mathbf{mul2} \\ \gamma^* \tilde{1} \tilde{0} (\lambda n'. \tilde{S}(\tilde{S}(\mathbf{mul2} \ n'))) &&& \text{by the definition of } \mathbf{MUL2} \text{ and two } \beta\text{-reductions} \\ \gamma^* (\lambda n'. \tilde{S}(\tilde{S}(\mathbf{mul2} \ n'))) \tilde{0} &&& \text{by the definition of } \tilde{1} \text{ and two } \beta\text{-reductions} \\ \gamma \tilde{S}(\tilde{S}(\mathbf{mul2} \ \tilde{0})) &&& \text{by another } \beta\text{-reduction} \\ \gamma^* \tilde{S}(\tilde{S}(\mathbf{MUL2} \ \mathbf{mul2} \ \tilde{0})) &&& \text{by the fix rule and the definition of } \mathbf{mul2} \\ \gamma^* \tilde{S}(\tilde{S}(\tilde{0} \tilde{0} (\lambda n'. \tilde{S}(\tilde{S}(\mathbf{mul2} \ n'))))) &&& \text{by the definition of } \mathbf{MUL2} \text{ and two } \beta\text{-reductions} \\ \gamma^* \tilde{S}(\tilde{S}(\tilde{0})) &&& \text{by the definition of } \tilde{0} \text{ and two } \beta\text{-reductions} \\ \gamma^* \tilde{2} &&& \text{by the definition of } \tilde{S} \text{ and two } \beta\text{-reductions}\end{aligned}$$

In fact, we can prove that `mul2` has the desired computational behavior:

Lemma 3. $\text{mul2 } \tilde{n} \succ^* \widetilde{2 * n}$

The proof is simple but tedious; the key point is showing that `mul2` satisfies $\text{mul2 } \tilde{0} \succ^* \tilde{0}$ and $\text{mul2 } \tilde{S} n \succ^* \tilde{S} (\tilde{S} (\text{mul2 } \tilde{n}))$. Then an induction on n shows the desired result.

Defining the fixpoint combinator To define `fix`, we will abstract over the template and assume it is some abstract value F . The definition of `fix` F (where `fix` is parametric in F) consists of two parts:

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

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

With this definition, `fix` F has the right reduction behavior:

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

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

1.3 Typing

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

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

We will write $A \rightarrow B \rightarrow C$ for $A \rightarrow (B \rightarrow C)$.

Note that this type system only has two kinds of types: integers and functions. This means all functions are between integers ($\text{int} \rightarrow \text{int}$) or functions on integers (like $(\text{int} \rightarrow \text{int}) \rightarrow \text{int}$).

Church-style typing

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

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

$$\begin{array}{c} \text{VAR} \\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \end{array} \quad \begin{array}{c} \text{LAM} \\ \frac{\Gamma, x : A \vdash E : B}{\Gamma \vdash \lambda x : A. E : A \rightarrow B} \end{array} \quad \begin{array}{c} \text{APP} \\ \frac{\Gamma \vdash E_1 : A \rightarrow B \quad \Gamma \vdash E_2 : A}{\Gamma \vdash E_1 E_2 : B} \end{array}$$

$$\begin{array}{c} \text{PLUS} \\ \frac{\Gamma \vdash E_1 : \text{int} \quad \Gamma \vdash E_2 : \text{int}}{\Gamma \vdash E_1 + E_2 : \text{int}} \end{array} \quad \begin{array}{c} \text{INT} \\ \Gamma \vdash \bar{n} : \text{int} \end{array}$$

Curry-style typing

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

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

$$\begin{array}{c} \text{VAR} \\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \end{array} \quad \begin{array}{c} \text{LAM} \\ \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x. e : A \rightarrow B} \end{array} \quad \begin{array}{c} \text{APP} \\ \frac{\Gamma \vdash e_1 : A \rightarrow B \quad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1 e_2 : B} \end{array}$$

$$\begin{array}{c} \text{PLUS} \\ \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \end{array} \quad \begin{array}{c} \text{INT} \\ \Gamma \vdash \bar{n} : \text{int} \end{array}$$

Context lookup

$$\boxed{x : A \in \Gamma}$$

To be fully precise we still have to define lookup in a variable context.

$$\begin{array}{c} \text{HERE} \\ x : A \in \Gamma, x : A \end{array} \quad \begin{array}{c} \text{THERE} \\ \frac{x : A \in \Gamma \quad x \neq y}{x : A \in \Gamma, y : B} \end{array}$$

This means lookup is right-to-left and only the first assignment with a given variable name counts, *i.e.*, $x : \text{int} \in \emptyset, x : \text{int}, x : \text{int} \rightarrow \text{int}$ does *not* hold.

General induction on typing We have already seen a few examples of inductively defined predicates like the typing judgments above, and examples of how to do induction on them. There will be a lot more to come. Let us briefly consider how induction on the judgment $\Gamma \vdash e : A$ works *in general*.

Induction can be performed when our goal is to prove a statement of the form:

$$\forall \Gamma, e, A. \Gamma \vdash e : A \Rightarrow IH(\Gamma, e, A)$$

Here, IH is the induction statement, *i.e.*, the proof we want to extract from the inductively defined predicate. It is often a good idea to state IH explicitly, and we will be doing it for most of our sample solutions.

When we now do induction on $\Gamma \vdash e : A$, we have to consider five cases, one for each typing rule. In each case we learn things about Γ , e , and A : they have to be of the right shape for the rule to apply. We also learn all the assumptions that were required to prove the rule. And if the rule has as an assumption a recursive occurrence of the typing judgment itself, we furthermore learn that the corresponding induction statement holds. In our proof notation, this looks as follows:

- **VAR**: $e = x$.
Have: $x : A \in \Gamma$. To Show: $IH(\Gamma, x, A)$.
(We will usually unfold and simplify IH in the “To Show”. If IH is an implication $P \Rightarrow Q$, then we will already list its left-hand side P as something we “Have” and only have the right-hand side Q as what is left “To Show”.)
- **LAM**: $e = \lambda x. e'$, $A = A' \rightarrow B'$.
Have: $\Gamma, x : A' \vdash e' : B'$ and $IH(\Gamma, x : A', e', B')$. To Show: $IH(\Gamma, \lambda x. e', A' \rightarrow B')$.
(The first assumption comes directly from the **LAM** rule, the second one is the corresponding induction hypothesis. Note how it uses exactly the same context, term, and type as the first assumption. We often do not need the first assumption and omit it.)
- **APP**: $e = e_1 e_2$, $A = B'$.
Have: $\Gamma \vdash e_1 : A' \rightarrow B'$ and $IH(\Gamma, e_1, A' \rightarrow B')$ and $\Gamma \vdash e_2 : A'$ and $IH(\Gamma, e_2, A')$.
To Show: $IH(\Gamma, e_1 e_2, B')$.
(This time we have two assumptions from the rule, and each of them has a corresponding induction hypothesis.)

- **PLUS:** $e = e_1 + e_2$, $A = \text{int}$.
 Have: $\Gamma \vdash e_1 : \text{int}$ and $IH(\Gamma, e_1, \text{int})$ and $\Gamma \vdash e_2 : \text{int}$ and $IH(\Gamma, e_2, \text{int})$.
 To Show: $IH(\Gamma, e_1 + e_2, \text{int})$.
 (This is structurally very similar to the previous case.)
- **INT:** $e = \bar{n}$, $A = \text{int}$.
 To Show: $IH(\Gamma, \bar{n}, \text{int})$.
 (There is no assumption, since this typing rule has no premises.)

Induction on other inductively defined predicates follows exactly the same pattern.

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

 Erase(\cdot)

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

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

1.4 Type Safety

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

Statement 5 (Type Safety). *If $\vdash e : A$, then e is safe.*

To define the notion of safety, we introduce the concepts of *reducible terms* and *progressive terms*.

Definition 6 (Reducible Terms). *A (runtime) term e is reducible if there exists e' s.t. $e \rightsquigarrow e'$.*

Definition 7 (Progressive Terms). *A (runtime) term e is progressive if either it is a value or it is reducible.*

Definition 8 (Progressive Terms). *A (runtime) term e is safe if for all e' s.t. $e \rightsquigarrow^* e'$, the term e' is progressive.*

Progressive terms are “well-behaved” in a certain sense: either they are already a value, or they can make a step, which hopefully brings them closer to being a value. However, this is a one-step property: if a progressive term takes a step, we know nothing about the term it steps to. Safety is the many-step version of being progressive, where e is safe if all terms reachable from e are progressive. (By transitivity of \rightsquigarrow^* , it follows that all terms reachable from e are safe.)

We prove type safety in two parts, called *progress* and *preservation*.

Progress says that a well-typed term is progressive. We need a simple helper lemma to show this theorem.

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

- if $A = \text{int}$, then $v = \bar{n}$ for some n
- if $A = A_1 \rightarrow A_2$ for some A_1, A_2 , then $v = \lambda x. e$ for some x, e

Proof. By inversion. □

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

Proof. We proceed by induction on $\vdash e : A$. The induction statement is as follows:

$$IH(\Gamma, e, A) := \Gamma = \emptyset \Rightarrow \text{progressive}(e)$$

Note that we have to generalize over Γ to even be able to do induction, but we explicitly keep around the information that the context is empty. This is crucial to make the proof go through. In the cases below, we have already replaced all the Γ by \emptyset again.

- **VAR:** $e = x$.
Have: $x : A \in \emptyset$. To Show: x is progressive.
Done by contradiction: inversion on $x : A \in \emptyset$ shows that this case is impossible.
- **LAM:** $e = \lambda x. e$ and $A = A' \rightarrow B'$.
Have: $\Gamma, x : A \vdash e : B'$. To Show: $\lambda x. e$ is progressive.
 $\lambda x. e$ is a value hence progressive.
- **APP:** $e = e_1 e_2$.
Have: $\vdash e_1 : A' \rightarrow A$, e_1 is progressive (induction hypothesis), and $\vdash e_2 : A'$.¹
To Show: $e_1 e_2$ is progressive.
We distinguish three cases:
 1. Let e_1 and e_2 be values. Then by **Lemma 9** $e_1 = \lambda x. e$ for some x and e . Thus, by **CTX** (with $K := \bullet$) and **BETA** we have $e_1 e_2 \rightsquigarrow e[e_2/x]$.
 2. Let $e_1 \rightsquigarrow e'_1$ (since it is progressive and not a value) and e_2 be a value. Then $e_1 e_2 \rightsquigarrow e'_1 e_2$ by **Lemma 1** with $K := (\bullet e_2)$.
 3. Let $e_2 \rightsquigarrow e'_2$. Then $e_1 e_2 \rightsquigarrow e_1 e'_2$ by **Lemma 1** with $K := (e_1 \bullet)$.
- **PLUS:** $e = e_1 + e_2$, $A = \text{int}$.
Have: $\vdash e_1 : \text{int}$ and $\vdash e_2 : \text{int}$, e_1 and e_2 are progressive (induction hypothesis).
To Show: $e_1 + e_2$ is progressive.
We distinguish three cases:
 1. Let e_1 and e_2 be values. Then by **Lemma 9** $e_1 = \bar{n}$ and $e_2 = \bar{m}$ for some n and m .
Thus, by **CTX** (with $K := \bullet$) and **PLUS** we have $e_1 + e_2 \rightsquigarrow \overline{n + m}$.
 2. Let $e_1 \rightsquigarrow e'_1$ and e_2 be a value. Then $e_1 + e_2 \rightsquigarrow e'_1 + e_2$ by **Lemma 1** with $K := (\bullet + e_2)$.
 3. Let $e_2 \rightsquigarrow e'_2$. Then $e_1 + e_2 \rightsquigarrow e_1 + e'_2$ by **Lemma 1** with $K := (e_1 + \bullet)$.
- **INT:** $e = \bar{n}$, $A = \text{int}$.
To Show: \bar{n} is progressive.
 \bar{n} is already a value.

□

Preservation means that when a well-typed term takes a step, the resulting term is still well-typed (at the same type). Before we can show that theorem, we first have to prove a sequence of intermediate results about our type system.

¹Very often, we discard hypotheses like $\vdash e_1 : A' \rightarrow A$ and only state their corresponding induction hypothesis (e_1 is progressive). Once every now and then, however, it is needed. This case is an example of that: we need the type judgement to invoke **Lemma 9**.

For the first lemma, we need the notion of one context being “included” in another. It is defined as follows:

$$\Gamma_1 \subseteq \Gamma_2 := \forall x, A. x : A \in \Gamma_1 \Rightarrow x : A \in \Gamma_2$$

This lemma is called “Weakening” since it shows that typing is preserved under larger contexts Γ . Larger contexts mean more assumptions, so the resulting statement is weaker.

Lemma 11 (Weakening). *If $\Gamma_1 \vdash e : A$ and $\Gamma_1 \subseteq \Gamma_2$, then $\Gamma_2 \vdash e : A$.*

Proof. We proceed by induction on $\Gamma_1 \vdash e : A$ generalizing Γ_2 .

This means that the induction statement is:

$$IH(\Gamma_1, e, A) := \forall \Gamma_2. \Gamma_1 \subseteq \Gamma_2 \Rightarrow \Gamma_2 \vdash e : A$$

- **VAR**: $e = x$.
Have: $x : A \in \Gamma_1$. To Show: $\Gamma_2 \vdash x : A$.
Done by **VAR** using assumption $\Gamma_1 \subseteq \Gamma_2$.
- **LAM**: $e = \lambda x. e', A = A' \rightarrow B'$.
Have: $\Gamma_1, x : A' \vdash e' : A' \rightarrow B'$ and $IH((\Gamma_1, x : A'), e', A')$
(Explanation: the first hypothesis comes directly from **LAM**, the second is its corresponding induction hypothesis, *i.e.*, it is the original induction statement with Γ_1 , e , and A replaced by the context, term, and type used in the premise of the **LAM** rule. We don’t actually need the first hypothesis, so in the remaining cases, we only mention having the induction hypothesis.)
To Show: $\Gamma_2 \vdash \lambda x. e : A' \rightarrow B'$.
Done by **LAM** and IH with $\Gamma_2 := \Gamma_2, x : A'$.
- **APP**: $e = e_1 e_2, A = B'$.
Have: $IH(\Gamma_1, e_1, (A' \rightarrow B'))$, $IH(\Gamma_1, e_2, A')$. To Show: $\Gamma_2 \vdash e_1 e_2 : B'$.
Done by **APP** and the induction hypotheses instantiated with Γ_2 .
- **PLUS**: $e = e_1 + e_2, A = \text{int}$.
Have: $IH(\Gamma_1, e_1, \text{int})$, $IH(\Gamma_1, e_2, \text{int})$. To Show: $\Gamma_2 \vdash e_1 + e_2 : \text{int}$.
Done by **PLUS** and the induction hypotheses instantiated with Γ_2 .
- **INT**: $e = \bar{n}, A = \text{int}$.
To Show: $\Gamma_2 \vdash \bar{n} : \text{int}$.
Done by **INT**.

□

Lemma 12 (Substitution). *If $\Gamma, x : A \vdash e : B$ and $\vdash e' : A$, then $\Gamma \vdash e[e'/x] : B$.*

Proof. By induction on e for arbitrary B and Γ . The induction statement becomes:

$$IH(e) := \forall B, \Gamma. \Gamma, x : A \vdash e : B \Rightarrow \Gamma \vdash e[e'/x] : B$$

Note that the $\vdash e' : A$ is unaffected by the induction (it contains neither e nor B nor Γ) and so it stays out of the induction statement.

- Case $e = y$:
Have: $\Gamma, x : A \vdash y : B$ and $\vdash e' : A$.
(Explanation: the first hypothesis comes from the induction statement, which is an implication, and the second is a separate assumption of our lemma that we will have in all cases.)
To Show: $\Gamma \vdash y[e'/x] : B$. By inversion of typing of y , we have: $y : B \in \Gamma, x : A$.

- Case $x = y$:
 Have: $A = B$ (from $x : B \in \Gamma, x : A$)
 To Show: $\Gamma \vdash e' : B$
 Done by weakening (**Lemma 11**) and $\vdash e' : A$.
- Case $x \neq y$:
 Have: $y : B \in \Gamma$.
 To Show: $\Gamma \vdash y : B$
 Done by **VAR**.
- Case $e = e_1 e_2$
 Have: $\Gamma, x : A \vdash e_1 e_2 : B$ and $\vdash e' : A$ and $IH(e_1)$ and $IH(e_2)$.
 To Show: $\Gamma \vdash (e_1 e_2)[e'/x] : B$.
 By inversion on typing, we have: $\Gamma, x : A \vdash e_1 : B' \rightarrow B$ and $\Gamma, x : A \vdash e_2 : B'$.
 By the definition of substitution, it suffices to show: $\Gamma \vdash e_1[e'/x] e_2[e'/x] : B$.
 Done by **APP**, using $IH(e_1)$ for typing $e_1[e'/x]$ and $IH(e_2)$ for typing $e_2[e'/x]$.
- Case $e = \lambda y. e_1$
 Have: $\Gamma, x : A \vdash \lambda y. e_1 : B$ and $\vdash e' : A$ and $IH(e_1)$.
 To Show: $\Gamma \vdash (\lambda y. e_1)[e'/x] : B$.
 By inversion of typing, we have: $B = B_1 \rightarrow B_2$ and $\Gamma, x : A, y : B_1 \vdash e_1 : B_2$.
 - Case $x \neq y$:
 To Show: $\Gamma \vdash \lambda y. e_1[e'/x] : B_1 \rightarrow B_2$.
 By **LAM**, it suffices to show: $\Gamma, y : B_1 \vdash e_1[e'/x] : B_2$.
 By IH , it suffices to show: $\Gamma, y : B_1, x : A \vdash e_1 : B_2$.
 We can apply weakening (**Lemma 11**) to swap the order of x and y , then we are done.
 - Case $x = y$:
 To Show: $\Gamma \vdash \lambda y. e_1 : B_1 \rightarrow B_2$.
 By **LAM**, it suffices to show: $\Gamma, y : B_1 \vdash e_1 : B_2$.
 Done by weakening (**Lemma 11**) the context to $\Gamma, x : A, y : B_1$.
 (Note that $\Gamma, x : A, y : B_1 \subseteq \Gamma, y : B_1$ since $x = y$.)
- Case $e = e_1 + e_2$
 Have: $\Gamma, x : A \vdash e_1 + e_2 : B$ and $\vdash e' : A$ and $IH(e_1)$ and $IH(e_2)$.
 To Show: $\Gamma \vdash (e_1 + e_2)[e'/x] : B$.
 By inversion on typing, we have: $B = \text{int}$ and $\Gamma, x : A \vdash e_1 : \text{int}$ and $\Gamma, x : A \vdash e_2 : \text{int}$.
 By the definition of substitution, it suffices to show: $\Gamma \vdash e_1[e'/x] + e_2[e'/x] : B$.
 Done by **PLUS**, using $IH(e_1)$ for typing $e_1[e'/x]$ and $IH(e_2)$ for typing $e_2[e'/x]$.
- Case $e = \bar{n}$
 Have: $\Gamma, x : A \vdash \bar{n} : B$ and $\vdash e' : A$.
 To Show: $\Gamma \vdash \bar{n}[e'/x] : B$.
 By inversion on typing, we have $B = \text{int}$.
 By the definition of substitution, it suffices to show: $\Gamma \vdash \bar{n} : B$.
 Done by **INT**.

□

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

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

- **BETA**, $e = (\lambda x. e_1) v$ and $e' = e_1[v/x]$: To show: $\vdash e_1[v/x] : A$.
 By inversion on $\vdash e : A$ we have $\vdash \lambda x. e_1 : A_0 \rightarrow A$ and $\vdash v : A_0$ for some A_0 . By inversion on $\vdash \lambda x. e_1 : A_0 \rightarrow A$ we have $x : A_0 \vdash e_1 : A$. By **Lemma 12**, we have $\vdash e_1[v/x] : A$.

- **PLUS**, $e = \bar{n} + \bar{m}$ and $e' = \overline{n+m}$: By inversion on $\vdash e : A$, we have $A = \text{int}$. We establish $\vdash \overline{n+m} : \text{int}$ by **INT**. \square

To complete the proof of preservation, we define the notion of *contextual typing* $\vdash K : A \Rightarrow B$, which expresses that an evaluation context is of type B if filled with an expression of type A . We could define a set of inductive typing rules dedicated to type-checking evaluation contexts, but it turns out to be simpler to define contextual typing as the property we expect from it: if filled with a well-typed expression, we obtain a well-typed term. (This “extensional” definition is due to Jules Jacobs.)

Contextual typing

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

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

Lemma 14 (Decomposition). *If $\vdash K[e] : A$, then there exists B s.t.*

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

Proof. We proceed by induction on K , with the following induction statement:

$$IH(K) := \forall A, e. (\vdash K[e] : A) \rightarrow \exists B. (\vdash K : B \Rightarrow A \text{ and } \vdash e : B)$$

- $K = \bullet$:
Have: $\vdash e : A$. To Show: $\exists B. \vdash \bullet : B \Rightarrow A$ and $\vdash e : B$.
Taking $B := A$, it remains to show that $\vdash \bullet : A \Rightarrow A$ and $\vdash e : A$.
We have the second; for the first:
We assume a fresh e' such that $\vdash e' : A$ and we have to show $\vdash \bullet[e'] : A$.
This is immediate.
- $K = e' K'$:
Have: $\vdash e' K'[e] : A$ and $IH(K')$. To Show: $\exists B. \vdash e' K' : B \Rightarrow A$ and $\vdash e : B$.
By inversion on $\vdash e' K'[e] : A$, we learn $\vdash e' : A' \rightarrow A$ and $\vdash K'[e] : A'$ for some A' .
Applying $IH(K')$ to the latter, we find there is some B' so that $\vdash K' : B' \Rightarrow A'$ and $\vdash e : B'$.
Taking $B := B'$, it remains to show that $\vdash e' K' : B' \Rightarrow A$ and $\vdash e : B'$.
We have the second; for the first:
We assume a fresh e'' such that $\vdash e'' : B'$ and we have to show $\vdash (e' K')[e''] : A$.
Using **APP**, it suffices to show $\vdash e' : A' \rightarrow A$ and $\vdash K'[e''] : A'$.
We have the first; for the second:
We apply typing of K' with e'' , so it suffices to show $\vdash e'' : B'$, which we also have.
- $K = K' v$:
Have: $\vdash K'[e] v : A$ and $IH(K')$. To Show: $\exists B. \vdash K' v : B \Rightarrow A$ and $\vdash e : B$.
By inversion on $\vdash K'[e] v : A$, we find $\vdash K'[e] : A' \Rightarrow A$ and $\vdash v : A'$.
Applying $IH(K')$ to the former, there is some B' so that $\vdash K' : B' \Rightarrow (A' \rightarrow A)$ and $\vdash e : B'$.
Taking $B := B'$, it remains to show that $\vdash K' v : B' \Rightarrow A$ and $\vdash e : B'$.
We have the second; for the first:
We assume a fresh e' such that $\vdash e' : B'$ and we have to show $\vdash (K' v)[e'] : A$.
Using **APP**, it suffices to show $\vdash K'[e'] : A' \rightarrow A$ and $\vdash v : A'$.
We have the second; for the first:
We apply typing of K' with e' , so it suffices to show $\vdash e' : B'$, which we also have.
- $K = e' + K'$:
Have: $\vdash e' + K'[e] : A$ and $IH(K')$. To Show: $\exists B. \vdash e' + K' : B \Rightarrow A$ and $\vdash e : B$.
By inversion on $\vdash e' + K'[e] : A$, we learn $A = \text{int}$, $\vdash e' : \text{int}$, and $\vdash K'[e] : \text{int}$.
Applying $IH(K')$ to the latter, we find there is some B' so that $\vdash K' : B' \Rightarrow \text{int}$ and $\vdash e : B'$.

Taking $B := B'$, it remains to show that $\vdash e' + K' : B' \Rightarrow \text{int}$ and $\vdash e : B'$.

We have the second; for the first:

We assume a fresh e'' such that $\vdash e'' : B'$ and we have to show $\vdash (e' + K')[e''] : \text{int}$.

Using **PLUS**, it suffices to show $\vdash e' : \text{int}$ and $\vdash K'[e''] : \text{int}$.

We have the first; for the second:

We apply typing of K' with e'' , so it suffices to show $\vdash e'' : B'$, which we also have.

- $K = K' + v$:

Have: $\vdash K'[e] + v : A$ and $IH(K')$. To Show: $\exists B. \vdash K' + v : B \Rightarrow A$ and $\vdash e : B$.

By inversion on $\vdash K'[e] + v : A$, we learn $A = \text{int}$, $\vdash K'[e] : \text{int}$, and $\vdash v : \text{int}$.

Applying $IH(K')$ to the second, we find there is some B' so that $\vdash K' : B' \Rightarrow \text{int}$ and $\vdash e : B'$.

Taking $B := B'$, it remains to show that $\vdash K' + v : B' \Rightarrow \text{int}$ and $\vdash e : B'$.

We have the second; for the first:

We assume a fresh e' such that $\vdash e' : B'$ and we have to show $\vdash (K' + v)[e'] : \text{int}$.

Using **PLUS**, it suffices to show $\vdash K'[e'] : \text{int}$ and $\vdash v : \text{int}$.

We have the second; for the first:

We apply typing of K' with e' , so it suffices to show $\vdash e' : B'$, which we also have.

□

Lemma 15 (Composition). *If $\vdash K : B \Rightarrow A$ and $\vdash e : B$, then $\vdash K[e] : A$.*

Proof. By definition. (This would be an induction for the inductive definition of contextual typing.) □

Theorem 16 (Preservation). *If $\vdash e : A$ and $e \rightsquigarrow e'$, then $\vdash e' : A$.*

Proof. Invert $e \rightsquigarrow e'$ to obtain K, e_1, e'_1 s.t. $e = K[e_1]$ and $e' = K[e'_1]$ and $e_1 \rightsquigarrow_b e'_1$.

By **Lemma 14**, there exists B s.t. $\vdash K : B \Rightarrow A$ and $\vdash e_1 : B$.

By **Lemma 13**, from $\vdash e_1 : B$ and $e_1 \rightsquigarrow_b e'_1$, we have $\vdash e'_1 : B$.

By **Lemma 15**, from $\vdash e'_1 : B$ and $\vdash K : B \Rightarrow A$, we have $\vdash K[e'_1] : A$, so we are done. □

Corollary 17 (Type Safety). *If $\vdash e : A$, then e is safe.*

1.5 Termination

In this section, we want to prove that every well-typed term eventually reduces to a value. This also establishes a relationship between typing and big-step evaluation.

Statement 18 (Termination). *If $\vdash e : A$, then there exists v s.t. $e \Downarrow v$.*

We define the notion of “semantically good” expressions and values, by defining predicates that say when an expression or value is considered “semantically well-typed” at a given type. In a more general setting, these predicates become relations, and the entire setup is also called a *logical relation*.

To be fully formally precise, we have to ensure that terms do not arbitrarily contain free variables. Unfortunately, this distracts from the core idea of how this proof is set up, so we print them in gray such that they can be more easily ignored. The book-keeping details would be different if we had chosen a different way to formalize substitution; the rest of the logical relation always follows the same general structure.

Those details aside, values are “good” if they match their type, and expressions are “good” if they evaluate to a “good” value.

Value Relation

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

$$\begin{aligned} \mathcal{V}[[\text{int}]] &:= \{\bar{n} \mid n \in \mathbb{Z}\} \\ \mathcal{V}[[A \rightarrow B]] &:= \{\lambda x. e \mid \text{fv}(e) \subseteq \{x\} \wedge \forall v. v \in \mathcal{V}[[A]] \Rightarrow e[v/x] \in \mathcal{E}[[B]]\} \end{aligned}$$

Expression Relation

 $\mathcal{E}[A]$

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

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

Proof. Follows directly from $v \downarrow v$. □

This notion of “good” values can be lifted to contexts: a substitution function γ is “good” for some context Γ if it assigns a “good” value (of appropriate type) to each variable in that context.

Context Relation

 $\mathcal{G}[\Gamma]$

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

Earlier in these lecture notes, we have discussed substitution of a single variable. For the definition of semantic typing, we need *parallel substitution*: the action of substituting multiple free variables by values at the same time, using a map γ from variable names to values.

Parallel Substitution

 $\gamma(e)$

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

We can now define a *semantic typing judgment*.

Semantic Typing

 $\Gamma \models e : A$

$$\Gamma \models e : A := \text{fv}(e) \subseteq \text{dom}(\Gamma) \wedge \forall \gamma \in \mathcal{G}[\Gamma]. \gamma(e) \in \mathcal{E}[A]$$

The theorem we want to ultimately prove about semantic typing is *semantic soundness*: every syntactically well-typed term also satisfies our semantic notion of typing. From there, the desired termination result will be a simple corollary.

Statement 20 (Semantic Soundness). *If $\Gamma \vdash e : A$, then $\Gamma \models e : A$.*

However, to prove this, we first need to show the *compatibility lemmas*: for each typing rule, we show its semantic counterpart, which is obtained by replacing all occurrences of syntactic typing in the rule by semantic typing.

Lemma 21 (Compatibility with **INT**). *We have $\Gamma \models n : \text{int}$.*

Proof. To show $\Gamma \models n : \text{int}$, we first show that $\text{fv}(n) \subseteq \text{dom}(\Gamma)$, which is immediate.

We may assume $\gamma \in \mathcal{G}[\Gamma]$ and have to show $\gamma(n) \in \mathcal{E}[\text{int}]$.

By value inclusion (**Lemma 19**) and the definition of substitution, it suffices to show $n \in \mathcal{V}[\text{int}]$.

This follows directly from the definition of $\mathcal{V}[-]$. □

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

Proof. To show $\Gamma \models x : A$, we first show that $\text{fv}(x) \subseteq \text{dom}(\Gamma)$, which follows from $x : A \in \Gamma$. We may assume $\gamma \in \mathcal{G}[\Gamma]$ and have to show $\gamma(x) \in \mathcal{E}[A]$. From that assumption and $x : A \in \Gamma$, it follows that $\gamma(x) \in \mathcal{V}[A]$. By value inclusion (Lemma 19), we are done. \square

Lemma 23 (Compatibility with **PLUS**). *If $\Gamma \models e_1 : \text{int}$ and $\Gamma \models e_2 : \text{int}$ then $\Gamma \models e_1 + e_2 : \text{int}$.*

Proof. To show $\Gamma \models e_1 + e_2 : \text{int}$, we first show that $\text{fv}(e_1 + e_2) \subseteq \text{dom}(\Gamma)$, which follows from $\text{fv}(e_1) \subseteq \text{dom}(\Gamma)$ and $\text{fv}(e_2) \subseteq \text{dom}(\Gamma)$.

We may assume $\gamma \in \mathcal{G}[\Gamma]$ and have to show $\gamma(e_1) + \gamma(e_2) \in \mathcal{E}[\text{int}]$.

From the assumptions, there exist v_1, v_2 s.t. $\gamma(e_1) \downarrow v_1 \in \mathcal{V}[\text{int}]$ and $\gamma(e_2) \downarrow v_2 \in \mathcal{V}[\text{int}]$.

From $v_1 \in \mathcal{V}[\text{int}]$, we have that $v_1 = \overline{n_1}$ for some n_1 , and similarly $v_2 = \overline{n_2}$ for some n_2 .

Hence we get that $\gamma(e_1) + \gamma(e_2) \downarrow \overline{n_1 + n_2} \in \mathcal{V}[\text{int}]$, which finishes the proof. \square

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

Proof. To show $\Gamma \models e_1 e_2 : B$, we first show that $\text{fv}(e_1 e_2) \subseteq \text{dom}(\Gamma)$, which follows from $\text{fv}(e_1) \subseteq \text{dom}(\Gamma)$ and $\text{fv}(e_2) \subseteq \text{dom}(\Gamma)$.

We may assume $\gamma \in \mathcal{G}[\Gamma]$ and have to show $\gamma(e_1) \gamma(e_2) \in \mathcal{E}[B]$.

From the assumptions, there exist v_1, v_2 s.t. $\gamma(e_1) \downarrow v_1 \in \mathcal{V}[A \rightarrow B]$ and $\gamma(e_2) \downarrow v_2 \in \mathcal{V}[A]$.

From $v_1 \in \mathcal{V}[A \rightarrow B]$, we have that there exist x, e s.t. $v_1 = \lambda x. e$ and $e[v_2/x] \in \mathcal{E}[B]$.

By the definition of $\mathcal{E}[_]$, we have that there exists v s.t. $e[v_2/x] \downarrow v \in \mathcal{V}[B]$.

Thus we have that $\gamma(e_1) \gamma(e_2) \downarrow v \in \mathcal{V}[B]$, which finishes the proof. \square

Lemma 25 (Compatibility with **LAM**). *If $\Gamma, x : A \models e : B$ then $\Gamma \models \lambda x. e : A \rightarrow B$.*

Proof. To show $\Gamma \models \lambda x. e : A \rightarrow B$, we first show that $\text{fv}(\lambda x. e) \subseteq \text{dom}(\Gamma)$, which follows from $\text{fv}(e) \subseteq \text{dom}(\Gamma, x : A)$.

We may assume $\gamma \in \mathcal{G}[\Gamma]$ and have to show $\gamma(\lambda x. e) \in \mathcal{E}[A \rightarrow B]$.

Unfolding the substitution and applying Lemma 19, it suffices to show $\lambda x. (\gamma[x \mapsto \perp])(e) \in \mathcal{V}[A \rightarrow B]$.

We have to show $\text{fv}((\gamma[x \mapsto \perp])(e)) \subseteq \{x\}$, which follows because $\text{fv}(e) \subseteq \text{dom}(\Gamma, x : A)$ and everything in γ is in $\mathcal{V}[_]$ and hence closed.

We suppose $v \in \mathcal{V}[A]$ and have to show $((\gamma[x \mapsto \perp])(e))[v/x] \in \mathcal{E}[B]$.

We have $((\gamma[x \mapsto \perp])(e))[v/x] = \gamma[x \mapsto v](e)$ because everything in γ is closed.

Let $\gamma' := \gamma[x \mapsto v]$; our goal then is $\gamma'(e) \in \mathcal{E}[B]$.

Applying semantic typing of e , it suffices to show $\gamma' \in \mathcal{G}[\Gamma, x : A]$.

This follows from $\gamma \in \mathcal{G}[\Gamma]$ and $v \in \mathcal{V}[A]$. \square

This is where we really needed those book-keeping assumptions about free variables: we need to know that everything in γ is closed. We achieve this by ensuring that “good” values are closed. But that means when we want to show that a λ -term is a “good” value, we need to prove that its body (in the lemma above, that’s $(\gamma[x \mapsto \perp])(e)$) has no free variables except for x . This boils down to knowing what the free variables in e are, and the only thing we know about e is that it is semantically well-typed, so we have to also add the requirement that semantically well-typed terms have their free variables captured by Γ .

Theorem 26 (Semantic Soundness). *If $\Gamma \vdash e : A$, then $\Gamma \models e : A$.*

Proof. By induction on $\Gamma \vdash e : A$, and then using the compatibility lemmas of the semantic typing proven above. In each case, our induction hypothesis exactly lines up with the corresponding compatibility lemma. \square

Corollary 27 (Termination). *If $\emptyset \vdash e : A$, then there exists v s.t. $e \downarrow v$.*

Proof. By Theorem 26, we have $\emptyset \models e : A$. Pick γ to be the empty substitution \emptyset , which clearly is in $\mathcal{G}[\emptyset]$. Hence $\emptyset(e) = e \in \mathcal{E}[A]$. By definition then, $\exists v. e \downarrow v$. \square

2 De Bruijn representation

So far, we have used strings to represent variable names, and we have used the naive substitution function defined all the way at the beginning of [Section 1](#). However, as we have already discussed, this definition suffers from the *capturing problem*. Specifically, consider the open term $e' := \lambda x. x + y$ (it is open because y is free in this term), and what happens when we substitute it in for z in $e := \lambda y. z$, *i.e.*, we compute $e[e'/z]$. The result is:

$$\lambda y. \lambda x. x + y$$

y used to be an unused variable, but now suddenly the free variable y from our term e' has been “captured” by the completely unrelated binder in e . e was meant to be a “constant” function that always returns the same value (z), but now its return value actually depends on y . Everything would break if we allowed that kind of substitution.

We have managed to keep everything together by ensuring that we only ever substitute closed terms, but when we defined our logical relation, the seams already started to show: we had to add some extra book-keeping conditions to prove that indeed everything we substitute is closed. As we want to extend our type system with more features, in particular polymorphism, this problem will get worse, and we don’t be able to “cheat” our way out of the issue like we have been done so far.

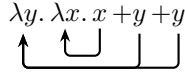
One approach would be to make substitution smarter, such that it renames the y in e' when plugging in our value e . The result of substitution would then be:

$$\lambda y'. \lambda x. x + y$$

This time, y is still free.

However, defining this renaming in Coq is cumbersome, and reasoning about it is even worse. So we are going to use a different approach: we are going to completely re-think how we even represent variables in terms. This new representation we are considering is called *De Bruijn representation*.

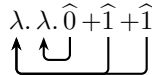
De Bruijn representation Why do we need variable names to begin with? The information we really need when we encounter a variable is: which λ does this belong to? To encode this information, we add a name to the λ , and then we use the same name whenever we mean “the variable for *that* λ ”. For instance:

$$\lambda y. \lambda x. x + y + y$$


The key idea of De Bruijn representation is to completely forego names, and instead represent variables by *the number of λ that should be skipped before we arrive at the one this variable refers to*.

This is best demonstrated with some examples. For instance, the term $\lambda x. x$ gets represented as $\lambda. \hat{0}$. The 0 indicates that this variable refers to the innermost λ . There is no longer a name at the λ since we do not need it any more. The term $\lambda y. y$ has exactly the same representation! The term $\lambda x. \lambda y. y$ is represented as $\lambda. \lambda. \hat{0}$: the variable refers to the innermost λ . The term $\lambda x. \lambda y. x$ is represented as $\lambda. \lambda. \hat{1}$, since now the variable refers to the λ “one layer out”.

The term from the example picture above now becomes:

$$\lambda. \lambda. \hat{0} + \hat{1} + \hat{1}$$


Note how each arrow starting at $\hat{1}$ skips over exactly 1 λ , while each arrow starting at $\hat{0}$ does not skip any λ .

One consequence of this definition is that the same variable has a different number in different situations. For example, $\lambda. \hat{0} + (\lambda. \hat{1}) \hat{0}$ corresponds to $\lambda x. x + (\lambda y. x) x$: the same variable is used three

times, but it looks different! The $\widehat{1}$ is under one more λ so the index needs to be incremented by 1 to refer to the outer λ . Here's a detailed rendering of that term:

$$\lambda.\widehat{0} + (\lambda.\widehat{1})\widehat{0}$$

Again, each arrow skips over exactly as many λ as indicated by the De Bruijn index where it starts.

To represent terms with free variables, we use indices that are larger than the number of λ that surround them. For instance, the term $\widehat{0} + \widehat{1}$ refers to free variables with indices 0 and 1, respectively. This is like $x + y$, assuming x has index 0 and y has index 1. The term $\lambda.\widehat{0} + \widehat{1}$ is like $\lambda z. z + x$: the left operand of the sum refers to the variable bound by the λ ; the right variable in the sum refers to the free variable with index 0. In general, if variable \widehat{n} is used below $k < n$ binders, then it refers to the free variable with index $n - k$. Basically, if we are supposed to skip n λ , but after skipping k λ we already reached the top, then we keep skipping into the context: the remaining number tells us the index of the variable in the context.

Formally speaking, the inductive definition of terms with De Bruijn variables is

Variables	$x, y \in \mathbb{N}$
Runtime Terms	$e ::= \widehat{x} \mid \lambda. e \mid e_1 e_2 \mid e_1 + e_2 \mid \overline{n}$
(Runtime) Values	$v ::= \lambda. e \mid \overline{n}$

2.1 Substitution

Now that we have an idea for how De Bruijn representation can unambiguously represent terms without using names, we consider how one performs substitution on such terms. The exact definition turns out to be subtle, but deriving it is a completely mechanical process. In fact, in Coq we will use a library called Autosubst [3] that automatically defines substitution for us and proves all the key lemmas needed to reason about substitution. To follow the rest of this course, it is not necessary to fully understand how exactly De Bruijn substitution is defined—we will mostly pretend that we have named binders everywhere with substitution that “just magically works”, and use De Bruijn representation for the Coq formalization to make it actually work. The important point is getting enough intuition for the De Bruijn substitution so that what it does on concrete examples “makes sense”.

So let us first consider our example from earlier. The term e' now looks like $\lambda.\widehat{0} + \widehat{1}$. The term e becomes $\lambda.\widehat{1}$. We consider $e[e'/\]$, which denotes e with the first (index 0) variable replaced with e' . The result of this should be $\lambda.\lambda.\widehat{0} + \widehat{2}$. This is *almost* e with the body of the λ replaced by e' , except that we had to adjust e' a bit: the $\widehat{1}$ in e' used to refer to the free variable with index 0, and we still want it to refer to that variable when we are done. However, now it lives under two λ , so we need to write $\widehat{2}$ to refer to the free variable with index 0. All free variables in e' need to have their index increased by 1 for each λ below which we traverse.

As a second example, consider $(\widehat{0}\widehat{5} + \widehat{1})[e'/\]$. The result is $(\lambda.\widehat{0} + \widehat{1})\widehat{5} + \widehat{0}$. This time e' is completely unchanged, since it is not below any λ , but the free variable $\widehat{1}$ in the original term got changed to $\widehat{0}$. Substituting a term replaces the free variable with index 0 but that term, and then removes that free variable, so all the other free variables have to be decremented by one.

Formal definition Let us start with the (comparatively) simpler cases of $e[e'/\]$:

$$(e_1 e_2)[e'/\] := e_1[e'/\] e_2[e'/\]$$

$$\widehat{x}[e'/\] := \begin{cases} e' & \text{if } x = 0 \\ \widehat{x-1} & \text{otherwise} \end{cases}$$

For application we just recurse into both sides, nothing interesting happens. When we find a variable, we do a case distinction: if this is the variable 0 that we were looking for, then we plug in

e' , otherwise we decrement the variable index—that’s the point we just explained with the second example above.

However, when we try to define what happens when substitution traverses through a λ , things become a bit more difficult. After all, below the first λ , if we encounter any $\widehat{0}$, those would not be free variables! They would refer to that λ instead. Plugging e' into $\lambda.\widehat{0} + \widehat{1}$ should yield $\widehat{0} + e'$, since as discussed above, $\widehat{1}$ refers to the free variable with index 0 when we are below one λ .

We could now write a substitution function that keeps track of which variable it is supposed to replace, but it turns out that the theory for the De Bruijn representation works out much more nicely if we instead define *parallel substitution* as our primitive substitution operation [4]. We have already seen that we need parallel substitution for defining the logical relation, so we would have to define it anyway—let’s catch two birds with one stone.

Parallel substitution (often just called “substitution” from now on) is written $e[\sigma]$, where σ is a map from variables to terms. Note that σ is a total map, it has to assign a term to *all* variables. That might sound strange, since only a finite number of variables will ever actually be relevant, but we can just map all other variables to themselves.

On the core lambda calculus (without integers or addition), substitution is defined as follows:

$$\begin{aligned}(e_1 e_2)[\sigma] &:= e_1[\sigma] e_2[\sigma] \\ \widehat{x}[\sigma] &:= \sigma(x) \\ (\lambda. e)[\sigma] &:= \lambda. e[\uparrow \sigma]\end{aligned}$$

where the “lift” operation \uparrow on substitutions is defined as:

$$(\uparrow \sigma)(x) := \begin{cases} \widehat{0} & \text{if } x = 0 \\ \sigma(x - 1)[\uparrow] & \text{otherwise} \end{cases}$$

and the “shift” \uparrow substitution is defined as follows:

$$\uparrow(x) := \widehat{x + 1}$$

In other words, $e[\uparrow]$ increments all free variables by one.

We can now define our desired substitution operation $e[e' /]$ via the “singleton” substitution that plugs in e' for the first variable, and decrements everything else by one:

$$(e' /)(x) := \begin{cases} e' & \text{if } x = 0 \\ \widehat{x - 1} & \text{otherwise} \end{cases}$$

The best way to make sense of this definition is to run through the two examples from above by hand and convince yourself that they give the right result:

$$\begin{aligned}(\lambda. \widehat{1})[\lambda. \widehat{0} + \widehat{1} /] &= \lambda. \lambda. \widehat{0} + \widehat{2} \\ (\widehat{0} \bar{5} + \widehat{1})[\lambda. \widehat{0} + \widehat{1} /] &= (\lambda. \widehat{0} + \widehat{1}) \bar{5} + \widehat{0}\end{aligned}$$

This definition turns out to work perfectly, except for one major flaw: it is recursive! We used \uparrow to define substitution, and we used substitution to define \uparrow . This recursive knot can be broken by exploiting that the substitution $\widehat{}$ needed to define \uparrow is very simple—it just increments all free variables by one. This is what we call a *renaming*, a map from variables to variables. The full story for defining De Bruijn substitution is to first define substitution for renamings, use that to define \uparrow , and then use that to define the main substitution operation. However, this is a technical detail. We can use the equations above as the defining equations for De Bruijn substitution without worrying about how exactly we can prove that they are well-formed.

2.2 Operational Semantics

Now that we have a suitable substitution operation, we can use it to define the operational semantics. We only show the definition of beta reduction in the base reduction judgment:

Base reduction

$$e_1 \rightsquigarrow_b e_2$$

$$\begin{array}{c} \text{BETA} \\ (\lambda. e) v \rightsquigarrow_b e[v /] \end{array}$$

2.3 Type System

More interesting is to consider what happens in the type system. Now that free variables in a term are just an index, contexts will be just lists of types!

$$\text{Variable Contexts} \quad \Gamma ::= \emptyset \mid \Gamma, A$$

Context lookup

$$\Gamma(n) = A$$

To define the typing rules, we need to define the lookup judgment $\Gamma(n) = A$ in such a typing context:

$$\begin{array}{c} \text{HERE} \\ (\Gamma, A)(0) = A \end{array} \qquad \begin{array}{c} \text{THERE} \\ \Gamma(x) = A \\ \hline (\Gamma, B)(x+1) = A \end{array}$$

Curry-style typing

$$\Gamma \vdash e : A$$

Then we define typing on runtime terms as follows:

$$\begin{array}{c} \text{VAR} \\ \Gamma(x) = A \\ \hline \Gamma \vdash \hat{x} : A \end{array} \qquad \begin{array}{c} \text{LAM} \\ \Gamma, A \vdash e : B \\ \hline \Gamma \vdash \lambda. e : A \rightarrow B \end{array} \qquad \begin{array}{c} \text{APP} \\ \Gamma \vdash e_1 : A \rightarrow B \quad \Gamma \vdash e_2 : A \\ \hline \Gamma \vdash e_1 e_2 : B \end{array}$$

$$\begin{array}{c} \text{PLUS} \\ \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \\ \hline \Gamma \vdash e_1 + e_2 : \text{int} \end{array} \qquad \begin{array}{c} \text{INT} \\ \Gamma \vdash \bar{n} : \text{int} \end{array}$$

This is basically identical to before. Note how in the **LAM** case, we add A to the context, implicitly giving it index 0 and increasing the indices of all other free variables by one.

Here are some examples of valid typing judgments:

$$\begin{aligned} & \vdash \lambda. \bar{1} + \hat{0} : \text{int} \rightarrow \text{int} \\ & \text{int} \vdash \lambda. \hat{1} + \hat{0} : \text{int} \rightarrow \text{int} \\ & \text{int} \rightarrow \text{int}, \text{int} \vdash \hat{1} \hat{0} : \text{int} \end{aligned}$$

2.4 Type Safety

More interesting is what happens in the type safety proof. Progress is basically unchanged, but the weakening and substitution lemmas required for preservation have to be adjusted to a De Bruijn world with parallel substitutions.

Remember that previously, weakening required $\Gamma_1 \subseteq \Gamma_2$. We will generalize this to $\Gamma_1 \subseteq_\delta \Gamma_2$, which indicates that the renaming δ embeds Γ_1 in Γ_2 :

$$\Gamma_1 \subseteq_\delta \Gamma_2 := \forall x, A. \Gamma_1(x) = A \Rightarrow \Gamma_2(\delta(x)) = A$$

So not only is Γ_1 a subset of Γ_2 , but moreover δ tells us for each type in Γ_1 where we can find the same type in Γ_2 . (Remember that renamings are a special case of substitution, they map variables to variables.)

Weakening then turns into the renaming lemma:

Lemma 28 (Renaming). *If $\Gamma_1 \subseteq_\delta \Gamma_2$ and $\Gamma_1 \vdash e : A$, then $\Gamma_2 \vdash e[\delta] : A$.*

By generalizing this notion of context inclusion from renamings to arbitrary substitutions, we arrive at the idea of a *type-preserving substitution* which is checked in context Γ_2 and assigns types to all variables in Γ_1 :

$$\Gamma_2 \vdash \sigma : \Gamma_1 := \forall x, A. \Gamma_1(x) = A \Rightarrow \Gamma_2 \vdash \sigma(x) : A$$

Note that for renamings, $\Gamma_1 \subseteq_\delta \Gamma_2$ is equivalent to $\Gamma_2 \vdash \delta : \Gamma_1$. (Mind the swapped order of contexts!)

We use this to state the (parallel) substitution lemma:

Lemma 29 (Substitution). *If $\Gamma_2 \vdash \sigma : \Gamma_1$ and $\Gamma_1 \vdash e : A$, then $\Gamma_2 \vdash e[\sigma] : A$.*

The proof of this lemma uses the renaming lemma. On the one hand, this approach of first showing how typing is preserved under renaming and then under arbitrary substitutions matches how we used to first show the weakening lemma (Lemma 11) and then the substitution lemma (Lemma 12). However, there also is a parallel with the definition of (parallel) substitution itself, where we first define how to apply a renaming to a term and then use that to define how to apply substitution to a term.

From the lemma for general substitutions, we can derive a lemma for single-term substitution:

Lemma 30 (One-Term Substitution). *If $\Gamma \vdash e' : B$ and $\Gamma, B \vdash e : A$, then $\Gamma \vdash e[e' /] : A$.*

Note that this lemma is significantly stronger than Lemma 12: in the old lemma, the term e' that was substituted had to be well-typed in the empty context, but now, it may use all the variables in Γ . We can now substitute open terms! This is only possible because De Bruijn substitution is capture-avoiding.

The rest of the type safety proof then proceeds pretty much exactly as before.

2.5 Termination

Finally, we consider what happens with our termination proof when we use De Bruijn representation. The value and expression relation are basically unchanged, except that we do not need to talk about free variables any more. We have arrived at the logical relation in its purest form:

Value Relation

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

$$\begin{aligned} \mathcal{V}[\text{int}] &:= \{\bar{n} \mid n \in \mathbb{Z}\} \\ \mathcal{V}[A \rightarrow B] &:= \{\lambda. e \mid \forall v. v \in \mathcal{V}[A] \Rightarrow e[v /] \in \mathcal{E}[B]\} \end{aligned}$$

Expression Relation

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

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

Context Relation

$$\mathcal{G}[A]$$

The context relation says when a value substitution γ is well-typed at a given context:

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

Since values are a subset of expressions, we will freely use value substitutions as if they are regular substitutions.

Semantic typing looks as before, but without worrying about free variables:

$$\Gamma \models e : A := \forall \gamma \in \mathcal{G}[\Gamma]. e[\gamma] \in \mathcal{E}[A]$$

The rest works pretty much exactly as before. The key step is in the compatibility lemma for **LAM**, where we end up having to show $e[\uparrow \gamma][v/] \in \mathcal{E}[B]$. This corresponds to $((\gamma[x \mapsto \perp])(e))[v/x] \in \mathcal{E}[B]$ in the old proof (go back to **Lemma 25** to compare). The $e[\uparrow \gamma]$ arises from $(\lambda. e)[\gamma]$, which is part of the definition of $\Gamma \models \lambda. e : A \rightarrow B$, and then to show that this is a “good” value of function type, we apply another substitution to this term. Dealing with this double-substitution is the crux of this compatibility lemma and the trickiest part of the entire termination proof. Before, we argued that $((\gamma[x \mapsto \perp])(e))[v/x] = \gamma[x \mapsto v](e)$ because everything in γ is closed. Now we use the general fact about De Bruijn substitutions that $e[\uparrow \gamma][v/] = e[v :: \gamma]$, where this “cons” operation on substitutions is defined as

$$(e :: \sigma)(x) := \begin{cases} e & \text{if } x = 0 \\ \sigma(x - 1) & \text{otherwise} \end{cases}$$

In other words, if we think of a substitution as an infinite list of terms (index 0 defines $\sigma(0)$, index 1 defines $\sigma(1)$, and so on), then $e :: \sigma$ adds a new element to the beginning of the list and shifts everything else by 1. (The $(e'/)$ substitution defined above can be written as $e' :: \text{id}$, where id is the “identity” substitution which maps n to \hat{n} .)

$e[\uparrow \gamma][v/] = e[v :: \gamma]$ is far from obvious! We will not prove it here. However, it turns out that you can define an algebra of (parallel) substitutions with a nice equational theory, and this fact follows from that theory—see Kathrin Stark’s PhD thesis [4] for all the details. The algebra also comes with normal forms and a decision procedure, which the Autosubst library implements in Coq, so we get the above equality for free from Autosubst.

After applying that lemma, we just need to show $v :: \gamma \in \mathcal{G}[\Gamma, A]$, which directly follows from $v \in \mathcal{V}[A]$ and $\gamma \in \mathcal{G}[\Gamma]$. The proof then completes without ever being concerned with free variables or terms being closed.

For the fully spelled-out details, see the accompanying Coq sources. As already mentioned, it is not important to understand all these details; the important part is to understand the intuition behind the De Bruijn representation and how we have to update our type system and logical relations to work with it.

3 System F: Polymorphism and Existential Types

We extend the STLC with polymorphism and existential types. Polymorphism is widespread in modern programming languages (often under the label “generic types”), while existential types serve as a way of creating data abstraction, with a rough correspondence to modules in ML-like languages and to interfaces in object-oriented languages.

3.1 System F

Type Variables	$\alpha, \beta \quad \dots$
Types	$A, B ::= \dots \mid \forall \alpha. A \mid \exists \alpha. A \mid \alpha$
Type Variable Contexts	$\Delta ::= \emptyset \mid \Delta, \alpha$
Source Terms	$E ::= \dots \mid \Lambda \alpha. E \mid E \langle A \rangle$ $\mid \text{pack } \exists \alpha. B \text{ from } A \text{ with } E \mid \text{unpack } E \text{ as } \alpha \text{ with } x \text{ in } E'$
Runtime Terms	$e ::= \dots \mid \Lambda. e \mid e \langle \rangle \mid \text{pack } e \mid \text{unpack } e \text{ as } x \text{ in } e'$
(Runtime) Values	$v ::= \dots \mid \Lambda. e \mid \text{pack } v$
Evaluation Contexts	$K ::= \dots \mid K \langle \rangle \mid \text{pack } K \mid \text{unpack } K \text{ as } x \text{ in } e$

Contextual operational semantics

$$e_1 \rightsquigarrow_b e_2$$

$$\begin{array}{ll} \text{BIGBETA} & \text{UNPACK} \\ (\Lambda. e) \langle \rangle \rightsquigarrow_b e & \text{unpack } (\text{pack } v) \text{ as } x \text{ in } e \rightsquigarrow_b e[v/x] \end{array}$$

The existing properties of the operational semantics, in particular the lifting lemma (Lemma 1), still hold in the extended calculus.

Because we now deal with type variables, we have to deal with a new kind of contexts: *type variable contexts*. The typing judgments will now carry both a type variable context and “normal” variable context. All the existing typing rules remain valid, with the type variable context being the same in all premises and the conclusion of the typing rules. However, for the typing rule for lambdas, we have to make sure that the argument type is actually well-formed in the current typing context: types are well-formed if they do not contain any free variables.

Free Variables (in Types)

$$\text{fv}(A)$$

$$\begin{aligned} \text{fv}(\text{int}) &:= \emptyset \\ \text{fv}(A \rightarrow B) &:= \text{fv}(A) \cup \text{fv}(B) \\ \text{fv}(\alpha) &:= \{\alpha\} \\ \text{fv}(\forall \alpha. A) &:= \text{fv}(A) \setminus \{\alpha\} \\ \text{fv}(\exists \alpha. A) &:= \text{fv}(A) \setminus \{\alpha\} \end{aligned}$$

Type Well-Formedness

$$\Delta \vdash A$$

$$\frac{\text{fv}(A) \subseteq \Delta}{\Delta \vdash A}$$

Church-style typing

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

$$\begin{array}{c}
\text{LAM} \\
\frac{\Delta \vdash A \quad \Delta; \Gamma, x : A \vdash E : B}{\Delta; \Gamma \vdash \lambda x : A. E : A \rightarrow B} \\
\text{APP} \\
\frac{\Delta; \Gamma \vdash E_1 : A \rightarrow B \quad \Delta; \Gamma \vdash E_2 : A}{\Delta; \Gamma \vdash E_1 E_2 : B} \\
\text{BIGLAM} \\
\frac{\Delta, \alpha; \Gamma \vdash E : A}{\Delta; \Gamma \vdash \Lambda \alpha. E : \forall \alpha. A} \\
\text{BIGAPP} \\
\frac{\Delta, \Gamma \vdash E : \forall \alpha. B \quad \Delta \vdash A}{\Delta; \Gamma \vdash E \langle A \rangle : B[A/\alpha]} \\
\text{PACK} \\
\frac{\Delta \vdash A \quad \Delta; \Gamma \vdash E : B[A/\alpha]}{\Delta; \Gamma \vdash \text{pack } \exists \alpha. B \text{ from } A \text{ with } E : \exists \alpha. B} \\
\text{UNPACK} \\
\frac{\Delta; \Gamma \vdash E : \exists \alpha. B \quad \Delta, \alpha; \Gamma, x : B \vdash E' : C \quad \Delta \vdash C}{\Delta; \Gamma \vdash \text{unpack } E \text{ as } \alpha \text{ with } x \text{ in } E' : C}
\end{array}$$

Curry-style typing

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

$$\begin{array}{c}
\text{LAM} \\
\frac{\Delta \vdash A \quad \Delta; \Gamma, x : A \vdash e : B}{\Delta; \Gamma \vdash \lambda x : A. e : A \rightarrow B} \\
\text{APP} \\
\frac{\Delta; \Gamma \vdash e_1 : A \rightarrow B \quad \Delta; \Gamma \vdash e_2 : A}{\Delta; \Gamma \vdash e_1 e_2 : B} \\
\text{BIGLAM} \\
\frac{\Delta, \alpha; \Gamma \vdash e : A}{\Delta; \Gamma \vdash \Lambda. e : \forall \alpha. A} \\
\text{BIGAPP} \\
\frac{\Delta, \Gamma \vdash e : \forall \alpha. B \quad \Delta \vdash A}{\Delta; \Gamma \vdash e \langle \rangle : B[A/\alpha]} \\
\text{PACK} \\
\frac{\Delta \vdash A \quad \Delta; \Gamma \vdash e : B[A/\alpha]}{\Delta; \Gamma \vdash \text{pack } e : \exists \alpha. B} \\
\text{UNPACK} \\
\frac{\Delta; \Gamma \vdash e : \exists \alpha. B \quad \Delta, \alpha; \Gamma, x : B \vdash e' : C \quad \Delta \vdash C}{\Delta; \Gamma \vdash \text{unpack } e \text{ as } x \text{ in } e' : C}
\end{array}$$

3.2 Encoding Data Types

System F allows us to encode standard data types using universal types. Specifically, we will be looking at some *Scott encodings*. In [Section 1.2](#), we used a Scott encoding to encode natural numbers in the untyped lambda-calculus; here we consider other, non-recursively defined data types.

The empty type The Scott encoding of the *empty type* (or *void type*), also written **0**, looks as follows:

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

This type has no inhabitants. If you don't believe us, just try writing a term of that type!

A function type $A \rightarrow \mathbf{0}$ indicates a function that *never* returns. Usually this means that the function might throw an exception, abort the program, or run into an infinite loop—but neither of these are possible in System F (we will show termination of this calculus later). For our language, we can actually be sure that such a function cannot even be called.

The unit type This is a Scott encoding of the *unit type*, also written **1**, looks as follows:

$$\begin{aligned}
\mathbf{1} &:= \forall \alpha. \alpha \rightarrow \alpha \\
() &:= \Lambda \alpha. \lambda x : \alpha. x
\end{aligned}$$

This type has exactly one inhabitant: the polymorphic identity function. After all, what can the function do? It has to return something of type α , and it knows absolutely nothing about that type, so the only option is to return the variable x .

A function type $A \rightarrow \mathbf{1}$ indicates a function that returns no information. Usually this means that the function might have side-effects, such as changing some global state—but System F is a pure language, there are no side-effects. So in System F, such a function is never worth calling, it cannot do anything. (Unfortunately, in many languages, the keyword `void` is used to indicate a function that returns no information. This is of course quite confusing to anyone who comes in with some type theory background, where the void type refers to $\mathbf{0}$, not $\mathbf{1}$.)

Booleans

$$\begin{aligned}\text{bool} &:= \forall \alpha. \alpha \rightarrow \alpha \rightarrow \alpha \\ \text{true} &:= \Lambda \alpha. \lambda t : \alpha. \lambda f : \alpha. t \\ \text{false} &:= \Lambda \alpha. \lambda t : \alpha. \lambda f : \alpha. f \\ \text{if}_C e \text{ then } e_1 \text{ else } e_2 &:= (e \langle \mathbf{1} \rightarrow C \rangle (\lambda _ . \mathbf{1}. e_1) (\lambda _ . \mathbf{1}. e_2)) () \\ \text{if } e \text{ then } e_1 \text{ else } e_2 &:= (e \langle \rangle (\lambda _ . e_1) (\lambda _ . e_2)) ()\end{aligned}$$

This type has exactly two inhabitants: we can either return the first α , or the second α . It is sometimes called $\mathbf{2}$, but more commonly referred to as **bool**.

The conditional exists in two versions, as a source term with type annotation C and as a runtime term without type annotation. The type C 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 ensure that they do not get evaluated before a branch is chosen.

We can show that the runtime versions of these terms behave as expected. For instance:

Statement 31. $\text{if false then } e_1 \text{ else } e_2 \rightsquigarrow^* e_2$.

Proof.

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

□

Product types Scott encodings work not only for simple base types, they also work for compound types. For instance, given two arbitrary types A and B , we can encode their product as follows:

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

Note that when constructing a pair, we explicitly let-bind the two arguments. This is because we want them to be evaluated immediately when the pair is constructed, and not just later when a projection is applied. Let-bindings are a convenient short-hand that is defined as follows:

$$\text{let } x = e_1 \text{ in } e_2 := (\lambda x. e_2) e_1$$

Due to our call-by-value semantics, this forces e_1 to be evaluated to a value before continuing with e_2 .

The reason that we call this an encoding of products is that the following reduction properties hold:

$$\pi_1 \langle v_1, v_2 \rangle \rightsquigarrow^* v_1 \qquad \pi_2 \langle v_1, v_2 \rangle \rightsquigarrow^* v_2$$

The following typing rules can also be shown:

$$\frac{\Delta; \Gamma \vdash e_1 : A \quad \Delta; \Gamma \vdash e_2 : B}{\Delta; \Gamma \vdash \langle e_1, e_2 \rangle : A \times B} \quad \frac{\Delta; \Gamma \vdash e : A \times B}{\Delta; \Gamma \vdash \pi_1 e : A} \quad \frac{\Delta; \Gamma \vdash e : A \times B}{\Delta; \Gamma \vdash \pi_2 e : B}$$

Together this demonstrates that our pairs behave like proper pairs both in the operational semantics and in the type system.

General Scott encoding In general, assume we have a data type T with constructors C_1, C_2, \dots, C_n defined inductively as follows (using Coq-style syntax):²

```
Inductive T :=
| C1 : A1 → A2 → ... → T
| C2 : B1 → B2 → ... → T
⋮
| Cn : X1 → X2 → ... → T.
```

Then the type of the corresponding Scott encoding looks as follows:

$$T := \forall \alpha. (A_1 \rightarrow A_2 \rightarrow \dots \rightarrow \alpha) \rightarrow (B_1 \rightarrow B_2 \rightarrow \dots \rightarrow \alpha) \rightarrow \dots \rightarrow (X_1 \rightarrow X_2 \rightarrow \dots \rightarrow \alpha) \rightarrow \alpha$$

This polymorphic function takes n arguments, one for each constructor, and returns an α . Each of these arguments takes exactly the argument types of the inductive constructors, and returns *alpha*.

The intuition for why this type corresponds to the inductive type is that an element of T must *somehow* produce a value of α without having any clue what α is. All it can do is call one of these functions. So it has to pick a function, *i.e.*, pick a constructor, and then pass to that function values that correspond to the types of the argument of that constructor. That's exactly the same data content as an element of T in Coq!

To show how the encoded constructors work, we give the (partially type-erased) definition of the Scott-encoded C_1 :

$$C_1 := \lambda a_1 : A_1. \lambda a_2 : A_2. \dots \Lambda \alpha. \lambda c_1, c_2, \dots, c_n. c_1 a_1 a_2 \dots$$

You can convince yourself that this term has type $A_1 \rightarrow A_2 \rightarrow \dots \rightarrow T$.

The only other ingredient we need to complete the encoding is case distinction. So we encode:

```
match e with
| C1 a1 a2 ... => e1
| C2 b1 b2 ... => e2
⋮
| Cn x1 x2 ... => en
end
```

²In type theory, “constructor” refers to the terms that *construct* values of inductive types. You have already seen quite a few constructors, for instance: `0`, `S` (for type `nat`), `true`, `false` (for type `bool`), `None`, `Some` (for type `option`).

Note that the a variables are free in e_1 , and so on. We assume that the entire match expression has type U . In Scott encoding, this term becomes:

$$e \langle U \rangle (\lambda a_1 : A_1. \lambda a_2 : A_2. \dots e_1) (\lambda b_1 : B_1. \lambda b_2 : B_2. \dots e_2) \dots (\lambda x_1 : X_1. \lambda x_2 : X_2. \dots e_n)$$

You can convince yourself that this has the right type: if e has type T , then we are instantiating the α with U . Now match arm e_1 has the a_i variables in scope at their appropriate type, and it must return something of type U , and similar for the other arms. Finally, the entire expression has type U , as it should.

In fact, when working with System F we do not need any primitive types at all. This is very different from the simple type system we considered in [Section 1](#). If we remove the base type `int`, that type system just collapses entirely: there are no types at all! Function types can only be formed when we already have constructed an input and output type for the function.

In contrast, in System F we can encode data types without having any primitive type. Everything we needed for the above encodings is the following grammar of types:

$$\text{Types } A, B ::= A \rightarrow B \mid \forall \alpha. A \mid \alpha$$

This core calculus is what is typically called “System F”; it does not usually contain integers or existential types. They can both be encoded!

However, to encode natural numbers (and then integers), we cannot use Scott encoding any more: while the Scott encoding of natural numbers works fine in the untyped lambda calculus, it cannot be given a type in System F. Instead, there exists an alternative encoding called *Church encoding* that can be used to encode natural numbers in a typed manner in System F. (We will not discuss Church encoding in this lecture.)

All this goes to show that while the grammar of types in System F looks deceptively simple, it is actually a really expressive type system.

3.3 Data Abstraction with Existential Types

We have seen how universal types can be used in System F. The other new feature in System F are *existential types* $\exists \alpha. A$. Existential types can be encoded using universal types via a Scott encoding, but the types are interesting enough that they deserve being studied as a primitive type in their own right.

The key intuition behind a type like $\exists \alpha. A$ is that it provides a form of *data abstraction*: given a value of type $\exists \alpha. A$, we do not know what the actual underlying type for α is, so the operations provided by A describe everything we can do with a α . This is very similar to a type where all fields are private, and A describes the public functions. For instance, $\exists \alpha. \alpha \times (\alpha \rightarrow \text{int}) \times (\alpha \rightarrow \alpha)$ might describe the type of a counter: the first component of the pair, of type α , provides a counter that is initialized with value 0; the second component, of type $\alpha \rightarrow \text{int}$, returns the current value of the counter; and the third component, of type $\alpha \rightarrow \alpha$, increments the counter by 1 and returns the new value. (We can use either the Scott encoding of products introduced in the previous subsection, or an extension of System F with native support for pairs along the lines of what you saw on the exercise sheets. Products are left-associative, so $A \times B \times C$ is short for $(A \times B) \times C$.)

Here is a small function that takes such a counter, increments the counter twice, and returns its final value:

```
counter_client :=  $\lambda c : \exists \alpha. \alpha \times (\alpha \rightarrow \text{int}) \times (\alpha \rightarrow \alpha).$ 
  unpack  $c$  as  $\alpha$  with  $c'$  in
    let  $x : \alpha = \pi_1 (\pi_1 c')$  in
    let  $y : \alpha = \pi_2 c' x$  in
    let  $y' : \alpha = \pi_2 c' y$  in
     $\pi_2 (\pi_1 c') y'$ 
```

The type of c' in this example is $\alpha \times (\alpha \rightarrow \text{int}) \times (\alpha \rightarrow \alpha)$.

As we can see, packaging up various operations in a pair and then using the pair projections is getting a bit tedious and also makes the code rather hard to read. So we will introduce some syntactic sugar that lets us write *record types*:

$$\{\text{name1} : A, \text{name2} : B, \dots, \text{nameN} : C\} := ((A \times B) \times \dots) \times C$$

For instance, we would write $\{\text{init} : \alpha, \text{get} : \alpha \rightarrow \text{int}, \text{inc} : \alpha \rightarrow \alpha\}$ as sugar for $\alpha \times (\alpha \rightarrow \text{int}) \times (\alpha \rightarrow \alpha)$.

Terms of that type are defined as follows:

$$\{\text{name1} = e_1, \text{name2} = e_2, \dots, \text{nameN} = e_n\} := \langle \langle e_1, e_2 \rangle, \dots \rangle, e_n \rangle$$

We then use projection functions to access the corresponding element of the record: when e has type $\{\text{name1} : A, \text{name2} : B, \dots, \text{nameN} : C\}$, we define:

$$\begin{aligned} e.\text{name1} &:= \pi_1 (\dots (\pi_1 e)) \\ e.\text{name2} &:= \pi_2 (\dots (\pi_1 e)) \\ &\vdots \\ e.\text{nameN} &:= \pi_N e \end{aligned}$$

With that, we can write the counter in a much more readable way:

$$\begin{aligned} \text{counter} &:= \exists \alpha. \{\text{init} : \alpha, \text{get} : \alpha \rightarrow \text{int}, \text{inc} : \alpha \rightarrow \alpha\} \\ \text{counter_client} &:= \lambda c : \text{counter}. \text{unpack } c \text{ as } \alpha \text{ with } c \text{ in} \\ &\quad \text{let } x = c.\text{init} \text{ in} \\ &\quad \text{let } x = c.\text{inc } x \text{ in} \\ &\quad \text{let } x = c.\text{inc } x \text{ in} \\ &\quad c.\text{get } x \end{aligned}$$

Here we also shadow the original c with the unpacked c , since there is no reason to access the original c in the scope of the `unpack`. For the same reason we shadow the counter variable x .

Our `counter_client` can be written without knowing anything about how the actual value of the counter is represented. That means we can now give multiple implementations, and the client will work with all of them. Here is the “obvious” implementation that just uses an integer:

$$\begin{aligned} \text{counter_imp} &:= \text{pack counter from int with} \\ &\quad \{\text{init} = \bar{0}, \text{get} = \lambda x : \text{int}. x, \text{inc} = \lambda x : \text{int}. x + \bar{1}\} \end{aligned}$$

However we can also do something silly, like store the counter relative to some arbitrary base value, and as long as we also adjust the public “getter” nobody will be able to tell the difference:

$$\begin{aligned} \text{counter_imp42} &:= \text{pack counter from int with} \\ &\quad \{\text{init} = \overline{42}, \text{get} = \lambda x : \text{int}. x + \overline{-42}, \text{inc} = \lambda x : \text{int}. x + \bar{1}\} \end{aligned}$$

We can even change the value representation completely, like storing a pair of two integers that redundantly both contain the counter value:

$$\begin{aligned} \text{counter_imp_pair} &:= \text{pack counter from int} \times \text{int with} \\ &\quad \{\text{init} = \langle \bar{0}, \bar{0} \rangle, \text{get} = \lambda x : \text{int} \times \text{int}. \pi_1 x, \text{inc} = \lambda x : \text{int} \times \text{int}. \langle \pi_1 x + \bar{1}, \pi_2 x + \bar{1} \rangle\} \end{aligned}$$

Our `counter_client` will behave exactly the same on all of these implementations; it is not able to tell the difference between the different representations. In that sense, existential types establish a strict *abstraction barrier*: nothing outside the `pack` can tell what is going on inside. This can be used to encode modules and sealing in System F, and can also serve as a representation of private fields that cannot be accessed from outside the type’s definition.

3.4 System F with De Bruijn representation

The terms and types presented above use regular named variables both in terms (x) and in types (α). However, as discussed in [Section 2](#), formalizing a proper capture-avoiding substitution with named variables in Coq is cumbersome; in our Coq formalization, we really want to use De Bruijn representation. Therefore, we follow a two-track system: in these notes, we will pretend to work in ordinary System F with named binders, because it significantly improves readability. However, since doing the same in Coq is not an option, we will use a version System F with De Bruijn types in Coq. In the following, we make precise what System F with De Bruijn types looks like.

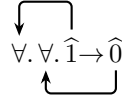
Specifically, we use De Bruijn indices both to represent variables in terms and variables in types. De Bruijn indices in types work exactly the same as they do in terms. Both \forall and \exists bind a variable, so they both “count” (literally) for De Bruijn indices: an index of $\hat{3}$ means the variable refers to the binder (\forall or \exists) up the syntax tree that one finds after skipping 3 other binders. Here are some examples:

- The type $\forall \alpha. \alpha \rightarrow \alpha$ is represented as



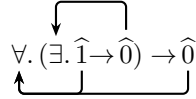
When looking for the binding occurrence of the variable α , we have to skip 0 binders, as the single type quantifier binds it.

- The type $\forall \alpha, \beta. \alpha \rightarrow \beta$ is represented as



When looking for the binding occurrence of α , we have to skip the quantifier introducing β .

- The type $\forall \alpha. (\exists \beta. \alpha \rightarrow \beta) \rightarrow \alpha$ is represented as



Note that the first occurrence of α is replaced by $\hat{1}$, as it sits below an additional quantifier, while the second occurrence is not below further binders and hence becomes $\hat{0}$.

The formal definition of the syntax looks as follows:

Types	$A, B ::= \dots \mid \forall. A \mid \exists. A \mid \hat{\alpha}$
Type Variable Contexts	$\Delta \in \mathbb{N}$
Runtime Terms	$e ::= \dots \mid \Lambda. e \mid e \langle \rangle \mid \text{pack } e \mid \text{unpack } e \text{ in } e'$
(Runtime) Values	$v ::= \dots \mid \Lambda. e \mid \text{pack } v$
Evaluation Contexts	$K ::= \dots \mid K \langle \rangle \mid \text{pack } K \mid \text{unpack } K \text{ in } e$

Note that type variable contexts Δ are simply a natural number: without names, the only information we require is *how many* type variables there are. Both λ and `unpack` bind a variable, so De Bruijn indices count how many of either of them are skipped until reaching the binder the refer to.

Contextual operational semantics

$$e_1 \rightsquigarrow_b e_2$$

BIGBETA

$$(\Lambda. e) \langle \rangle \rightsquigarrow_b e$$

UNPACK

$$\text{unpack } (\text{pack } v) \text{ in } e \rightsquigarrow_b e[v / \]$$

Type Well-Formedness

 $\Delta \vdash A$

$$\begin{array}{c}
 \text{WF-INT} \\
 \Delta \vdash \text{int}
 \end{array}
 \quad
 \frac{\text{WF-LAM} \quad \Delta \vdash A \quad \Delta \vdash B}{\Delta \vdash A \rightarrow B}
 \quad
 \frac{\text{WF-TVAR} \quad n < \Delta}{\Delta \vdash \hat{n}}
 \quad
 \frac{\text{WF-TFORALL} \quad 1 + \Delta \vdash A}{\Delta \vdash \forall. A}
 \quad
 \frac{\text{WF-TEXISTS} \quad 1 + \Delta \vdash A}{\Delta \vdash \exists. A}$$

In the typing judgments, we have to take a bit of care when introducing new type variables to the context. We introduce new type variables when we move underneath a type binder (*i.e.*, \exists . or \forall .). As with substitution, when we move underneath a binder, we must be careful to not screw up the mapping between free variables and the binders they refer to (in this case in the context Γ). That is, the variable $\hat{0}$ will now point to the binder we just moved under, and all other variables have to be increased by one. We do the increase with the operation $A[\uparrow]$ (defined in the same way as $e[\uparrow]$ in [Section 2.1](#)) and lift it to typing contexts $\Gamma[\uparrow]$ pointwise.

Curry-style typing

 $\Delta; \Gamma \vdash e : A$

$$\begin{array}{c}
 \dots \\
 \frac{\text{LAM} \quad \Delta \vdash A \quad \Delta; \Gamma, A \vdash e : B}{\Delta; \Gamma \vdash \lambda. e : A \rightarrow B}
 \quad
 \frac{\text{APP} \quad \Delta; \Gamma \vdash e_1 : A \rightarrow B \quad \Delta; \Gamma \vdash e_2 : A}{\Delta; \Gamma \vdash e_1 e_2 : B}
 \\
 \frac{\text{BIGLAM} \quad 1 + \Delta; \Gamma[\uparrow] \vdash e : A}{\Delta; \Gamma \vdash \Lambda. e : \forall. A}
 \quad
 \frac{\text{BIGAPP} \quad \Delta, \Gamma \vdash e : \forall. B \quad \Delta \vdash A}{\Delta; \Gamma \vdash e \langle \rangle : B[A/ \]}
 \\
 \frac{\text{PACK} \quad \Delta \vdash A \quad \Delta; \Gamma \vdash e : A[B/ \]}{\Delta; \Gamma \vdash \text{pack } e : \exists. A}
 \quad
 \frac{\text{UNPACK} \quad \Delta; \Gamma \vdash e : \exists. A \quad 1 + \Delta; \Gamma[\uparrow], A \vdash e' : B[\uparrow] \quad \Delta \vdash B}{\Delta; \Gamma \vdash \text{unpack } e \text{ in } e' : B}
 \end{array}$$

Now that we have defined this, we immediately go back to the named representation to keep things more readable.

3.5 Type Safety

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

For progress, as before we start by stating what the canonical forms of our new types are:

Lemma 32 (Canonical forms). *If $\vdash v : A$, then:*

- if $A = \forall \alpha. B$ for some B , then $v = \Lambda. e$ for some e
- if $A = \exists \alpha. B$ for some B , then $v = \text{pack } v'$ for some v'

Proof. By inversion. □

Theorem 33 (Progress). *Theorem 10 remains valid: If $\vdash e : A$, then e is progressive.*

Proof. Remember we are doing induction on $\vdash e : A$. The induction statement changes slightly:

$$IH(\Delta, \Gamma, e, A) := \Delta = \emptyset \wedge \Gamma = \emptyset \Rightarrow \text{progressive}(e)$$

As before, we have to generalize over Δ and Γ to even be able to do induction, but we explicitly keep around the information that the contexts are empty. This is crucial to make the proof go through. In the cases below, we have already replaced all the Δ and Γ by \emptyset again.

The new cases are:

- **BIGLAM**: $e = \Lambda. e_1$ and $A = \forall \alpha. B$.
Have: $\alpha; \emptyset \vdash e : B$. To Show: $\Lambda. e_1$ is progressive.
 $\Lambda. e_1$ is a value and hence progressive.
- **BIGAPP**: $e = e_1 \langle \rangle$.
Have $\vdash e_1 : \forall \alpha. B$ and $IH(\emptyset, \emptyset, e_1, \forall \alpha. B)$ and $A = B[C/\alpha]$ and $\vdash C$. To Show: $e_1 \langle \rangle$ is progressive.
By IH, e_1 is progressive. We distinguish two cases:
 - e_1 is a value. By canonical forms, $e_1 = \Lambda. e'$ for some e' .
Then $e_1 \langle \rangle \rightsquigarrow e'$ by **BIGBETA** and **CTX**, so we are done.
 - $e_1 \rightsquigarrow e'_1$ for some e'_1 .
Then $e_1 \langle \rangle \rightsquigarrow e'_1 \langle \rangle$ by **Lemma 1** with $K := \bullet \langle \rangle$.
- **PACK, UNPACK**: See exercise sheet. □

For preservation, we again need some rather technical lemmas showing that substitution does not affect typing. Now that we have two kinds of substitution (type substitution and term substitution), we also need two substitution lemmas.

Lemma 34 (Type Substitution). *If $\Delta, \alpha; \Gamma \vdash e : A$ and $\Delta \vdash B$, then $\Delta; \Gamma[B/\alpha] \vdash e : A[B/\alpha]$.*

Lemma 35 (Term Substitution). *If $\Delta; \Gamma, x : A \vdash e : B$ and $\Delta; \Gamma \vdash e' : A$, then $\Delta; \Gamma \vdash e[e'/x] : B$.*

From this, we can show base preservation.

Lemma 36 (Base preservation). *Lemma 13 remains valid: If $\vdash e : A$ and $e \rightsquigarrow_b e'$, then $\vdash e' : A$.*

Proof. Remember that we are doing inversion on $e \rightsquigarrow_b e'$. The new cases are:

- **BIGBETA**, $e = (\Lambda. e_1) \langle \rangle$ and $e' = e_1$.
To show: $\vdash e_1 : A$.
By inversion on $\vdash e : A$ we have $\vdash \Lambda. e_1 : \forall \alpha. B$ and $A = B[C/\alpha]$ and $\vdash C$.
By another inversion on typing of $\Lambda. e_1$, we have $\alpha; \emptyset \vdash e_1 : B$.
By type substitution we obtain $\vdash e_1 : B[C/\alpha]$, so we are done.
- **UNPACK**: See exercise sheet. □

Technically speaking, we must update the composition and decomposition lemmas to handle the new evaluation contexts. However, we will omit this technical proof, it has no interesting proof content.

The proof of the actual preservation lemma then proceeds exactly as before:

Theorem 37 (Preservation). *Theorem 16 remains valid: If $\vdash e : A$ and $e \rightsquigarrow e'$, then $\vdash e' : A$.*

Therefore, System F is type safe.

Corollary 38 (Type Safety). *If $\vdash e : A$, then e is safe.*

Overall, the type safety proof for System F went pretty much exactly as before. Type safety proofs by progress and preservation are boring: the same recipe works for a wide range of languages, and once one has done this a few times, one generally does not have to think very hard to get the setup right. This is a feature! Progress and preservation was specifically *designed* to make type safety proofs boring, so that we could get type safety proofs for a wide range of languages.

However, this also shows that type safety doesn't actually interact deeply with what is going on in the language. The increase in expressivity from the simply-typed λ -calculus to System F is *huge*, and that is not reflected in type safety at all. In contrast, updating our proof of termination for System F will require some clever new ideas.

3.6 Termination

Expressivity and termination are acting against each other: the more programs one can write, the more likely it is that one can also write a non-terminating program. However, while System F is very expressive, it is still terminating, and that's what we are going to show in this subsection.

We extend the semantic model to handle universal and existential types. The naive approach would be to interpret $\forall\alpha. A$ as something like

$$\mathcal{V}[\forall\alpha. A] := \{\Lambda. e \mid \forall B. e \in \mathcal{E}[A[B/\alpha]]\}$$

However, that would not work! The type $A[B/\alpha]$ is potentially *bigger* than $\forall\alpha. A$, so the mutual structural recursion we are using to define \mathcal{V} and \mathcal{E} is no longer sufficient.

Fundamentally the problem is that we are quantifying over all *syntactic* types B , but we can't interpret all of these types since that includes even the type whose interpretation we are defining right now!³ The solution to the problem is an ingenious trick with profound consequences: rather than quantifying over all *syntactic* types, we quantify over all *semantic* types. A *semantic type* is what $\mathcal{V}[A]$ produces: a set of values. We will denote semantic types with τ .

Semantic Types

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

$$\mathit{SemType} := \mathbb{P}(\mathit{Val})$$

On the one hand, this seems to make our job a lot harder: there are *many* more semantic types than syntactic types. Every syntactic type has a corresponding semantic type defined by \mathcal{V} , but not vice versa. For instance, the set of all even integers is a valid semantic type. The set of all integers that encode Turing machines that halt on all inputs is also a valid semantic type. (Formally: the set of syntactic types is countable, but the set of semantic types is not.) On the other hand, this does break our recursive knot, so we can complete the definition—and as we will see, this extra power from there being so many semantic types will be very useful.

To make this work, we extend the value relation with a semantic type substitution δ that maps each type variable to the corresponding semantic type. We assume that this is a total substitution; for unused type variables δ does not matter so it can be initialized with an arbitrary semantic type (like the set of all values, or the empty set). We also need to extend the big-step semantics to our new syntactic constructs, since that is used in the expression relation.

Big-Step Semantics

$$e \Downarrow v$$

$$\begin{array}{c} \dots \quad \text{BIGLAMBDA} \quad \frac{}{\Lambda. e \Downarrow \Lambda. e} \quad \text{BIGAPP} \quad \frac{e_1 \Downarrow \Lambda. e \quad e \Downarrow v}{e_1 \langle \rangle \Downarrow v} \quad \text{PACK} \quad \frac{e \Downarrow v}{\text{pack } e \Downarrow \text{pack } v} \quad \text{UNPACK} \quad \frac{e \Downarrow \text{pack } v \quad e'[v/x] \Downarrow v'}{\text{unpack } e \text{ as } x \text{ in } e' \Downarrow v'} \end{array}$$

Value Relation

$$\mathcal{V}[A]_\delta$$

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

³The fact that $\forall\alpha. A$ quantifies over all types including itself is called *impredicativity*. It is a powerful feature in logic and type systems, but also a great challenge to formal soundness proofs. The next section will show how quickly impredicativity can have surprising consequences.

Expression Relation

$$\mathcal{E}[A]_\delta$$

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

Context Relation

$$\mathcal{G}[\Gamma]_\delta$$

$$\begin{array}{c} \text{CREL-EMPTY} \\ \emptyset \in \mathcal{G}[\Gamma]_\delta \end{array}$$

$$\begin{array}{c} \text{CREL-ELEM} \\ \frac{\gamma \in \mathcal{G}[\Gamma]_\delta \quad v \in \mathcal{V}[A]_\delta}{\gamma[x \mapsto v] \in \mathcal{G}[\Gamma, x : A]_\delta} \end{array}$$

Semantic Typing

$$\Delta ; \Gamma \models e : A$$

$$\Delta ; \Gamma \models e : A := \forall \delta. \forall \gamma \in \mathcal{G}[\Gamma]_\delta. e[\gamma] \in \mathcal{E}[A]_\delta$$

As before, the next step is to prove the compatibility lemmas. The existing lemmas need to be adapted to account for the type variable substitution δ , but that is straightforward. We show the new cases for the universal types; the cases for existential types will be an exercise.

To make the compatibility lemmas work, we need some rather boring lemmas about type substitution. We do not give their proofs here.

Lemma (Boring Lemma 1). *If δ_1 and δ_2 agree on the free type variables of Γ and A , then the value, expression, and context relations are the same with both type interpretations:*

$$\begin{aligned} \mathcal{V}[A]_{\delta_1} &= \mathcal{V}[A]_{\delta_2} \\ \mathcal{E}[A]_{\delta_1} &= \mathcal{E}[A]_{\delta_2} \\ \mathcal{G}[\Gamma]_{\delta_1} &= \mathcal{G}[\Gamma]_{\delta_2} \end{aligned}$$

Lemma (Boring Lemma 2). *Substituting a syntactic type and then applying the semantic interpretation is equivalent to first interpreting that type and then adding it to the semantic type substitution:*

$$\begin{aligned} \mathcal{V}[B[A/\alpha]]_\delta &= \mathcal{V}[B]_{\delta[\alpha \mapsto \mathcal{V}[A]_\delta]} \\ \mathcal{E}[B[A/\alpha]]_\delta &= \mathcal{E}[B]_{\delta[\alpha \mapsto \mathcal{V}[A]_\delta]} \end{aligned}$$

Now we can finally tackle the new cases for the termination proof.

Lemma 39 (Compatibility with **BIGLAM**). *If $\Delta, \alpha ; \Gamma \models e : A$ then $\Delta ; \Gamma \models \Lambda. e : \forall \alpha. A$.*

Proof. We may assume some δ and some $\gamma \in \mathcal{G}[\Gamma]_\delta$ and have to show $\Lambda. e[\gamma] \in \mathcal{E}[\forall \alpha. A]_\delta$.

By value inclusion (**Lemma 19** still holds), it suffices to show $\Lambda. e[\gamma] \in \mathcal{V}[\forall \alpha. A]_\delta$.

So suppose $\tau \in \text{SemType}$; we have to show $e[\gamma] \in \mathcal{E}[A]_{\delta[\alpha \mapsto \tau]}$.

Applying $\Delta, \alpha ; \Gamma \models e : A$ with type substitution $\delta' := \delta[\alpha \mapsto \tau]$, it suffices to show $\gamma \in \mathcal{G}[\Gamma]_{\delta'}$.

Since α cannot be free in Γ , this is equivalent to $\gamma \in \mathcal{G}[\Gamma]_\delta$, so we are done by Boring Lemma 1. \square

(Intuitively, α cannot be free in Γ since in $\Delta ; \Gamma \models \Lambda. e : \forall \alpha. A$, variable α is only bound in A but not in Γ . If there is any other α in Δ , we can rename it to avoid the collisions. To actually make this formal requires extra work, similar to the bookkeeping we had to do in **Section 1.5** to ensure that values are closed. The proper formalization with De Bruijn variables avoids that work, as can be seen in the Coq formalization.)

Lemma 40 (Compatibility with **BIGAPP**). *If $\Delta ; \Gamma \models e : \forall \alpha. B$ then $\Delta ; \Gamma \models e \langle \rangle : B[A/\alpha]$.*

Proof. We may assume some δ and some $\gamma \in \mathcal{G}[\Gamma]_\delta$ and have to show $e[\gamma] \langle \rangle \in \mathcal{E}[B[A/\alpha]]_\delta$.

From the assumption, there exists v such that $e[\gamma] \downarrow v$ and $v \in \mathcal{V}[\forall \alpha. B]_\delta$.

By the definition of \mathcal{V} , it follows that $v = \Lambda. e'$ and $\forall \tau \in \text{SemType}. e' \in \mathcal{E}[B]_{(\delta, \alpha \mapsto \tau)}$ for some e' .

Pick $\tau := \mathcal{V}[A]_\delta$. By the previous line and Boring Lemma 2, we have $e' \in \mathcal{E}[B[A/\alpha]]_\delta$.

By the definition of \mathcal{E} , there exists $v' \in \mathcal{V}[B[A/\alpha]]_\delta$ such that $e' \downarrow v'$.

By big-step rule **BIGAPP**, it follows that $e[\gamma] \langle \rangle \downarrow v'$, so we are done. \square

Theorem 41 (Semantic Soundness).

Theorem 26 remains valid: If $\Delta ; \Gamma \vdash e : A$, then $\Delta ; \Gamma \models e : A$.

Proof. By induction on $\Delta ; \Gamma \vdash e : A$ and then using the compatibility lemmas. \square

By far the hardest part about this proof was to come up with the idea of modeling the universal type as quantifying over an arbitrary *semantic* type. Once we made that choice, everything fell into place and the proof was largely completed by unfolding definitions and some Boring Lemmas.

It may seem like we “just” proved termination now, but in fact, the proof we have done here entirely subsumes the type safety result from the previous section, as demonstrated by the following lemma:

Lemma 42. *If $e \Downarrow v$, then e is safe.*

Proof Sketch. This result is obtained via the following sequence of lemmas:

- Base reduction is deterministic: if $e \rightsquigarrow_b e_1$ and $e \rightsquigarrow_b e_2$, then $e_1 = e_2$.
- There is at most one way to decompose an expression e into an evaluation context and a hole such that the term in the hole can take a base step: if $e = K_1[e_1] = K_2[e_2]$ and $e_1 \rightsquigarrow_b e'_1$ and $e_2 \rightsquigarrow_b e'_2$, then $K_1 = K_2$ and $e_1 = e_2$.
- Contextual reduction is deterministic: if $e \rightsquigarrow e_1$ and $e \rightsquigarrow e_2$, then $e_1 = e_2$.
- When considering two contextual reduction sequences starting at e , then one of them is a prefix of the other, meaning one of the “end terms” can go on reducing to the other: if $e \rightsquigarrow^* e_1$ and $e \rightsquigarrow^* e_2$, then $e_1 \rightsquigarrow^* e_2$ or $e_2 \rightsquigarrow^* e_1$.
- Big-step reduction implies contextual reduction: if $e \Downarrow v$ then $e \rightsquigarrow^* v$.
- If a term e big-steps to v , then all terms e reduces to can reduce to v : if $e \Downarrow v$ and $e \rightsquigarrow^* e'$, then $e' \rightsquigarrow^* v$.
- If a term reduces to a value, it is progressive: if $e' \rightsquigarrow^* v$, then e' is progressive.

We do not give the proofs of these lemmas here; you can find them in the Coq development for the course.

Lemma 43 (Semantic Type Safety). *If $\models e : A$, then e is safe.*

Proof. We define δ_{emp} by $\delta_{\text{emp}}(\alpha) := \emptyset$ for all α . Instantiating semantic typing of e with $\delta := \delta_{\text{emp}}$ and $\gamma := \emptyset$, we obtain $e \in \mathcal{E}[[A]]$, and thus $e \Downarrow v \in \mathcal{V}[[A]]$. By Lemma 42, this implies that e is safe, so we are done. \square

This proof relied heavily on our language being deterministic—but in fact, that assumption is already pretty much baked into our definition of \mathcal{E} : we just require that there exists *some* value that e reduces to that is a valid value. For a non-deterministic language, we would instead require that *all* reduction sequences starting at e end in a valid value (*i.e.*, they never get stuck or loop forever).

Together with Theorem 41, this implies Corollary 63 (Type Safety). In other words, doing a semantic soundness proof for a language can entirely replace the traditional progress-and-preservation proof: the result obtained via the semantic model is strictly stronger than the result obtained via purely syntactic means. (And it also requires much fewer technical lemmas about substitution.)

3.7 Type Casts Break Termination

In this section, we are going to demonstrate that the termination result we just proved is far from obvious. It may seem like termination is straight-forward, but that is because (a) System F is carefully designed to actually still enforce termination, and (b) we arranged every aspect of this setup to make the proof as nice as possible. It turns out that adding seemingly trivial features that have nothing to do with loops or recursion can break the type system in the sense of letting us write non-terminating programs.

Specifically, for this section we consider adding type casts. The operator $\text{cast } \langle A \rangle \langle B \rangle e_1 e_2$ attempts to cast e_2 of type A to type B . This is checked with a dynamic type comparison: if, at runtime, $A = B$, then the operator returns e_2 ; otherwise it returns the “backup” value given by e_1 which always has type B . The typing rule for cast is as follows:

$$\frac{\Delta ; \Gamma \vdash e_1 : B \quad \Delta ; \Gamma \vdash e_2 : A}{\Delta ; \Gamma \vdash \text{cast } \langle A \rangle \langle B \rangle e_1 e_2 : B}$$

The dynamic behavior of cast is defined by the following base reduction steps:

$$\frac{A = B}{\text{cast } \langle A \rangle \langle B \rangle v_1 v_2 \rightsquigarrow_b v_2} \quad \frac{A \neq B}{\text{cast } \langle A \rangle \langle B \rangle v_1 v_2 \rightsquigarrow_b v_1}$$

(Technically, we have to change runtime terms and the base reduction rules to have types in them. We will not spell out that change here.)

You can convince yourself that in both cases, the type is preserved, so type safety still holds for this calculus.

However, termination no longer holds. To demonstrate that, we will encode the following diverging untyped lambda-term that we already saw in [Section 1.2](#) into System F:

$$\omega := \lambda x. x x \quad \Omega := \omega \omega$$

Similar to before, we will then be able to show that $\Omega \rightsquigarrow^* \Omega$, thus showing that we have constructed an infinite loop.

To do this encoding, we define a type D that you can think of as being a type for *dynamic* objects. The idea is that we can encode an untyped language into a typed language by giving all terms the same type, D , and then providing operations to convert between D and all the more specific types one can actually do something with. In our case, we will need two such operations: $\text{callD } e$ turns a term e of type D into something of type $D \rightarrow D$ (that we can then call like any regular function), and $\text{makeD } e$ turns a term e of type $D \rightarrow D$ into something of type D (that we can then pass around like any other “dynamic” object without remembering its exact type).

Concretely, we define:

$$D := \forall \alpha. \alpha \rightarrow \alpha \quad \text{callD } e := e \langle D \rangle \quad \text{makeD } e := \Lambda \alpha. \text{cast } \langle D \rightarrow D \rangle \langle \alpha \rightarrow \alpha \rangle (\lambda x : \alpha. x) e$$

Convince yourself that these terms indeed have the types described above.

Now we define typed versions of ω and Ω . Both of these have type D .

$$\omega : D := \text{makeD } (\lambda x : D. (\text{callD } x) x) \\ \Omega : D := (\text{callD } \omega) \omega$$

This type-checks because $\lambda x : D. (\text{callD } x) x$ has type $D \rightarrow D$.

Before evaluating these terms, we expand them by unfolding the callD and makeD :

$$\omega = \Lambda \alpha. \text{cast } \langle D \rightarrow D \rangle \langle \alpha \rightarrow \alpha \rangle (\lambda x : \alpha. x) (\lambda x : D. x \langle D \rangle x) \\ \Omega = \omega \langle D \rangle \omega$$

Note, in particular, that ω is a value.

Now consider what happens when we evaluate Ω :

$$\begin{aligned}
\Omega &= (\Lambda\alpha. \text{cast } \langle D \rightarrow D \rangle \langle \alpha \rightarrow \alpha \rangle (\lambda x : \alpha. x) (\lambda x : D. x \langle D \rangle x)) \langle D \rangle \omega \\
&\rightsquigarrow (\text{cast } \langle D \rightarrow D \rangle \langle D \rightarrow D \rangle (\lambda x : D. x) (\lambda x : D. x \langle D \rangle x)) \omega \\
&\rightsquigarrow (\lambda x : D. x \langle D \rangle x) \omega \\
&\rightsquigarrow \omega \langle D \rangle \omega \\
&= \Omega
\end{aligned}$$

The first reduction step specializes the big Λ to D , replacing all the α by D . The second reduction step reduces the **cast**, using the case where $A = B$ since both are $D \rightarrow D$. The final reduction step is just beta reduction. Then we arrive back where we started. We found a well-typed term that loops forever!

Fundamentally, the reason this works is that type quantification in System F is *impredicative*, i.e., in a type like $\forall\alpha. \alpha \rightarrow \alpha$, the α can be replaced by *every* type, including this type itself! That means we can apply a term of type D to itself, making the system highly recursive. Miraculously, in plain System F, this does *not* lead to any non-terminating programs, but just a seemingly tiny extension such as type casts fundamentally changes this equation.

This also goes to show that adding any form of type cast or type switch to a language is a pretty deep change that leads to the loss of some core type theoretical properties. Such operations should only be added after careful consideration.

3.8 Free Theorems

Let us get back to pure System F without problematic features such as type casts. We have successfully proven that all well-typed System F terms terminate. That's a big deal, but there's more! In particular, we can now prove some of the claims we made earlier in [Section 3.2](#) that certain types involving the universal quantifier have no term, or only a limited number of (meaningfully different) terms. This class of theorems was coined “free theorems” by Wadler [5].

We will use the following corollary of semantic soundness for closed terms:

Corollary 44 (Semantic Soundness on Closed Terms).

If $\vdash e : A$, then $e \in \mathcal{E}[\![A]\!]_{\delta_{\text{emp}}}$, where δ_{emp} is defined by $\delta_{\text{emp}}(\alpha) := \emptyset$ for all α .

Proof. By [Theorem 41](#) and instantiating semantic soundness with $\delta := \delta_{\text{emp}}$ and $\gamma := \emptyset$, we have $e[\emptyset] \in \mathcal{E}[\![A]\!]_{\delta_{\text{emp}}}$. By $e[\emptyset] = e$ (the empty substitution changes nothing), we are done. \square

Lemma 45 (Free theorem for $\mathbf{0}$). *We prove that there exists no term closed term e with type $\forall\alpha. \alpha$.*

Proof. Suppose that $\vdash e : \forall\alpha. \alpha$. To show: \perp .

By [Corollary 44](#), we have $e \in \mathcal{E}[\![\forall\alpha. \alpha]\!]_{\delta_{\text{emp}}}$.

Therefore, we have some $v \in \mathcal{V}[\![\forall\alpha. \alpha]\!]_{\delta_{\text{emp}}}$ such that $e \downarrow v$.

Pick $\tau = \emptyset$, then $v = \Lambda. e'$ and $e' \in \mathcal{E}[\![\alpha]\!]_{\delta_{\text{emp}}[\alpha \mapsto \emptyset]}$.

Hence we have some $v' \in \mathcal{V}[\![\alpha]\!]_{\delta_{\text{emp}}[\alpha \mapsto \emptyset]}$ such that $e' \downarrow v'$.

But that is a contradiction, since $\mathcal{V}[\![\alpha]\!]_{\delta_{\text{emp}}[\alpha \mapsto \emptyset]} = \emptyset$. \square

Lemma 46 (Free theorem for $\mathbf{1}$). *We prove that all inhabitants of $\forall\alpha. \alpha \rightarrow \alpha$ are identity functions, in the sense that given a closed term f of that type, for any closed value v we have $f \langle \rangle v \downarrow v$.*

Proof. Suppose $\vdash f : \forall\alpha. \alpha \rightarrow \alpha$. To show: $f \langle \rangle v \downarrow v$.

By [Corollary 44](#), we have $f \downarrow f_v \in \mathcal{V}[\![\forall\alpha. \alpha \rightarrow \alpha]\!]_{\delta_{\text{emp}}}$.

Pick $\tau = \{v\}$. We have $f_v = \Lambda. e'$ and $e' \in \mathcal{E}[\![\alpha \rightarrow \alpha]\!]_{\delta_{\text{emp}}[\alpha \mapsto \tau]}$.

Therefore $e' \downarrow v' \in \mathcal{V}[\![\alpha \rightarrow \alpha]\!]_{\delta_{\text{emp}}[\alpha \mapsto \tau]}$.

From $v \in \tau$, we have $v' = \lambda x. e''$ and $e''[v/x] \in \mathcal{E}[\![\alpha]\!]_{\delta_{\text{emp}}[\alpha \mapsto \tau]}$.

Thus $e''[v/x] \downarrow v'' \in \mathcal{V}[\![\alpha]\!]_{\delta_{\text{emp}}[\alpha \mapsto \tau]}$, and hence $v'' = v$.

Therefore we are done: we have $f \downarrow \Lambda. e'$ and $e' \downarrow \lambda x. e''$ and $e''[v/x] \downarrow v$. \square

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 data type encodings are *faithful* encodings, in the sense that the type only has exactly the inhabitants we want it to have. As an example, we look at the encoding of `bool` values. Remember that `bool` := $\forall \alpha. \alpha \rightarrow \alpha \rightarrow \alpha$. The free theorem for this type looks as follows:

Statement 47 (Free Theorem for `bool` values). *For any value v such that $\vdash v : \text{bool}$, we have that $\forall v_1, v_2. \exists v'. v \langle \rangle v_1 v_2 \downarrow v' \in \{v_1, v_2\}$.*

Statement 47 allows us to say that (if v then v_1 else v_2) $\downarrow v' \in \{v_1, v_2\}$. This result is great, but it is not quite as strong as it could be. Ideally, we would have the following:

Statement 48 (Stronger Theorem for `bool` values). *For any value v such that $\vdash v : \text{bool}$, we have that $(\forall v_1, v_2. v \langle \rangle v_1 v_2 \downarrow v_1)$, or $(\forall v_1, v_2. v \langle \rangle v_1 v_2 \downarrow v_2)$.*

This statement essentially says that every value is either `true` or `false`, which is what characterizes a faithful representation of the Boolean type. The weaker **Statement 47** allows for the same value v to sometimes act like `true` and sometimes act like `false`, depending on the values v_1 and v_2 . In other words, **Statement 47** does not imply that there are only two Boolean values! We need the stronger **Statement 48** for that.

However, our semantic model is not strong enough to prove **Statement 48**. We show this by adding an extension to our language, with which we can build a `bool` value that satisfies **Statement 47** but violates **Statement 48**.

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

$$\text{if0}(\bar{0}, v) \rightsquigarrow_b \bar{0} \qquad \frac{v \neq \bar{0}}{\text{if0}(v, w) \rightsquigarrow_b w} \qquad \frac{\Delta ; \Gamma \vdash e_1 : A \quad \Delta ; \Gamma \vdash e_2 : A}{\Delta ; \Gamma \vdash \text{if0}(e_1, e_2) : A}$$

Even with this extension, type safety and semantic soundness still hold.⁴ However, we can now construct the following value:

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

We can see that v_{bad} has type `bool` and satisfies **Statement 47** but not **Statement 48**. We have:

$$v_{\text{bad}} \langle \text{int} \rangle \bar{0} \bar{1} \downarrow \bar{0} \qquad v_{\text{bad}} \langle \text{int} \rangle \bar{1} \bar{0} \downarrow \bar{0}$$

And therefore:

$$\text{if } v_{\text{bad}} \text{ then } \bar{0} \text{ else } \bar{1} \downarrow \bar{0} \qquad \text{if } v_{\text{bad}} \text{ then } \bar{1} \text{ else } \bar{0} \downarrow \bar{0}$$

In other words, sometimes v_{bad} behaves like `true` and sometimes it behaves like `false`.

In order to prove that our Scott encodings are faithful encodings, we would need to extend our model to reasoning about *relational parametricity* as originally proposed by Reynolds [2]. That would rule out the existence of operations like `if0`, which our current model cannot rule out. However, we will not do that in this class.

⁴It may seem surprising that semantic soundness still holds, given that we saw earlier that adding type casts breaks semantic soundness. The reason this works is that to define `if0`, we did *not* have to add types to our runtime terms, so we can still use our previous approach of interpreting universal types by quantifying over all semantic types. In contrast, `cast` requires types to be present in runtime terms, so that no longer works. In particular, one crucial limitation of `if0` is that it takes two terms of *the same* type, and returns one of them.

3.9 Semantic Type Safety of Unsafe Code with Existential Types

In [Section 3.3](#), we have seen that existential types can be used to encode the concept of data abstraction and encapsulation, of types that have a private representation that cannot be observed from the outside. In this section, we will show that this can even be extended to *encapsulation of unsafe code*: we can write code that is not obviously type-safe, in the sense that our type system will not be able to type-check this code. However, by hiding the internals of that code behind an existential type, we can ensure that all *users* of this code still benefit from the System F type safety and termination guarantees. And using our semantic model, we can even prove this to be the case!

To do this, we first need to introduce an unsafe operation into our language: `assert`. The term `assert e` gets stuck when e does not evaluate to `true`.

Source Terms	$E ::= \dots$	<code>assert E</code>
Runtime Terms	$e ::= \dots$	<code>assert e</code>
Evaluation Contexts	$K ::= \dots$	<code>assert K</code>

Contextual operational semantics

$$e_1 \rightsquigarrow_b e_2$$

$$\dots \quad \frac{\text{ASSERT-TRUE}}{\text{assert true} \rightsquigarrow_b ()}$$

Big-Step Semantics

$$e \Downarrow v$$

$$\dots \quad \frac{\frac{\text{ASSERT-TRUE}}{e \Downarrow \text{true}}}{\text{assert } e \Downarrow ()}$$

Note that there is no reduction rule for `assert false`! We consider `assert false` to be an uncontrolled program crash, or some other undesirable outcome that we want to be sure does not occur. This means that, for instance, $\lambda x : \text{bool}. \text{assert } x$ is not, in general, a safe function, and we thus cannot give a typing rule for `assert`.

In a fully dependent type system that cannot just track $x : \text{bool}$, but can also track the exact value of x , we could still give a typing rule for `assert`. However, that is not the type system we are considering here, and most real-world languages also do not have such a powerful type system. Many real-world languages *do* have unsafe operations though: from `unsafe` code in Rust to `Obj.magic` in OCaml to `unsafePerfromIO` in Haskell to the `unsafe` package in Go or Java to interfaces with native system libraries in any managed language. The situation we are discussing in our idealized formal language models a frequently occurring problem in real software.

The ideal approach to dealing with this safety issue is *encapsulation*: we provide an abstract API around our unsafe code that ensures that users of our library cannot be accidentally exposed to the unsafety. As a model of that in System F, we are going to consider the following signature for a type representing a single bit:

$$\text{BIT} := \exists \alpha. \{ \text{zero} : \alpha, \text{flip} : \alpha \rightarrow \alpha, \text{get} : \alpha \rightarrow \text{bool} \}$$

We can implement this signature as follows.

$$\text{MyBit} := \text{pack BIT from int with } \{ \text{zero} = \bar{0}, \text{flip} = \lambda x : \text{int}. \bar{1} - x, \text{get} = \lambda x : \text{int}. x > \bar{0} \}$$

This implementation is well-typed. A client may use it as follows:

```

unpack MyBit as  $\alpha$  with  $c$  in
  let  $b = c.zero$  in
  let  $b = c.flip\ b$  in
  let  $b = c.flip\ b$  in
   $c.get\ b$ 

```

This term evaluates to `false`.

Observe that the interface ensures that values of the abstract type are only ever 0 or 1. To demonstrate the use of unsafe code, we will adjust this code such that it gets stuck (“crashes”) if that condition is ever violated:

$$\text{MyBit}_{\text{unsafe}} := \text{pack } \{\text{zero} = \bar{0}, \text{flip} = \lambda x. \text{assert } (x = \bar{0} \vee x = \bar{1}) ; \bar{1} - x, \\ \text{get} = \lambda x. \text{assert } (x = \bar{0} \vee x = \bar{1}) ; x > \bar{0}\}$$

This is a run-time term (without type annotations) since terms with `assert` anyway cannot be type-checked. Here, sequential composition $e_1 ; e_2$ is encoded as `let $x = e_1$ in e_2` for some unused variable x . (As defined before, let-bindings `let $x = e_1$ in e_2` are in turn encoded as $(\lambda x. e_2) e_1$.)

Given that the value of the abstract type can only ever be 0 or 1, it should be safe to use $\text{MyBit}_{\text{unsafe}}$ despite its use of assertions that could make the program crash. This code is not *syntactically safe*, which means we cannot type-check it, but it is still *semantically safe*, which means according to our semantic type system, $\text{MyBit}_{\text{unsafe}}$ is actually a valid inhabitant of type `BIT`. Let us prove this claim:

Lemma 49 (Semantically Safe Booleans). $\text{MyBit}_{\text{unsafe}} \in \mathcal{V}[\![\text{BIT}]\!]_{\delta_{\text{emp}}}$.

Proof. We have to show $\text{MyBit}_{\text{unsafe}} \in \mathcal{V}[\![\exists \alpha. \{\text{zero} : \alpha, \text{flip} : \alpha \rightarrow \alpha, \text{get} : \alpha \rightarrow \text{bool}\}]\!]_{\delta_{\text{emp}}}$. So we pick $\tau := \{\bar{0}, \bar{1}\}$ and show $v_{\text{MyBit}} \in \mathcal{V}[\![\{\text{zero} : \tau, \text{flip} : \tau \rightarrow \alpha, \text{get} : \tau \rightarrow \text{bool}\}]\!]$ where

$$v_{\text{MyBit}} := \{\text{zero} = \bar{0}, \text{flip} = \lambda x. \text{assert } (x = \bar{0} \vee x = \bar{1}) ; \bar{1} - x, \\ \text{get} = \lambda x. \text{assert } (x = \bar{0} \vee x = \bar{1}) ; x > \bar{0}\}$$

This is a record type, which is sugar for a product type, so we have three goals to show: each of the fields of the record must have the matching semantic type.

- **zero:** we have to show $\bar{0} \in \mathcal{V}[\![\tau]\!]$.
This follows immediately from the definition of τ .
- **flip:** we have to show $\lambda x. \text{assert } (x = \bar{0} \vee x = \bar{1}) ; \bar{1} - x \in \mathcal{V}[\![\tau \rightarrow \tau]\!]$.
So suppose $v \in \tau$; we have to show $\text{assert } (v = \bar{0} \vee v = \bar{1}) ; \bar{1} - v \in \mathcal{E}[\![\tau]\!]$.
We have that $v = \bar{n}$ for $n \in \{0, 1\}$, and therefore $(v = \bar{0} \vee v = \bar{1}) \downarrow \text{true}$.
It follows that $\text{assert } \dots \downarrow ()$, and $\dots ; \bar{1} - v \downarrow \bar{1} - n$.
This finishes the proof since $\bar{1} - n \in \tau$.
- **get:** we have to show $\lambda x. \text{assert } (x = \bar{0} \vee x = \bar{1}) ; x > \bar{0} \in \mathcal{V}[\![\tau \rightarrow \text{bool}]\!]$.
With similar reasoning at above, we show that $\text{assert } \dots \downarrow ()$.
The proof finishes since $v > \bar{0}$ evaluates to a Boolean. □

The implications of this lemma are profound: we now have that *any* well-typed code that uses $\text{MyBit}_{\text{unsafe}}$ in arbitrary ways, as long as what it does is permitted by the type `BIT`, is actually type safe!

Lemma 50. Assume an arbitrary term e and type A such that $x : \text{BIT} \vdash e : A$. Then $e[\text{MyBit}_{\text{unsafe}}/x]$ is type safe, i.e., for all e' such that $e[\text{MyBit}_{\text{unsafe}}/x] \rightsquigarrow^* e'$, we have that e' is progressive.

Proof. By [Theorem 41](#) (Semantic Soundness), we have that $x : \text{BIT} \models e : A$.

We instantiate semantic soundness with $\delta := \delta_{\text{emp}}$ and $\gamma := [x \mapsto \text{MyBit}_{\text{unsafe}}]$.

We have to show $\gamma \in \mathcal{G}[[x : \text{BIT}]]$, which follows from [Lemma 49](#).

Hence we have $e[\gamma] \in \mathcal{E}[[A]]$.

With $e[\gamma] = e[\text{MyBit}_{\text{unsafe}}/x]$ and [Lemma 43](#) (Semantic Type Safety), we are done. \square

This result is extremely useful, since its only assumption (that e is *syntactically* well-typed) can be established fully automatically by a type checker. In other words, we can now write *arbitrary code* using $\text{MyBit}_{\text{unsafe}}$, and as long as that code passes the regular typing requirements, the entire program enjoys all the guarantees of well-typed programs—despite the use of unsafe assertions inside the implementation of $\text{MyBit}_{\text{unsafe}}$. We only had to verify the semantic safety of $\text{MyBit}_{\text{unsafe}}$ by hand, and the magic of our semantic model then tells us that all of the (infinitely many) syntactically well-typed programs using $\text{MyBit}_{\text{unsafe}}$ are safe.

This technique works not only for languages like System F that are deterministic and where all terms terminate, it works for any language. For languages with non-termination and non-determinism, we have to use a slightly different expression relation \mathcal{E} : instead of saying that all valid expressions evaluate to a valid value, we say that *if* they reduce to a value, then that value is valid, and furthermore they can never reach a stuck state.

Semantic type soundness is not just an alternative to the syntactic proof via progress-and-preservation, it shows that the type system “makes sense” in a much deeper way and opens the door for reasoning about well-encapsulated unsafe code.

4 Mutable State

In this chapter, we extend the language with references. The core operations on references are allocation, load, and store.⁵ The runtime value of a reference is called a *location*. Locations cannot occur in source terms; they can never be written in the source code, and can only be generated at runtime.

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

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

$$h ; (\lambda x. e) v \rightsquigarrow_b h ; e[v/x]$$

The base reduction rules for allocation, load, and store, however, interact with the heap:

Base reduction

$$\boxed{h_1 ; e_1 \rightsquigarrow_b h_2 ; e_2}$$

$$\begin{array}{ccc} \text{NEW} & \text{LOAD} & \text{STORE} \\ \frac{\ell = \text{fresh}(\text{dom}(h))}{h ; \text{new } v \rightsquigarrow_b h[\ell \mapsto v] ; \ell} & \frac{h(\ell) = v}{h ; !\ell \rightsquigarrow_b h ; v} & \frac{\ell \in \text{dom}(h)}{h ; \ell \leftarrow v \rightsquigarrow_b h[\ell \mapsto v] ; v} \end{array}$$

When we say $h(\ell) = v$, this implicitly also asserts that $\ell \in \text{dom}(h)$. Here, **fresh** is a mathematical function that picks the “next fresh address” for the current heap. This is always possible since h is a finite heap and there are infinitely many locations. The key property of **fresh** is that $\text{fresh}(X) \notin X$. We do not actually care how **fresh** picks the location, as long as that property is satisfied. We use such a function to keep our semantics deterministic, which simplifies the metatheory. (The alternative would be for the **NEW** rule to allow reducing with *any* l that satisfies $l \notin \text{dom}(h)$, but that would be non-deterministic.)

Contextual reduction

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

$$\frac{\text{CTX} \quad h ; e \rightsquigarrow_b h' ; e'}{h ; K[e] \rightsquigarrow h' ; K[e']}$$

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

Church-style typing

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

$$\begin{array}{ccc} \text{NEW} & \text{LOAD} & \text{STORE} \\ \frac{\Delta ; \Gamma \vdash E : A}{\Delta ; \Gamma \vdash \text{new } E : \text{ref } A} & \frac{\Delta ; \Gamma \vdash E : \text{ref } A}{\Delta ; \Gamma \vdash !E : A} & \frac{\Delta ; \Gamma \vdash E_1 : \text{ref } A \quad \Delta ; \Gamma \vdash E_2 : A}{\Delta ; \Gamma \vdash E_1 \leftarrow E_2 : A} \end{array}$$

⁵We assume a garbage collector, so there is no deallocation operation.

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

$$p := \begin{array}{l} \text{let } x = \text{new } \bar{0} \text{ in} \\ \text{let } c : \mathbf{1} \rightarrow \text{int} = (\lambda y : \mathbf{1}. x \leftarrow !x + 1) \text{ in} \\ c() - c() \end{array}$$

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

$$\begin{aligned} h ; p &\rightsquigarrow h[\ell \mapsto \bar{0}] ; \text{let } c = (\lambda y. \ell \leftarrow !\ell + 1) \text{ in } c() + c() & \ell = \text{fresh}(\text{dom}(h)) \\ &\rightsquigarrow h[\ell \mapsto \bar{0}] ; (\lambda y. \ell \leftarrow !\ell + 1)() - (\lambda y. \ell \leftarrow !\ell + 1)() \\ &\rightsquigarrow h[\ell \mapsto \bar{0}] ; (\lambda y. \ell \leftarrow !\ell + 1)() - (\ell \leftarrow \bar{1}) \\ &\rightsquigarrow h[\ell \mapsto \bar{1}] ; (\lambda y. \ell \leftarrow !\ell + 1)() - \bar{1} \\ &\rightsquigarrow h[\ell \mapsto \bar{1}] ; (\ell \leftarrow \bar{2}) - \bar{1} \\ &\rightsquigarrow h[\ell \mapsto \bar{2}] ; \bar{2} - \bar{1} \\ &\rightsquigarrow h[\ell \mapsto \bar{2}] ; \bar{1} \end{aligned}$$

4.1 Data Abstraction via Local State

Previously we have seen how existential types can be used to encode a form of data abstraction. 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.

$$\begin{aligned} \text{MUTBIT} &:= \{\text{flip} : \mathbf{1} \rightarrow \mathbf{1}, \text{get} : \mathbf{1} \rightarrow \text{bool}\} \\ \text{MyMutBit} &:= \text{let } x = \text{new } \bar{0} \\ &\quad \text{in } \{\text{flip} := \lambda y : \mathbf{1}. x \leftarrow \bar{1} - !x ; (), \\ &\quad \text{get} := \lambda y : \mathbf{1}. !x > 0\} \end{aligned}$$

Note that `MyMutBit` is *not* a value! It is an expression that, when executed, allocates a new reference and then returns something of type `MUTBIT`. However, the fact that a reference is used is not visible in that type. All we can see is that there are two functions we can call; the fact that these functions are “magically linked” by referring to the same underlying memory is hidden.

We could use this “bit” type in a client as follows:

$$\begin{array}{l} \text{let } b = \text{MyMutBit in} \\ b.\text{flip}() ; b.\text{get}() \end{array}$$

This term evaluates to `true`. Follow the reduction step-by-step if you are not convinced.

Similar to the previous example with existential types, the internal state of the bit is not accessible to clients of this library. Previously, that was enforced by the type system hiding the “packed” existential type and only giving access to the explicitly declared operations; now, this is enforced by the reference x being private to the implementation, with no way for clients to read or write that location.

For instance, here is a different implementation that uses a Boolean to represent the internal state of the bit:

$$\begin{aligned} \text{MyMutBit}_{\text{bool}} &:= \text{let } x = \text{new false} \\ &\quad \text{in } \{\text{flip} := \lambda y : \mathbf{1}. x \leftarrow \text{if } !x \text{ then false else true} ; (), \\ &\quad \text{get} := \lambda y : \mathbf{1}. !x\} \end{aligned}$$

This has the same type as `MyMutBit`, and if we plug it in our example client above (or any other client), it will behave the exact same way.

Just like in [Section 3.9](#), we can even introduce unsafe assertions which check our intended invariant: that the content of the location is always 0 or 1.

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

Using a semantic model, we will again be able to show that even though this program is not *syntactically* well-typed (since it uses unsafe assertions), it is still *semantically* well-typed: clients using it at type `MUTBIT` are guaranteed to enjoy the usual type safety guarantees.

4.2 Type Safety

However, before we consider a semantic model, we start as usual by proving type safety. We have to extend basically every part of this proof to account for the presence of a heap.

The very notion of a “safe term” needs updating since we now always need to track in which heap the term is supposed to be reduced:

Definition 51 (Reducible Terms). *A (runtime) term e is reducible in heap h if there exists h', e' s.t. $h ; e \rightsquigarrow h' ; e'$.*

Definition 52 (Progressive Terms). *A (runtime) term e is progressive in heap h if either it is a value or it is reducible in h .*

Definition 53 (Progressive Terms). *A (runtime) term e is safe in heap h if for all h', e' s.t. $h ; e \rightsquigarrow^* h' ; e'$, the term e' is progressive in h' .*

Next, we define typing on runtime terms. For source terms, we could largely ignore the heap, since source terms cannot contain locations ℓ . However, at runtime, locations *do* show up, so to prove preservation we need a typing rule for them. To this end, we extend the typing rules with a heap typing context Σ that assigns types to locations:

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

This new context is used only in the typing rule for locations. The remaining rules just carry Σ around unchanged. (In particular, there is no typing rule that adds anything to Σ . Instead, the progress proof will extend Σ when a new location is generated. We will get back to this when we discuss preservation.)

Curry-style typing

$\Sigma ; \Delta ; \Gamma \vdash e : A$

$\frac{\text{NEW} \quad \Sigma ; \Delta ; \Gamma \vdash e : A}{\Sigma ; \Delta ; \Gamma \vdash \text{new } e : \text{ref } A}$	$\frac{\text{LOAD} \quad \Sigma ; \Delta ; \Gamma \vdash e : \text{ref } A}{\Sigma ; \Delta ; \Gamma \vdash !e : A}$	$\frac{\text{STORE} \quad \Sigma ; \Delta ; \Gamma \vdash e_1 : \text{ref } A \quad \Sigma ; \Delta ; \Gamma \vdash e_2 : A}{\Sigma ; \Delta ; \Gamma \vdash e_1 \leftarrow e_2 : A}$
$\frac{\text{LOC} \quad \ell : \text{ref } A \in \Sigma}{\Sigma ; \Delta ; \Gamma \vdash \ell : \text{ref } A}$		

We also need a notion of when a heap h matches a heap context Σ : each location declared in Σ must exist in h and must have a value of the right type.

Heap Typing

$h : \Sigma$

$$h : \Sigma := \forall \ell : \text{ref } A \in \Sigma. \exists v. h(\ell) = v \wedge \Sigma; \emptyset; \emptyset \vdash v : A$$

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 is type-checked in a heap typing that contains ℓ .

Our venerable lifting lemma gets an update to account for the presence of heaps:

Lemma 54 (Context Lifting). *If $h_1 ; e_1 \rightsquigarrow h_2 ; e_2$, then $h_1 ; K[e_1] \rightsquigarrow h_2 ; K[e_2]$.*

Proof. As before, via context composition. □

We also get a new canonical forms lemma, as usual when we add a new type:

Lemma 55 (Canonical forms). *If $\Sigma; \emptyset; \emptyset \vdash v : \text{ref } A$, then $v = \ell$ for some ℓ s.t. $\ell : \text{ref } A \in \Sigma$.*

Proof. By inversion. □

Now we are ready to re-state the progress lemma: for $h ; e$ to be able to make progress, e needs to be well-typed in a heap context Σ and h must be valid for that context. Note that we require Γ and Δ to be empty for progress (as before), but we allow the heap context Σ to be non-empty. This is because during program execution, the heap may grow, so a lemma that only works for empty heaps would not be strong enough! In contrast, if a term is closed (well-typed with empty Γ and Δ), then it remains closed during execution.

Theorem 56 (Progress).

If $\Sigma; \emptyset; \emptyset \vdash e : A$ and $h : \Sigma$, then e is progressive in h .

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

$$IH(\Sigma, \Delta, \Gamma, e, A) := \Delta = \emptyset \wedge \Gamma = \emptyset \Rightarrow \text{progressive}(h ; e)$$

The new cases are:

- **LOC**: $e = \ell$ and $A = \text{ref } B$.
Have: $\ell : \text{ref } A \in \Sigma$. To show: ℓ is progressive.
 ℓ is a value and hence progressive.
- **NEW**: $e = \text{new } e_1$ and $A = \text{ref } B$.
Have: $\Sigma; \emptyset; \emptyset \vdash e_1 : B$ and $IH(\Sigma, \emptyset, \emptyset, e_1, B)$.
To show: $\text{new } e_1$ is progressive in h .
By IH , we have that e_1 is progressive. We distinguish two cases:
 - e_1 is a value.
By **NEW**, we have $h ; \text{new } e_1 \rightsquigarrow h[\ell \mapsto e_1] ; \ell$ for $\ell = \text{fresh}(\text{dom}(h))$, so we are done.
 - $h ; e_1 \rightsquigarrow h' ; e'_1$ for some h', e'_1 . Then $h ; \text{new } e_1 \rightsquigarrow h' ; \text{new } e'_1$ by **Lemma 54** with $K := \text{new } \bullet$.
- **LOAD**: $e = !e_1$.
Have $\Sigma; \emptyset; \emptyset \vdash e_1 : \text{ref } A$ and $IH(\Sigma, \emptyset, \emptyset, e_1, \text{ref } A)$.
To show: $!e_1$ is progressive in h .
By IH , we have that e_1 is progressive. We distinguish two cases:

- e_1 is a value.
Then, by canonical forms, we have $e_1 = \ell$ and $\ell : \text{ref } A \in \Sigma$.
Thus, by $h : \Sigma$, we have some v s.t. $h(\ell) = v$ and $\Sigma ; \emptyset ; \emptyset \vdash v : A$.
By **LOAD**, it follows that $h ; !e_1 \rightsquigarrow h ; v$, so we are done.
- $h ; e_1 \rightsquigarrow h' ; e'_1$ for some h', e'_1 .
Then $h ; !e_1 \rightsquigarrow h' ; !e'_1$ by **Lemma 54** with $K := !\bullet$.

- **STORE**: See exercise sheet.

□

Next, we come to preservation. As usual we start with base preservation. Recall that so far, we stated preservation as: if $\vdash e : A$ and $e \rightsquigarrow_b e'$, then $\vdash e' : A$. However, that will no longer work: as already discussed for progress, we need to allow a non-empty heap context Σ . Naively we might then attempt to prove the following: if $\Sigma ; \emptyset ; \emptyset \vdash e : A$ and $e \rightsquigarrow_b e'$, then $\Sigma ; \emptyset ; \emptyset \vdash e' : A$. This theorem however does not hold: when executing a **new**, a new allocation gets added to the heap, so we need to update the heap typing accordingly! This shows that the heap typing needs to be able to *grow* during execution. We will thus prove:

Lemma 57 (Base preservation).

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

Proof. Remember that we are doing inversion on $e \rightsquigarrow_b e'$. Existing cases remain mostly unchanged. (We have $h' = h$ and pick $\Sigma' := \Sigma$.) The new cases are:

- **NEW**: $e = \text{new } v$ and $e' = \ell$ and $\ell = \text{fresh}(\text{dom}(h))$ and $h' = h[\ell \mapsto v]$.
By inversion on typing of e , we have $\Sigma ; \emptyset ; \emptyset \vdash v : B$ and $A = \text{ref } B$.
We pick $\Sigma' := \Sigma, \ell : \text{ref } B$. To show: $\Sigma \subseteq \Sigma'$ and $h' : \Sigma'$ and $\Sigma' ; \emptyset ; \emptyset \vdash \ell : \text{ref } B$.
The first follows from $\ell \notin \text{dom}(h)$ and **Lemma 59** below.
The second follows from typing of v and **Lemma 60** below.
The third follows from **LOC**.
- **LOAD**: See exercise sheet.
- **STORE**: $e = \ell \leftarrow v$ and $h' = h[\ell \mapsto v]$ and $e' = v$ and $\ell \in \text{dom}(h)$.
We pick $\Sigma' := \Sigma$. To show: $\Sigma \subseteq \Sigma'$ (trivial) and $h' : \Sigma'$ (1) and $\Sigma' ; \emptyset ; \emptyset \vdash v : A$ (2).
By inversion on typing of e , we have $\Sigma ; \emptyset ; \emptyset \vdash \ell : \text{ref } A$ and $\Sigma ; \emptyset ; \emptyset \vdash v : A$.
Thus we have (2), and only (1) remains to be shown.
By inversion on typing of ℓ , we have $\ell : \text{ref } B \in \Sigma$.
This suffices to re-establish heap typing (see **Lemma 61** below).

□

We have used the following lemmas here, which we leave without proof (you can find their proofs in the Coq development):

Lemma 58 (Weakening of heap typing). If $\Sigma ; \Delta ; \Gamma \vdash e : A$ and $\Sigma' \supseteq \Sigma$, then $\Sigma' ; \Delta ; \Gamma \vdash e : A$.

Lemma 59 (Heap context extension). If $h : \Sigma$ and $\ell \notin \text{dom}(h)$, then $\ell \notin \Sigma$ and hence $\Sigma \subseteq \Sigma, \ell : \text{ref } A$.

Lemma 60 (Heap typing preservation for **NEW**). If $h : \Sigma$ and $\ell \notin \text{dom}(h)$ and $\Sigma ; \emptyset ; \emptyset \vdash v : B$, then $h[\ell \mapsto v] : \Sigma, \ell : \text{ref } B$.

Lemma 61 (Heap typing preservation for **STORE**). If $h : \Sigma$ and $\ell : \text{ref } B \in \Sigma$ and $\Sigma ; \emptyset ; \emptyset \vdash v : B$, then $h[\ell \mapsto v] : \Sigma$.

Finally, we need to lift base preservation to preservation on the contextual semantics. This works largely as before, but we need to update the definition of evaluation context typing to account for the possibility of the heap typing to grow:

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

With this definition, and the same proof strategy as before, we obtain:

Theorem 62 (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 $\Sigma'; \emptyset; \emptyset \vdash e' : A$.*

And finally, as the crowning achievement of all this work, we obtain type safety for System F with mutable state:

Corollary 63 (Type Safety). *If $\vdash e : A$, then e is safe in every heap.*

Mutable state is a non-trivial feature, so proving type safety for this language is quite an achievement!

4.3 Recursion via state

In the previous chapter we have seen how to encode recursion using type casts. With mutable higher-order state, there is another way to define recursive functions: we can use a reference to store the function, and then the function can load from that reference to call itself. The only issue with this idea is: we have to initialize the reference to some value, and we have to declare the function *after* creating the reference so that the function can use the reference. What do we choose as initial value? It turns out we can just use some dummy function, like the identity function; we will make sure that this gets overwritten with the actually intended function before the reference is loaded the first time.

Here is the code to implement an infinite loop with this approach:

```
p := let g = new (λx : 1. x) in
    let f = λx : 1. (!g) x in
    g ← f ; f ()
```

Here is what happens when this program executes:

$$\begin{aligned} h; p &\rightsquigarrow^* h[\ell \mapsto (\lambda x. x)]; \ell \leftarrow (\lambda x. (!\ell) x); (\lambda x. (!\ell) x) () & \ell = \text{fresh}(\text{dom}(h)) \\ &\rightsquigarrow^* h[\ell \mapsto (\lambda x. (!\ell) x)]; (\lambda x. (!\ell) x) () \\ &\rightsquigarrow^* h[\ell \mapsto (\lambda x. (!\ell) x)]; (!\ell) () \\ &\rightsquigarrow^* h[\ell \mapsto (\lambda x. (!\ell) x)]; (\lambda x. (!\ell) x) () \\ &\vdots \end{aligned}$$

We can also define something more useful with this approach, like a function that checks whether a non-negative number is even:

```
even := let g = new (λx : int. x) in
    let f = λx : int. if x = 0 then true else if x = 1 then false else (!g) (x - 2) in
    g ← f ; f
```

This approach to recursion is known as *Landin's Knot*.

References

- [1] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 2nd revised edition, 1984.
- [2] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP*, 1983.
- [3] S. Schäfer, T. Tebbi, and G. Smolka. Autosubst: Reasoning with De Bruijn terms and parallel substitutions. In *Interactive Theorem Proving*, 2015.
- [4] K. Stark. Mechanising syntax with binders in Coq, 2019.
- [5] P. Wadler. Theorems for free! In *FPCA*, 1989.