Formal Foundations of Programming Languages
Lecture Notes

Ralf Jung   Max Vistrup

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

Contents

1   Simply Typed Lambda Calculus

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

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

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 $(\lambda x.\, x + \overline{1})\, \overline{41}$ proceeds with $\overline{41} + \overline{1}$ 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:

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

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 $\mathsf{fv}(e)$ of free variables in e is defined as

$$
\begin{array}{rl}
\mathsf{fv}(x) &:= \{x\} \\
\mathsf{fv}(\lambda x.\, e) &:= \mathsf{fv}(e) \setminus \{x\} \\
\mathsf{fv}(e_1\, e_2) &:= \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) \\
\mathsf{fv}(e_1 + e_2) &:= \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) \\
\mathsf{fv}(\overline{n}) &:= \emptyset
\end{array}
$$

The problem with the naive definition of substitution above is called *variable capturing*. Variable capturing occurs if we insert an expression with a free variable such as $\overline{2} + x$ into an expression which binds the free variable. For example, if we naively substitute $\overline{2} + x$ for y in $\lambda x.\, y + x$, then we obtain $\lambda x.\, (\overline{2} + x) + x$. Since x was free in $\overline{2} + x$ but is bound in $\lambda x.\, (\overline{2} + x) + x$, one speaks of variable capturing. Variable capturing is problematic, because the programmer 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, $(\lambda x.\, y + x)[\overline{2} + x / y]$ should result in $\lambda z.\, (\overline{2} + x) + z$, 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: $\mathsf{fv}(e) = \emptyset$), 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.

$$
\frac{}{\overline{n} \downarrow \overline{n}} \text{ LITERAL}
\qquad
\frac{}{\lambda x.\, e \downarrow \lambda x.\, e} \text{ LAMBDA}
\qquad
\frac{e_1 \downarrow \lambda x.\, e \quad e_2 \downarrow v_2 \quad e[v_2/x] \downarrow v}{e_1\, e_2 \downarrow v} \text{ APP}
\qquad
\frac{e_1 \downarrow \overline{n_1} \quad e_2 \downarrow \overline{n_2}}{e_1 + e_2 \downarrow \overline{n_1 + n_2}} \text{ PLUS}
$$

Exercise 1 Prove that the big-step semantics $e \downarrow v$ is *deterministic*. That is, show that if $e \downarrow v$ and $e \downarrow v'$, then $v = v'$. •

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:

$$
\frac{e_2 \succ e_2'}{e_1\, e_2 \succ e_1\, e_2'} \text{ APP-STRUCT-R}
\qquad
\frac{e_1 \succ e_1'}{e_1\, v_2 \succ e_1'\, v_2} \text{ APP-STRUCT-L}
\qquad
\frac{}{(\lambda x.\, e)\, v \succ e[v/x]} \text{ BETA}
$$

$$
\frac{e_2 \succ e_2'}{e_1 + e_2 \succ e_1 + e_2'} \text{ PLUS-STRUCT-R}
\qquad
\frac{e_1 \succ e_1'}{e_1 + v_2 \succ e_1' + v_2} \text{ PLUS-STRUCT-L}
\qquad
\frac{}{\overline{n} + \overline{m} \succ \overline{n + m}} \text{ PLUS}
$$

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 $\boxed{K[e]}$

$$\bullet[e] := e \qquad\qquad (e' + K)[e] := e' + K[e]$$
$$(e'\,K)[e] := e'(K[e]) \qquad\qquad (K + v)[e] := K[e] + v$$
$$(K\,v)[e] := (K[e])\,v$$

Base reduction $\boxed{e_1 \rightsquigarrow_{\text{b}} e_2}$

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

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

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

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 $e_1' \rightsquigarrow_{\text{b}} e_2'$ 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. $\square$

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

$$\bullet \circ K_2 := K_2 \qquad\qquad (e' + K) \circ K_2 := e' + (K \circ K_2)$$
$$(e'\,K) \circ K_2 := e'\,(K \circ K_2) \qquad\qquad (K + v) \circ K_2 := (K \circ K_2) + v$$
$$(K\,v) \circ K_2 := (K \circ K_2)\,v$$

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 $\leadsto^*$: if $e_1 \leadsto^* e_2$, then $K[e_1] \leadsto^* K[e_2]$. 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.

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

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

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

As it turns out, Ω reduces to itself! Thus, there are terms in the untyped lambda calculus such as Ω which *diverge* (*i.e.*, their reduction chains do not terminate). 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 $\mathsf{S}\, n$, 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 $\widetilde{0}$ for the Scott encoding of 0, and so on:

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

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

$$\widetilde{0} := \lambda z, s. \, z$$

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

Note that $\widetilde{n}$ 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:

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

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

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 $\mathsf{pred} := \lambda n. \, n \, \widetilde{0} \, (\lambda n'. \, n')$ computes the predecessor of a Scott encoded natural number.

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

$$\mathsf{mul2} := \lambda n. \, n \, \widetilde{0} \, (\lambda n'. \, \widetilde{S}(\widetilde{S}(\mathsf{mul2}\; n')))$$

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, $\mathsf{mul2}$, 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 fix (sometimes called fixpoint combinator or Y-combinator). The idea of 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 $\mathsf{fix}\; F\; v$ reduces to $F\; (\mathsf{fix}\; F)\; v$. In other words, we end up calling F such that rec becomes $\mathsf{fix}\; F$ (and the value v is just forwarded).

In other words, the desired reduction behavior of fix is

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

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

$$\mathsf{MUL2} := \lambda rec, \lambda n. \, n \, \widetilde{0} \, (\lambda n'. \, \widetilde{S}(\widetilde{S}(rec\; n')))$$

$$\mathsf{mul2} := \mathsf{fix}\; \mathsf{MUL2}$$

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

$\mathsf{mul2}\,\widetilde{1}$	$=$	$\mathsf{fix}\;\mathsf{MUL2}\;\widetilde{1}$	by the definition of $\mathsf{mul2}$
	$\succ^*$	$\mathsf{MUL2}\;(\mathsf{fix}\;\mathsf{MUL2})\;\widetilde{1}$	by the fix reduction rule
	$=$	$\mathsf{MUL2}\;\mathsf{mul2}\;\widetilde{1}$	by the definition of $\mathsf{mul2}$
	$\succ^*$	$\widetilde{1}\,\widetilde{0}\,(\lambda n'. \, \widetilde{S}(\widetilde{S}(\mathsf{mul2}\; n')))$	by the definition of $\mathsf{MUL2}$ and two β-reductions
	$\succ^*$	$(\lambda n'. \, \widetilde{S}(\widetilde{S}(\mathsf{mul2}\; n')))\,\widetilde{0}$	by the definition of $\widetilde{1}$ and two β-reductions
	$\succ$	$\widetilde{S}(\widetilde{S}(\mathsf{mul2}\,\widetilde{0}))$	by another β-reduction
	$\succ^*$	$\widetilde{S}(\widetilde{S}(\mathsf{MUL2}\,\mathsf{mul2}\,\widetilde{0}))$	by the fix rule and the definition of $\mathsf{mul2}$
	$\succ^*$	$\widetilde{S}(\widetilde{S}(\widetilde{0}\,\widetilde{0}\,(\lambda n'. \, \widetilde{S}(\widetilde{S}(\mathsf{mul2}\; n')))))$	by the definition of $\mathsf{MUL2}$ and two β-reductions
	$\succ^*$	$\widetilde{S}(\widetilde{S}(\widetilde{0}))$	by the definition of $\widetilde{0}$ and two β-reductions
	$\succ^*$	$\widetilde{2}$	by the definition of $\widetilde{S}$ and two β-reductions

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

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

The proof is simple but tedious; the key point is showing that mul2 satisfies mul2 $\widetilde{0} \succ^* \widetilde{0}$ and mul2 $\widetilde{S\,n} \succ^* \widetilde{S}\ (\widetilde{S}\ (\text{mul2}\ \widetilde{n}))$. 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:

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

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.

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

Church-style typing $\boxed{\Gamma \vdash E : A}$
Typing on source terms amounts to *checking* whether a source term is properly annotated.

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

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

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

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

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

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

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

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

$$\text{HERE} \quad \frac{}{x : A \in \Gamma, x : A} \qquad \text{THERE} \quad \frac{x : A \in \Gamma \qquad x \neq y}{x : A \in \Gamma, y : B}$$

This means lookup is right-to-left and only the first assignment with a given variable name counts, *i.e.*, $x : \mathsf{int} \in \emptyset, x : \mathsf{int}, x : \mathsf{int} \to \mathsf{int}$ 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: $e = \lambda x.\, e'$, $A = A' \to B'$.
  Have: $\Gamma, x : A' \vdash e' : B'$ and $IH((\Gamma, x : A'), e', B')$. To Show: $IH(\Gamma, \lambda x.\, e', A' \to B')$.
  (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: $\Gamma \vdash e_1 : A' \to B'$ and $IH(\Gamma, e_1, A' \to B')$ 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.)

Draft of November 13, 2023

- PLUS: $e = e_1 + e_2$, $A = \mathsf{int}$.
  Have: $\Gamma \vdash e_1 : \mathsf{int}$ and $IH(\Gamma, e_1, \mathsf{int})$ and $\Gamma \vdash e_2 : \mathsf{int}$ and $IH(\Gamma, e_2, \mathsf{int})$.
  To Show: $IH(\Gamma, e_1 + e_2, \mathsf{int})$.
  (This is structurally very similar to the previous case.)

- INT: $e = \overline{n}$, $A = \mathsf{int}$.
  To Show: $IH(\Gamma, \overline{n}, \mathsf{int})$.
  (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 $\boxed{\text{Erase}(\cdot)}$

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

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

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 $\vdash e : A$ and $e \rightsquigarrow^{\star} e'$, then e' is progressive.*

Here, we call the following terms progressive:

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

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. Type safety then says that if we start with a well-typed term, then not only will that term we progressive, but all terms we can reach via reduction are also progressive.

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

– *if $A = \mathsf{int}$, then $v = \overline{n}$ for some n*

– *if $A = A_1 \to A_2$ for some A_1, A_2, then $v = \lambda x.\, e$ for some x, e*

Proof. By inversion. □

Theorem 8 (Progress). *If $\vdash e : A$, then e is progressive.*

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

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

- 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: $e = \lambda x.\, e$, $A = A' \to B'$.
  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: $\vdash e_1 : A' \to A$, 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 7 $e_1 = \lambda x.\, e$ 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: $e = e_1 + e_2$, $A = \mathsf{int}$.
  Have: $\vdash e_1 : \mathsf{int}$ and $\vdash e_2 : \mathsf{int}$, 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 7 $e_1 = \overline{n}$ and $e_2 = \overline{m}$ 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: $e = \overline{n}$, $A = \mathsf{int}$.
  To Show: $\overline{n}$ is progressive.
  $\overline{n}$ is already a value.

$\square$

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.

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 9 (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:

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

---

[1] Very often, we discard hypotheses like $\vdash e_1 : A' \to A$ 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 7.

- 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: $e = \lambda x.\, e'$, $A = A' \to B'$.
  Have: $\Gamma_1, x : A' \vdash e' : A' \to B'$ 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: $\Gamma_2 \vdash \lambda x.\, e : A' \to B'$.
  Done by LAM and IH with $\Gamma_2 := \Gamma_2, x : A'$.

- APP: $e = e_1\, e_2$, $A = B'$.
  Have: $IH(\Gamma_1, e_1, (A' \to B'))$, $IH(\Gamma_1, e_2, A')$. To Show: $\Gamma_2 \vdash e_1 e_2 : B'$.
  Done by APP and the induction hypotheses instantiated with Γ_2.

- PLUS: $e = e_1 + e_2$, $A = \mathsf{int}$.
  Have: $IH(\Gamma_1, e_1, \mathsf{int})$, $IH(\Gamma_1, e_2, \mathsf{int})$. To Show: $\Gamma_2 \vdash e_1 + e_2 : \mathsf{int}$.
  Done by PLUS and the induction hypotheses instantiated with Γ_2.

- INT: $e = \overline{n}$, $A = \mathsf{int}$.
  To Show: $\Gamma_2 : \overline{n} : \mathsf{int}$.
  Done by INT.

$\square$

Lemma 10 (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 9) 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: $\Gamma, x : A \vdash e_1 : B' \to B$ 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: $B = B_1 \to B_2$ and $\Gamma, x : A, y : B_1 \vdash e_1 : B_2$.

  - Case $x \neq y$:
    To Show: $\Gamma \vdash \lambda y.\, e_1[e'/x] : B_1 \to B_2$.
    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 9) to swap the order of x and y, then we are done.

  - Case $x = y$:
    To Show: $\Gamma \vdash \lambda y.\, e_1 : B_1 \to B_2$.
    By LAM, it suffices to show: $\Gamma, y : B_1 \vdash e_1 : B_2$.
    Done by weakening (Lemma 9) 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: $B = \mathsf{int}$ and $\Gamma, x : A \vdash e_1 : \mathsf{int}$ and $\Gamma, x : A \vdash e_2 : \mathsf{int}$.
  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 $e = \overline{n}$
  Have: $\Gamma, x : A \vdash \overline{n} : B$ and $\vdash e' : A$.
  To Show: $\Gamma \vdash \overline{n}[e'/x] : B$.
  By inversion on typing, we have $B = \mathsf{int}$.
  By the definition of substitution, it suffices to show: $\Gamma \vdash \overline{n} : B$.
  Done by INT.

$\square$

Lemma 11 (Base preservation). *If $\vdash e : A$ and $e \rightsquigarrow_{\mathrm{b}} e'$, then $\vdash e' : A$.*

Proof. By inversion on $e \rightsquigarrow_{\mathrm{b}} e'$:

Case 1: BETA, $e = (\lambda x.\, e_1)\, v$ and $e' = e_1[v/x]$. It remains to show that $\vdash e_1[v/x] : A$. By inversion on $\vdash e : A$ we have $\vdash \lambda x.\, e_1 : A_0 \to A$ and $\vdash v : A_0$ for some A_0. By inversion on $\vdash \lambda x.\, e_1 : A_0 \to A$ we have $x : A_0 \vdash e_1 : A$. By Lemma 10, we have $\vdash e_1[v/x] : A$.

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

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 12 (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:

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

- $K = \bullet$:
  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 $\vdash e' : A' \to A$ 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 $\vdash e' : A' \to A$ 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 $\vdash K' : B' \Rightarrow (A' \to A)$ 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 $\vdash K'[e'] : A' \to A$ 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 $A = \mathsf{int}$, $\vdash e' : \mathsf{int}$, and $\vdash K'[e] : \mathsf{int}$.
  Applying $IH(K')$ to the latter, we find there is some B' so that $\vdash K' : B' \Rightarrow \mathsf{int}$ and $\vdash e : B'$.
  Taking $B := B'$, it remains to show that $\vdash e' + K' : B' \Rightarrow \mathsf{int}$ 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 $\vdash (e' + K')[e''] : \mathsf{int}$.
  Using PLUS, it suffices to show $\vdash e' : \mathsf{int}$ and $\vdash K'[e''] : \mathsf{int}$.
  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 $A = \mathsf{int}$, $\vdash K'[e] : \mathsf{int}$, and $\vdash v : \mathsf{int}$.
  Applying $IH(K')$ to the second, we find there is some B' so that $\vdash K' : B' \Rightarrow \mathsf{int}$ and $\vdash e : B'$.
  Taking $B := B'$, it remains to show that $\vdash K' + v : B' \Rightarrow \mathsf{int}$ 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 $\vdash (K' + v)[e'] : \mathsf{int}$.

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

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. $\qquad\square$

Lemma 13 (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.) $\quad\square$

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

Proof. Invert $e \rightsquigarrow e'$ to obtain K, e_1, e_1' s.t. $e = K[e_1]$ and $e' = K[e_1']$ and $e_1 \rightsquigarrow_{\mathsf{b}} e_1'$.

By Lemma 12, there exists B s.t. $\vdash K : B \Rightarrow A$ and $\vdash e_1 : B$.

By Lemma 11, from $\vdash e_1 : B$ and $e_1 \rightsquigarrow_{\mathsf{b}} e_1'$, we have $\vdash e_1' : B$.

By Lemma 13, from $\vdash e_1' : B$ and $\vdash K : B \Rightarrow A$, we have $\vdash K[e_1'] : A$, so we are done. $\quad\square$

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

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 16 (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 $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\boxed{\mathcal{V}[\![A]\!]}$

$$\mathcal{V}[\![\mathsf{int}]\!] := \{\overline{n} \mid n \in \mathbb{Z}\}$$
$$\mathcal{V}[\![A \rightarrow B]\!] := \{\lambda x.\, e \mid \mathsf{fv}(e) \subseteq \{x\} \wedge \forall v.\, v \in \mathcal{V}[\![A]\!] \Rightarrow e[v/x] \in \mathcal{E}[\![B]\!]\}$$

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

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

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

Proof. Follows directly from $v \downarrow v$. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\square$

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.

Draft of November 13, 2023

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

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

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 $\boxed{\gamma(e)}$

$$\gamma(x) := \begin{cases} v & \text{if } \gamma(x) = v \\ x & \text{otherwise} \end{cases}$$

$$\gamma(\overline{n}) := \overline{n}$$
$$\gamma(\lambda x. e) := \lambda x. (\gamma[x \mapsto \bot]) e$$
$$\gamma(e_1 e_2) := \gamma(e_1) \gamma(e_2)$$
$$\gamma(e_1 + e_2) := \gamma(e_1) + \gamma(e_2)$$

We can now define a *semantic typing judgment.*

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

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

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

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 19 (Compatibility with INT). *We have $\Gamma \vDash n : \mathsf{int}$.*

Proof. To show $\Gamma \vDash n : \mathsf{int}$, we first show that $\mathsf{fv}(n) \subseteq \mathsf{dom}(\Gamma)$, which is immediate.
We may assume $\gamma \in \mathcal{G}[\![\Gamma]\!]$ and have to show $\gamma(n) \in \mathcal{E}[\![\mathsf{int}]\!]$.
By value inclusion (Lemma 17) and the definition of substitution, it suffices to show $n \in \mathcal{V}[\![\mathsf{int}]\!]$.
This follows directly from the definition of $\mathcal{V}[\![\_]\!]$. $\square$

Lemma 20 (Compatibility with VAR). *If $x : A \in \Gamma$ then $\Gamma \vDash x : A$.*

Proof. To show $\Gamma \vDash x : A$, we first show that $\mathsf{fv}(x) \subseteq \mathsf{dom}(\Gamma)$, 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 17), we are done. $\square$

Lemma 21 (Compatibility with PLUS). *If $\Gamma \vDash e_1 : \mathsf{int}$ and $\Gamma \vDash e_2 : \mathsf{int}$ then $\Gamma \vDash e_1 + e_2 : \mathsf{int}$.*

Proof. To show $\Gamma \vDash e_1 + e_2 : \mathsf{int}$, we first show that $\mathsf{fv}(e_1 + e_2) \subseteq \mathsf{dom}(\Gamma)$, which follows from $\mathsf{fv}(e_1) \subseteq \mathsf{dom}(\Gamma)$ and $\mathsf{fv}(e_2) \subseteq \mathsf{dom}(\Gamma)$.
We may assume $\gamma \in \mathcal{G}[\![\Gamma]\!]$ and have to show $\gamma(e_1) + \gamma(e_2) \in \mathcal{E}[\![\mathsf{int}]\!]$.
From the assumptions, there exist v_1, v_2 s.t. $\gamma(e_1) \downarrow v_1 \in \mathcal{V}[\![\mathsf{int}]\!]$ and $\gamma(e_2) \downarrow v_2 \in \mathcal{V}[\![\mathsf{int}]\!]$.
From $v_1 \in \mathcal{V}[\![\mathsf{int}]\!]$, 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 $\gamma(e_1) + \gamma(e_2) \downarrow \overline{n_1 + n_2} \in \mathcal{V}[\![\mathsf{int}]\!]$, which finishes the proof. $\qquad\square$

Lemma 22 (Compatibility with APP)**.** *If* $\Gamma \vDash e_1 : A \to B$ *and* $\Gamma \vDash e_2 : A$ *then* $\Gamma \vDash e_1\, e_2 : B$.

Proof. To show $\Gamma \vDash e_1\, e_2 : B$, we first show that $\mathsf{fv}(e_1\, e_2) \subseteq \mathsf{dom}(\Gamma)$, which follows from $\mathsf{fv}(e_1) \subseteq \mathsf{dom}(\Gamma)$ and $\mathsf{fv}(e_2) \subseteq \mathsf{dom}(\Gamma)$.
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. $\gamma(e_1) \downarrow v_1 \in \mathcal{V}[\![A \to B]\!]$ and $\gamma(e_2) \downarrow v_2 \in \mathcal{V}[\![A]\!]$.
From $v_1 \in \mathcal{V}[\![A \to B]\!]$, 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. $\qquad\square$

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

Proof. To show $\Gamma \vDash \lambda x.\, e : A \to B$, we first show that $\mathsf{fv}(\lambda x.\, e) \subseteq \mathsf{dom}(\Gamma)$, which follows from $\mathsf{fv}(e) \subseteq \mathsf{dom}(\Gamma, x : A)$.
We may assume $\gamma \in \mathcal{G}[\![\Gamma]\!]$ and have to show $\gamma(\lambda x.\, e) \in \mathcal{E}[\![A \to B]\!]$.
Unfolding the substitution and applying Lemma 17, it suffices to show $\lambda x.\, (\gamma[x \mapsto \bot])(e) \in \mathcal{V}[\![A \to B]\!]$.
We have to show $\mathsf{fv}((\gamma[x \mapsto \bot])(e)) \subseteq \{x\}$, which follows because $\mathsf{fv}(e) \subseteq \mathsf{dom}(\Gamma, x : A)$ and everything in γ is in $\mathcal{V}[\![\_]\!]$ and hence closed.
We suppose $v \in \mathcal{V}[\![A]\!]$ and have to show $((\gamma[x \mapsto \bot])(e))[v/x] \in \mathcal{E}[\![B]\!]$.
We have $((\gamma[x \mapsto \bot])(e))[v/x] = \gamma[x \mapsto v](e)$ 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 $v \in \mathcal{V}[\![A]\!]$. $\qquad\square$

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 $(\gamma[x \mapsto \bot])(e)$) 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 24 (Semantic Soundness)**.** *If* $\Gamma \vdash e : A$, *then* $\Gamma \vDash e : A$.

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

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

Proof. By Theorem 24, we have $\emptyset \vDash e : A$. 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, $\exists v.\, e \downarrow v$. $\qquad\square$

Draft of November 13, 2023

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:

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 $\lambda.\, \widehat{0}$. 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 $\lambda.\, \lambda.\, \widehat{0}$: the variable refers to the innermost λ. The term $\lambda x.\, \lambda y.\, x$ is represented as $\lambda.\, \lambda.\, \widehat{1}$, since now the variable refers to the λ "one layer out".

The term from the example picture above now becomes:

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

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.

One consequence of this definition is that the same variable has a different number in different situations. For example, $\lambda.\,\widehat{0} + (\lambda.\,\widehat{1})\,\widehat{0}$ 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 updated by 1 to refer to the outer λ.

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

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

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 [2] 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 $e[e'/\,]$:

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

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

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 [3]. 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:

$$(e_1\ e_2)[\sigma] := e_1[\sigma]\ e_2[\sigma]$$
$$x[\sigma] := \sigma(x)$$
$$(\lambda.\,e)[\sigma] := \lambda.\,e[\Uparrow \sigma]$$

where the "shift" operation $\Uparrow$ on substitutions is defined as:

$$(\Uparrow \sigma)(n) := \begin{cases} \widehat{0} & \text{if } n = 0 \\ \sigma(n-1)[\widehat{\mathsf{S}}] & \text{otherwise} \end{cases}$$

And the "successor" substitution $\widehat{\mathsf{S}}$ is defined as:

$$\widehat{\mathsf{S}}(n) := \widehat{n+1}$$

In other words, substituting with $\widehat{\mathsf{S}}$ 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:

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

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

$$(\lambda.\,\widehat{1})[\lambda.\,\widehat{0} + \widehat{1}/\ ] = \lambda.\,\lambda.\,\widehat{0} + \widehat{2}$$
$$(\widehat{0}\,\widehat{5} + \widehat{1})[\lambda.\,\widehat{0} + \widehat{1}/\ ] = (\lambda.\,\widehat{0} + \widehat{1})\,\widehat{5} + \widehat{0}$$

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 $\widehat{\mathsf{S}}$ 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 $\boxed{e_1 \leadsto_{\mathrm{b}} e_2}$

$$
\begin{array}{c}
\text{BETA} \\
(\lambda.\, e)\, v \leadsto_{\mathrm{b}} e[v/\;]
\end{array}
$$

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 $\boxed{\Gamma(n) = A}$

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

$$
\begin{array}{cc}
\begin{array}{c}
\text{HERE} \\
(\Gamma, A)(0) = A
\end{array}
&
\begin{array}{c}
\text{THERE} \\
\dfrac{\Gamma(n) = A}{(\Gamma, B)(n+1) = A}
\end{array}
\end{array}
$$

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

Then we define typing on runtime terms as follows:

$$
\begin{array}{ccc}
\begin{array}{c}
\text{VAR} \\
\dfrac{\Gamma(n) = A}{\Gamma \vdash \widehat{n} : A}
\end{array}
&
\begin{array}{c}
\text{LAM} \\
\dfrac{\Gamma, A \vdash e : B}{\Gamma \vdash \lambda.\, e : A \to B}
\end{array}
&
\begin{array}{c}
\text{APP} \\
\dfrac{\Gamma \vdash e_1 : A \to B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\, e_2 : B}
\end{array}
\end{array}
$$

$$
\begin{array}{cc}
\begin{array}{c}
\text{PLUS} \\
\dfrac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{int}}
\end{array}
&
\begin{array}{c}
\text{INT} \\
\Gamma \vdash \overline{n} : \mathsf{int}
\end{array}
\end{array}
$$

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:

$$
\vdash \lambda.\, \overline{1} + \widehat{0} : \mathsf{int} \to \mathsf{int}
$$

$$
\mathsf{int} \vdash \lambda.\, \widehat{1} + \widehat{0} : \mathsf{int} \to \mathsf{int}
$$

$$
\mathsf{int} \to \mathsf{int}, \mathsf{int} \vdash \widehat{1}\, \widehat{0} : \mathsf{int}
$$

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:

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

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 26 (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 substituion* which is checked in context Γ_2 and assigns types to all variables in Γ_1:

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

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 27 (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 9) and then the substitution lemma (Lemma 10). 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 28 (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 10: 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 $\boxed{\mathcal{V}[\![A]\!]}$

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

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

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

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

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

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

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:

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

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

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 $((\gamma[x \mapsto \bot])(e))[v/x] \in \mathcal{E}[\![B]\!]$ in the old proof (go back to Lemma 23 to compare). The $e[\Uparrow \gamma]$ arises from $(\lambda.\, e)[\gamma]$, which is part of the definition of $\Gamma \vDash \lambda.\, e : A \to B$, 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 $((\gamma[x \mapsto \bot])(e))[v/x] = \gamma[x \mapsto v](e)$ 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

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

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 $e' :: \mathsf{id}$, where id is the "identity" substitution which maps n to $\widehat{n}$.)

$e[\Uparrow \gamma][v/\,] = e[v :: \gamma]$ 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 [3] 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.

References

[1] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 2nd revised edition, 1984.

[2] S. Schäfer, T. Tebbi, and G. Smolka. Autosubst: Reasoning with De Bruijn terms and parallel substitutions. In *Interactive Theorem Proving*, 2015.

[3] K. Stark. Mechanising syntax with binders in Coq, 2019.

Draft of November 13, 2023