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

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.

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

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

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 $\overline{2} + x$ 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* [3]: all bound and free variables are distinct and this invariant is maintained implicitly.

Since we *mechanize* our proofs in Coq, we cannot assume Barendregt's variable convention. Instead, we use the (slightly broken) substitution operation above. In our use cases, the substitution will only insert *closed* terms (*i.e.*, terms without any free variables: $\mathsf{fv}(e) = \emptyset$), which avoids the problem of variable capture entirely. (In later sections, Section **??**, 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.

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

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:

$$\textsc{app-struct-r}\ \frac{e_2 \succ e_2'}{e_1\ e_2 \succ e_1\ e_2'} \qquad \textsc{app-struct-l}\ \frac{e_1 \succ e_1'}{e_1\ v_2 \succ e_1'\ v_2} \qquad \textsc{beta}\ (\lambda x.\, e)\ v \succ e[v/x]$$

$$\textsc{plus-struct-l}\ \frac{e_1 \succ e_1'}{e_1 + v_2 \succ e_1' + v_2} \qquad \textsc{plus-struct-r}\ \frac{e_2 \succ e_2'}{e_1 + e_2 \succ e_1 + e_2'} \qquad \textsc{plus}\ \overline{n} + \overline{m} \succ \overline{n + m}$$

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.

Exercise 2 Prove that the structural semantics $e \succ e'$ is *deterministic*. That is, show that if $e \succ e'$ and $e \succ e''$, then $e' = e''$. •

Exercise 3 The semantics $e \succ e'$ is a *call-by-value* semantics, meaning arguments are evaluated to values before they are inserted into lambda abstractions. The call-by-value approach is used by many programming languages (e.g., Java, C, Standard ML, and OCaml). Alternatively, one can defer the evaluation of function arguments and, instead, insert their unevaluated form directly into lambda abstractions. This style of operational semantics is called *call-by-name* semantics and is followed by some functional languages (e.g., Haskell). The core rule of this semantics is:

$$\frac{}{(\lambda x.\, e)\, e' \succ_{\mathsf{cbn}} e[e'/x]}\ \text{CBN-BETA}$$

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

•

Exercise 4 Prove that the big-step and the small-step semantics $e \succ e'$ are equivalent:

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

Here, $\succ^*$ is the reflexive transitive closure of $\succ$, *i.e.*, $e \succ^* v$ means that we can step from e to v using a sequence of reduction steps (including possibly using no steps at all). •

Exercise 5 The evaluation order for $e \succ e'$ is right-to-left. We are now going to consider left-to-right evaluation order.

1. Define a semantics $e \succ_{\mathsf{ltr}} e'$, which evaluates expressions in left-to-right order.
2. Pick terms e and e' such that $e \succ_{\mathsf{ltr}} e'$ but not $e \succ e'$.
3. Show that both are equivalent if we evaluate to values.
   Hint: It suffices to show $e \succ^*_{\mathsf{ltr}} v \quad \textit{iff} \quad e \downarrow v$.
4. Think of the programming languages you have encountered in your past. Is it in one of them possible to obtain different results, depending on left-to-right or right-to-left evaluation order?

•

Contextual Semantics

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

For the contextual semantics, we first define evaluation contexts:

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

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

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

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

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

$$\frac{}{(\lambda x.\, e)\,v \rightsquigarrow_{\mathrm{b}} e[v/x]}\ \text{BETA} \qquad \frac{}{\overline{n} + \overline{m} \rightsquigarrow_{\mathrm{b}} \overline{n+m}}\ \text{PLUS} \qquad \frac{e_1 \rightsquigarrow_{\mathrm{b}} e_2}{K[e_1] \rightsquigarrow K[e_2]}\ \text{CTX}$$

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 $e_1' \rightsquigarrow_{\mathrm{b}} e_2'$ and $e_1 = K'[e_1']$ and $e_2 = K'[e_2']$. To construct a proof of $K[K'[e_1']] \rightsquigarrow K[K'[e_2']]$, 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. □

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

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

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

Proof Sketch. By induction on K_1.

Exercise 6 As an example for a reasoning principle that can be stated more elegantly in the contextual semantics than the structural semantics, we consider context lifting for the reflexive transitive closure of $\rightsquigarrow$. Prove the following: if $e_1 \rightsquigarrow^* e_2$, then $K[e_1] \rightsquigarrow^* K[e_2]$.

To obtain 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 state. •

Exercise 7 Prove that the structural and the contextual semantics are equivalent:

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

•

1.2 The Untyped λ-calculus

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

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
  O => 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:

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

(We use $\lambda z, s.\, z$ as a short-hand for $\lambda z.\, \lambda s.\, z$.)

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

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

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. (Note that function application is left-associative, so $n\ \widetilde{0}\ e$ is short for $(n\ \widetilde{0})\ e$.)

Exercise 8 Define a function on Scott encoded natural numbers which returns the identity for 0 and diverges for every other 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, 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 mul2 with fix. We first define the template MUL2, and then we take the fixpoint to obtain mul2:

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

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

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

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

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

The proof is simple but tedious; the key point is showing that mul2 satisfies $\mathsf{mul2}\ \widetilde{0} \succ^* \widetilde{0}$ and $\mathsf{mul2}\ \widetilde{S}\, n \succ^* \widetilde{S}\ (\widetilde{S}\ (\mathsf{mul2}\ \widetilde{n}))$. 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 $\mathsf{fix}\ F$ (where fix is parametric in F) consists of two parts:

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

The idea is that fix is defined using a term fix', which relies on self-application (like Ω) to implement recursion. 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 $\mathsf{fix}\ F$. What this means in the scope of fix' is that we apply F to the term $\lambda x.\, f\ f\ F\ x$.

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

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

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

1.3 Typing

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

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

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

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

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

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

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

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

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

$$\text{INT}\ \Gamma \vdash \overline{n} : \mathsf{int}$$

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

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

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

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.

Exercise 9 (Typing Uniqueness) Prove that Church typing is unique:

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

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

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

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

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

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

Exercise 10 Prove the Erasure lemma. •

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

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 BETA we have $e_1 e_2 \rightsquigarrow e[e_2/x]$.
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 $K := (e_1\, \bullet)$.

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 (what we are "getting out" of the induction) 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: $e = \lambda x.\, e'$, $A = A' \to B'$.
  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 LAM applied to 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 applied to 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.

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: $\Gamma, y : B_1, x : A \vdash e_1 : B$.
    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 $\Gamma, x : A, y : B_1$.
- 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 weakening (Lemma 9).

□

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. □

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. We write $IH(K)$ for the induction statement

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

- $K = \bullet$:
  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 $\vdash e : A$.
  Both of these we already have as assumptions.
  (Explanation: we assume some $\vdash e : A$ and have to show $\vdash \bullet[e] : A$)

- $K = K'\ v$:
  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 $\vdash K'[e] : A' \to A$ 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; the first is done using APP and the typing of K' and v.
  (Explanation: we assume some $\vdash e : B'$ and have to show $\vdash (K'\ v)[e] : A$)

- $K = e'\ K'$:
  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; the first is done using APP.

- $K = K' + v$:
  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 $A = \mathsf{nat}$, $\vdash K'[e] : \mathsf{nat}$, and $\vdash v : \mathsf{nat}$.
  Applying $IH(K')$ to the second, we find there is some B' so that $\vdash K' : B' \Rightarrow \mathsf{nat}$ and $\vdash e : B'$.
  We are done by taking $B := B'$ and using PLUS.

- $K = e' + K'$:
  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 $A = \mathsf{nat}$, $\vdash e' : \mathsf{nat}$, and $\vdash K'[e] : \mathsf{nat}$.
  Applying $IH(K')$ to the latter, we find there is some B' so that $\vdash K' : B' \Rightarrow \mathsf{nat}$ and $\vdash e : B'$.
  We are done by taking $B := B'$ and using PLUS.

□

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.) □

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 $e_1 \rightsquigarrow_{\mathrm{b}} e_1'$.
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 $e_1 \rightsquigarrow_{\mathrm{b}} e_1'$, 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. □

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

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

References

[1] A. Ahmed. *Semantics of types for mutable state*. PhD thesis, Princeton University, 2004.

[2] A. W. Appel and D. A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *TOPLAS*, 2001.

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

[4] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP*, 1983.

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

[6] P. Wadler. Theorems for free! In *FPCA*, 1989.