Erasure

For most languages, **types are not needed at run-time**.

Consider this **type erasure function**.

$$
\begin{array}{ll}
\mathit{erase}(x) & = x \\
\mathit{erase}(\lambda x : T_1.t_2) & = \lambda x.\mathit{erase}(t_2) \\
\mathit{erase}(t_1\ t_2) & = \mathit{erase}(t_1)\ \mathit{erase}(t_2)
\end{array}
$$

By careful design, we have:

$$t \to t' \implies \mathit{erase}(t) \to \mathit{erase}(t').$$

But also

$$\mathit{erase}(t) \to m' \implies \exists t' \mid t \to t' \land \mathit{erase}(t') = m'$$

Proved by induction on evaluation derivations.

Curry Style language definition

The approach we have followed:

- Start with terms representing desired behaviours (syntax).
- Formalize those behaviours using evaluation rules (semantics).
- Use a typing system to reject undesired behaviours (typing).

This is often called a **Curry-Style** language definition, because semantics are given priority over typing.

i.e., we can remove the typing and still have a functional system.

Church Style language definition

A different approach is as follows:

- Start with terms representing desired behaviours (syntax).
- Identify the well-typed terms using typing rules (typing).
- Give semantics **only** to well-typed terms (semantics).

Under **Church-Style** language design, typing is given priority.

- Questions like "How does an ill-typed term behave?" don't occur, because ill-typed term cannot *even be evaluated!*
- Historically:
  - Explicitly typed languages have normally been presented Church-Style.
  - Implicitly typed languages have normally been presented Curry-Style.
- Thus Church-style is sometimes confused with explicit typing (and vice-versa for Curry).

Atomic Types

PLs provides a set of atomic types, often including:

- Booleans ($\mathbb{B}$), Natural Numbers ($\mathbb{N}$), Integers ($\mathbb{Z}$), Characters, Strings, etc.
- **Do not confuse** floats for Real Numbers ($\mathbb{R}$)! We will avoid all talk of both floats and reals in this course.

These are sometimes known as **primitives**. These are normally accompanied by a set of **primitive operations**, such as:

- $+$, $-$, $\times$, $==$, &&, $||$, etc.

Adding these is very easy, with the only difficulty appearing when we try to add *partial* functions.

Atomic Type Semantics

Augment language with a set $\mathcal{A}$ of **uninterpreted** base types.

→ $\boxed{\mathsf{A}}$ *Extends* $\lambda_{\to}$ *(9-1)*

New syntactic forms			
T	::=	...	*types:*
		$\boxed{\mathsf{A}}$	*base type*

Helpful in the following examples:

$$(\lambda x : A.x) : A \Rightarrow A$$

$$(\lambda f : A \Rightarrow A.\lambda x : A.f\ (f\ x)) : (A \Rightarrow A) \Rightarrow A \Rightarrow A$$

Statements

What does := *return* ?

Statements

What does := *return* ? It doesn't, but it has a **side-effect** on *memory*.

So: how do we *type* side-effects?
Let us first do "sequencing". Easiest done by first introducing a *Unit* type.

Unit Type Semantics

$\rightarrow$ `Unit` *Extends $\lambda_{\rightarrow}$ (9-1)*

New syntactic forms

t	::=	...	*terms:*
		`unit`	*constant* `unit`
v	::=	...	*values:*
		`unit`	*constant* `unit`
T	::=	...	*types:*
		`Unit`	*unit type*

New typing rules $\boxed{\Gamma \vdash t : T}$

$$\Gamma \vdash \texttt{unit} : \texttt{Unit} \qquad \text{(T-UNIT)}$$

New derived forms

$$t_1 ; t_2 \stackrel{\text{def}}{=} (\lambda x{:}\texttt{Unit}.t_2)\ t_1 \quad \text{where } x \notin FV(t_2)$$

Sequencing

In languages with side effects, want to "execute" some commands.

Solution? Make commands return value *unit*.
Sequencing of commands is then denoted ;
As usual, can add it to language as a new term, or make it derived.

As a New Term

Grammar:

$\langle t \rangle$ ::= ...
  | $\langle t \rangle$; $\langle t \rangle$

Evaluation rules:

$$\frac{t_1 \to t_1'}{t_1; t_2 \to t_1'; t_2} \qquad \text{(E-Seq)}$$

$$\texttt{unit}; t_2 \to t_2 \qquad \text{(E-SeqNext)}$$

Typing rule

$$\frac{\Gamma \vdash t_1 : \mathit{Unit} \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash (t_1; t_2) : T_2}$$

Derived Form Approach

As smaller languages mean smaller proofs...

$$t_1; t_2 \stackrel{def}{=} (\lambda x : Unit. t_2)\ t_1 \tag{1}$$

Which throws away the value associated to t_1 (in call-by-value semantics), and yields t_2

Surface vs Core Language

Derived forms are everywhere in modern programming languages, where they are often called **syntactic sugar**.

- They allow the programmer to use the language more easily by providing abstractions of the language used by the compiler.
- Ultimately, however, programs must be **desugared** before object code generation.
  - Higher-level constructs are replaced with equivalent terms in the core language.
- This forms the distinction between:
  - The **external language**, or that of the programmer.
  - The **internal language**, or what the compiler (eventually) works with.

Sequencing is a Derived Form

Definition

$\lambda^{\mathcal{E}}$ as the simply typed λ-Calculus, enriched with Unit, unit, $t_1; t_2$, E-Seq, E-SeqNext, and T-Seq.

Definition

$\lambda^{\mathcal{I}}$ as the simply typed λ-Calculus, Unit type and unit term only.

Define $e \in \lambda^{\mathcal{E}} \to \lambda^{\mathcal{I}}$ as a meta-level **elaboration function**. It replaces all instances of $t_1; t_2$ with $(\lambda x : Unit.t_2)\ t_1$.

THEOREM [Sequencing is a Derived Form] For each term t of $\lambda^{\mathcal{E}}$, we have:

$$t \xrightarrow{\mathcal{E}} t' \iff e(t) \xrightarrow{\mathcal{I}} e(t')$$

$$\Gamma \vdash^{\mathcal{E}} t : T \iff \Gamma \vdash^{\mathcal{I}} e(t) : T$$

Ascription Semantics

"We say that t has type T" – this is **not** casting!

$\rightarrow$ as *Extends $\lambda_{\rightarrow}$ (9-1)*

New syntactic forms

t	::=	...	*terms:*
		t as T	*ascription*

New evaluation rules $\boxed{t \longrightarrow t'}$

$$v_1 \text{ as } T \longrightarrow v_1 \qquad \text{(E-Ascribe)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1 \text{ as } T \longrightarrow t_1' \text{ as } T} \qquad \text{(E-Ascribe1)}$$

New typing rules $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \vdash t_1 : T}{\Gamma \vdash t_1 \text{ as } T : T} \qquad \text{(T-Ascribe)}$$

This is most useful once we introduce **polymorphism**, but is already useful as **documentation**.

Let Bindings: naming sub-expressions

Semantically, we want $(let\ x = t_1\ in\ t_2)$ to evaluate to a substitution of x for t_1 in t_2.

→ `let` *Extends* $\lambda_{\to}$ *(9-1)*

New syntactic forms

$\mathtt{t} ::= \ldots$ *terms:*

$\mathtt{let\ x{=}t\ in\ t}$ *let binding*

New evaluation rules $\boxed{\mathtt{t} \longrightarrow \mathtt{t'}}$

$$\mathtt{let\ x{=}v_1\ in\ t_2} \longrightarrow [\mathtt{x} \mapsto \mathtt{v_1}]\mathtt{t_2} \quad \text{(E-LetV)}$$

$$\frac{\mathtt{t_1} \longrightarrow \mathtt{t_1'}}{\mathtt{let\ x{=}t_1\ in\ t_2} \longrightarrow \mathtt{let\ x{=}t_1'\ in\ t_2}} \quad \text{(E-Let)}$$

New typing rules $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\Gamma \vdash \mathtt{t_1} : \mathtt{T_1} \qquad \Gamma, \mathtt{x}{:}\mathtt{T_1} \vdash \mathtt{t_2} : \mathtt{T_2}}{\Gamma \vdash \mathtt{let\ x{=}t_1\ in\ t_2} : \mathtt{T_2}} \quad \text{(T-Let)}$$

Let semantics

Intuitively, we want:

$$let\ x = t_1\ in\ t_2 \stackrel{def}{=} (\lambda x : T_1.t_2)\ t_1$$

But where does T_1 come from?
Best to think of `let-in` as a *fusion* of λ and application. We have t_1 in our hands, use it!

- Have two options:
  - Regard elaboration as a transformation on typing derivations.
  - **Decorate** terms with the results of typechecking.

  So: evaluation semantics of let bindings can be desugared, but the typing behaviour *must* be built into the inner language.