#### Erasure

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

Consider this type erasure function.

$$erase(x) = x$$
  
 $erase(\lambda x : T_1.t_2) = \lambda x.erase(t_2)$   
 $erase(t_1 t_2) = erase(t_1) erase(t_2)$ 

By careful design, we have:

$$t \rightarrow t' \implies \textit{erase}(t) \rightarrow \textit{erase}(t').$$

But also

$$erase(t) 
ightarrow m' \implies \exists t' \mid t 
ightarrow t' \land erase(t') = m'$$

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 A of **uninterpreted** base types.

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

New syntactic forms

types: 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?

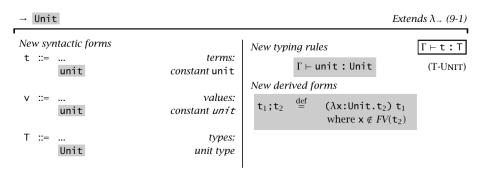
### **Statements**

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**



### 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:

$$rac{t_1
ightarrow t_1'}{t_1;\,t_2
ightarrow t_1';\,t_2}$$
 unit;  $t_2
ightarrow t_2$  (E

(E-Seq)

Typing rule

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

### Derived Form Approach

As smaller languages mean smaller proofs...

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

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

# 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  $\lambda$ -Calculus, enriched with Unit, unit,  $t_1$ ;  $t_2$ , E-Seq, E-SeqNext, and T-Seq.

#### **Definition**

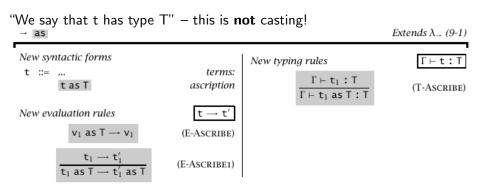
 $\lambda^{\mathcal{I}}$  as the simply typed  $\lambda$ -Calculus, Unit type and unit term only.

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:

$$t \xrightarrow{\mathcal{E}} t' \iff e(t) \xrightarrow{\mathcal{I}} e(t')$$
$$\Gamma \vdash^{\mathcal{E}} t : T \iff \Gamma \vdash^{\mathcal{I}} e(t) : T$$

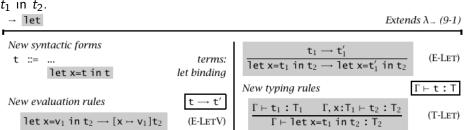
## **Ascription Semantics**



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 semantics

Intuitively, we want:

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

But where does  $T_1$  come from?

Best to think of let-in as a fusion of  $\lambda$  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.