COMPSCI 3MI3 - Principles of Programming Languages

Topic 9 - External vs Internal Languages

NCC Moore

McMaster University

Fall 2021

Adapted from "Types and Programming Languages" by Benjamin C. Pierce

The Curry Howard Correspondence

Erasure and Typeability

Church-Style vs. Curry-Style

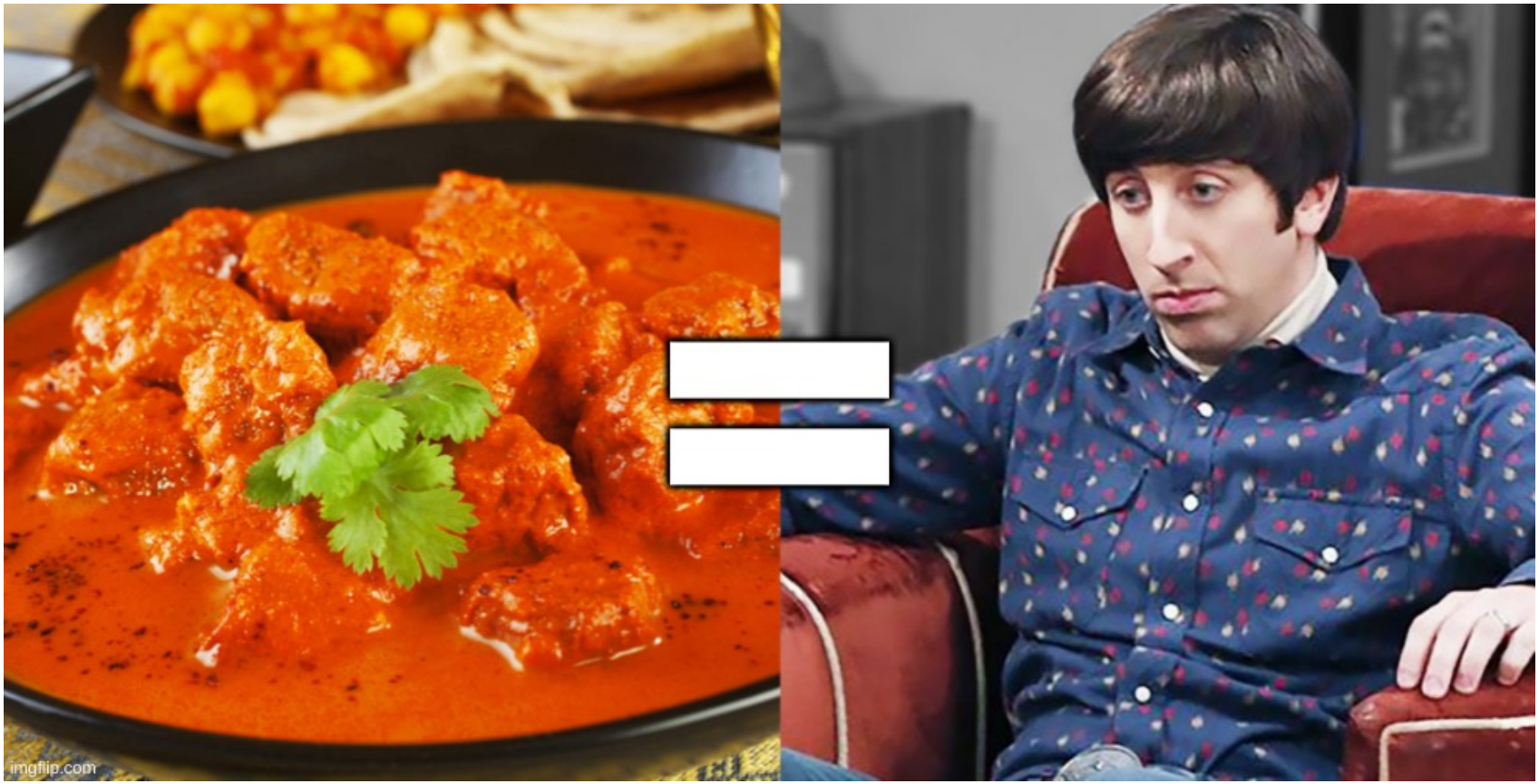
Atomic Types

Formalizing Sequencing

Ascription

Let Bindings

The Curry Howard Correspondence



Reasoning with Typing Relations

Consider the $\Rightarrow$ type constructor.

- We have a mechanism for **introducing** it (T-Abs)
- We have a mechanism for **eliminating** it (T-App)

If you go on to study advanced topics in type theory (beyond the scope of this course), you will find that many type constructors use introduction and elimination rules, and that this is often how we talk about them.

But what else do we have rules for introducing and eliminating?

I Don't Like the Implication!

That's right folks! It's our old friend **implication**, from propositional logic!

- With natural deduction, we **introduce** implication by proposing a hypothesis, and demonstrating that some other term of the logic can be derived from it.

$$\Gamma, A \vdash B \tag{1}$$

can be rewritten as

$$\Gamma \vdash A \implies B \tag{2}$$

- We **eliminate** implication by combining an implication with the premise of the implication. i.e.,

$$A \wedge (A \implies B) \vdash B \tag{3}$$

The Curry Howard Correspondence

It turns out that there is more than a little conceptual overlap between type systems and logic.

Logic	Type Theory
Propositions	Types
$P \subset Q$	$P \Rightarrow Q$
$P \wedge Q$	$P \times Q$[1]
Proof of proposition P	Proof of $t : P$
Proposition P is provable	type P is inhabited

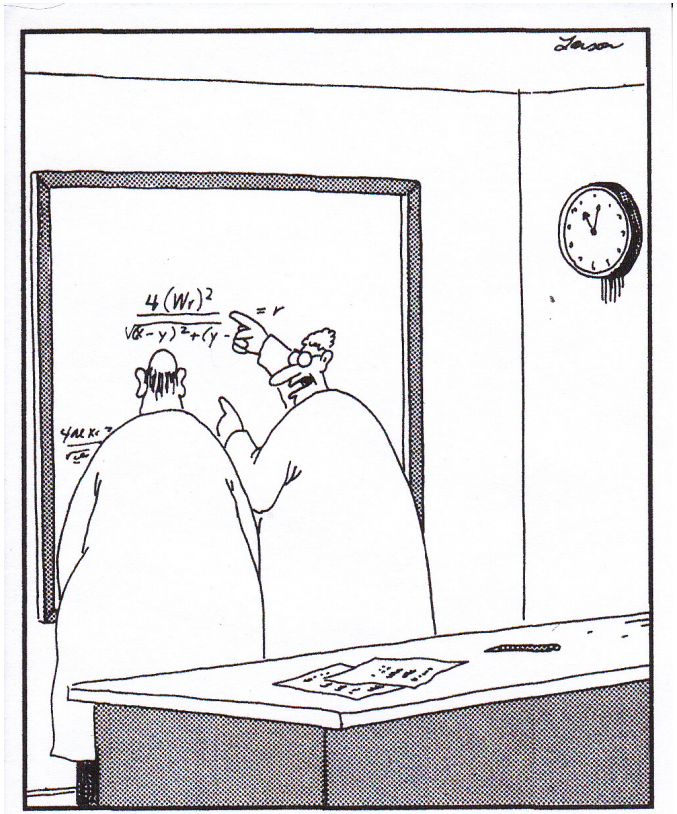
[1] we'll see this one a bit later on.

I Wonder if Haskell Curry Even Liked Curry

As this property was being investigated (historically speaking), a greater and greater correspondence was found.

- The advancement of mathematical logic as a science has yielded great benefits in type theory, and vice versa!
  - Essentially, the fields of Type Theory and Mathematical Logic can cheat off each other's homework.
- Type System F, which uses quantification over parametric polymorphic types has its correspondence in second-order constructive logic.
- *Linear Logic* (Girard, 1987) led to the development of *Linear Type Systems* (Wadler 1990, and many others)
- *Modal Logics* have been used to help design frameworks for *partial evaluation* and *run-time code generation* (Davies and Pfenning, 1996).

Erasure and Typeability



"Yes, yes, I *know* that, Sidney... *every*body knows *that!*... But look: Four wrongs *squared*, minus two wrongs to the fourth power, divided by this formula, *do* make a right."

Practical Considerations

Let's consider for a moment how we would use a type system in a real-world compiler (that is, in compiled languages).

- Ordinarily, typechecking is a **preprocessing** step.
  - Essentially, we use typechecking to verify the integrity of a program's AST.
  - Used in this manner, it is easy to confuse a type error with a syntax error.
- Some advanced programming languages use type annotations during **generation** as well.
- However, typing information *almost never* makes it into the **object code**.
  - Typing information is only rarely relevant to **execution**.

Erased... From History!

At some point during compilation, typing annotations are normally removed to improve performance.

- We can formalize this idea with a **type erasure function**.

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

- This operation is recursive traversal of our term, removing typing information from λ's as we go.
- Because we have carefully kept our typing system separate from our evaluation semantics, we can remove typing information with no effect on a program's actual execution. i.e.,

$$t \to t' \implies \mathit{erase}(t) \to \mathit{erase}(t'). \tag{4}$$

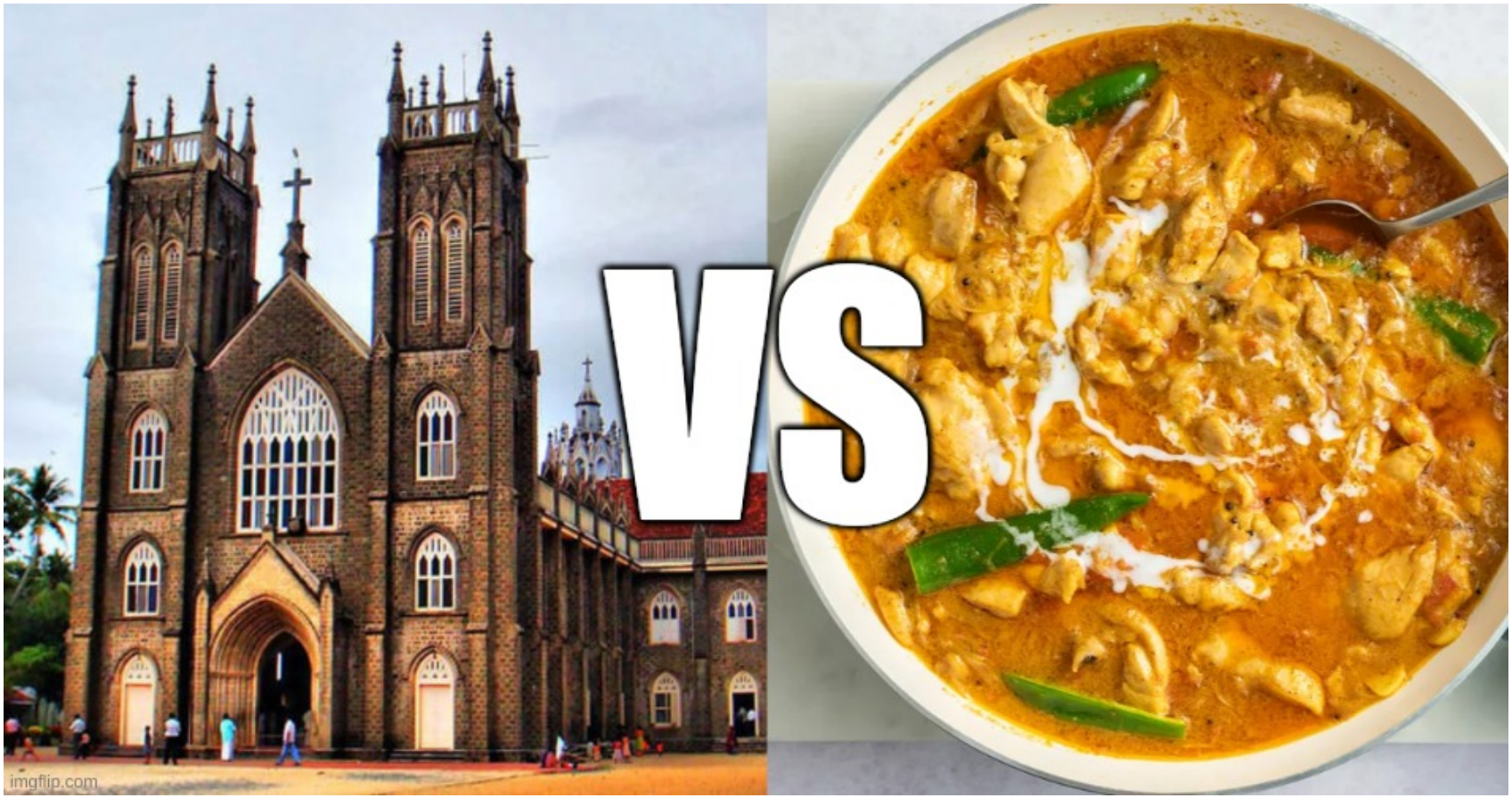
DELETE!! DELETE!!

Another interesting property of type erasure is as follows:

$$erase(t) \to m' \implies \exists t' \mid t \to t' \wedge erase(t') = m' \tag{5}$$

- This is to say, if we erase the types from a term t and evaluate it one step, this implies the existence of a typed term t' which, once erased, is equivalent to m'.
- The gist of these two properties is that it doesn't matter if we erase or evaluate first, we get to the same result.
- Both properties can be proved almost trivially using induction on evaluation derivations.

Church-Style vs. Curry-Style



Curry Style!

Our general approach to language design in this class has been:

- Start with some terms representing desired behaviours (syntax).
- Formalize those behaviours using evaluation rules (semantics).
- Apply 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!

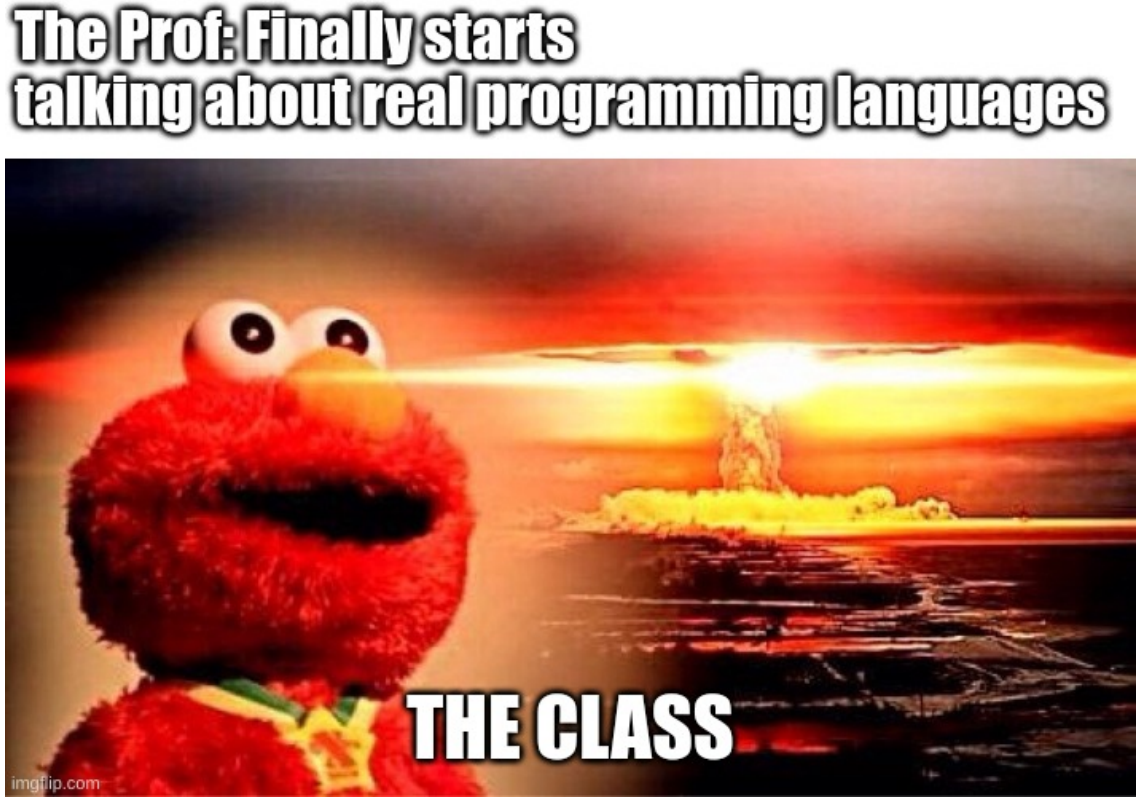
A different approach is as follows:

- Start with some 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 no ill-typed term can *even be evaluated!*
- Historically,
  - Explicitly typed languages have normally been presented Church-Style.
  - Implicitly typed languages have normally been presented Curry-Style.
- So, Church-style is sometimes confused with explicit typing (and the same in reverse for Curry).

Atomic Types



Only now, in the first week of November, can we begin to have an intelligent conversation about the design of programming languages.

Atomic Types

Every programming language provides a set of atomic types, which normally includes:

- Booleans ($\mathbb{B}$), Unsigned Integers ($\mathbb{N}$), Integers ($\mathbb{Z}$), Real Numbers ($\mathbb{R}$), Characters, Strings, etc.

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

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

We have discussed $\mathbb{B}$ and $\mathbb{N}$ at length so far. The rest of these operators and types can be added to our language semantics in a similar way.

- For the rest of this course, we will occasionally bring in the rest of these data types, in order to make our examples more interesting.

Abstract All The Types!

In order to reason with our ever expanding language definitions, it will be useful to abstract away the details of these types and their operations.

- Let's suppose that our language comes with some set $\mathcal{A}$ of **uninterpreted**, or *unknown* base types.
- Let's also suppose we have *no primitive operations* to worry about.

We can accomplish this by including the elements of $\mathcal{A}$ in the set of types T for our language.

Atomic Type Semantics

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

New syntactic forms

T	::=	...	*types:*
		A	*base type*

Note that this extends simply typed pure λ-Calculus.

- We will use A, B, C, etc. as the names of base types.
- We will also use A to mean a **meta-variable** which ranges over the atomic types.
  - The two uses will always be distinguishable from context.

Operation Consternation

It doesn't seem very useful to have a variable for a bunch of types we haven't specified, especially when they don't even do anything.

- Actually, this is not so different from the way we've been using types so far.
- This allows us to assign types to variables in λ abstractions, and reason about them.

For Example:

$$(\lambda x : A.x) : A \Rightarrow A \tag{6}$$

Is an identity function on some type A.

$$(\lambda f : A \Rightarrow A.\lambda x : A.f\ (f\ x)) : (A \Rightarrow A) \Rightarrow A \Rightarrow A \tag{7}$$

Is a function that repeats twice the behaviour of some given function f on an argument x.

Now Things are Getting Interesting!

So far, our languages have consisted entirely of stateless expression simplifications.

- If we want to reason about "real world languages," it will be necessary to find formal mathematical expression for the concepts of **assignment** and **sequencing**.
  - In essence, we need to move from just talking about expressions to also talking about **statements**.
- Consider a simple imperative assignment operation in a language like C.
  - What does it return?
  - What type would it have?

True to form, we will answer these questions as laboriously as possible.

Assignment

In general, we don't think of assignment statements[2] as directly returning anything.

- The interesting part of an assignment statement is *the effect it has on memory*.
- In other words, we want a particular **side-effect**.
  - Side effects include memory manipulation, displaying data using `stdout`, taking keyboard inputs, and many other useful things computers do.

Assuming our goal for Type Theory as a science is for all correct behaviour to be well-typed, how would we go about giving a type to such operations?

[2] More on assignment next week folks!

Enter the Unit Type

Essentially, we want something like a blank or empty type.

- We shall design such a type: *Unit*.
- In the last section, the Atomic Types were **uninterpreted**. That is, we didn't care about the internal details or evaluation mechanisms.
- *Unit* will be **interpreted**, but very simple.
- Semantically, we want *Unit* to behave roughly like the `void` type in languages like C and Java.

Unit Type Semantics

→ Unit — *Extends* $\lambda_{\to}$ *(9-1)*

New syntactic forms

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

New typing rules

$$\Gamma \vdash \texttt{t} : \texttt{T}$$

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

New derived forms

$$\texttt{t}_1;\texttt{t}_2 \stackrel{\text{def}}{=} (\lambda \texttt{x:Unit}.\texttt{t}_2)\ \texttt{t}_1 \quad \text{where } \texttt{x} \notin FV(\texttt{t}_2)$$

The above semantics extend simply typed pure λ-Calculus.

- Don't worry about $t_1; t_2$. That's sequencing and we're talking about it next.

Unit Type Semantics

In the above semantics...

- We introduce not only a new type, *Unit*, but a new term `unit`.
- We also define `unit` as a value.
- We have given no evaluation rules to `unit`, so it is a *normal form*.
- The new typing rule T-Unit sets the term `unit` as having type *Unit*.
  - In the absence of further additions, we can easily derive an inversion lemma and canonical form from this description.

Sequencing and Wildcards

Sequencing

In languages with side effects, it is often useful to evaluate two or more expressions in sequence.

- In programming languages (that aren't snake language), this is achieved with ;
- For this course, we will consider $t_1; t_2$ to be a **sequencing operator**.
- Semantically, we want ; to execute t_1, throw away it's trivial direct result, and then execute t_2.
  - We assume that t_1 has some kind of useful side effect, so that it's not completely pointless.
- In terms of typing, we expect $t_1 : \mathit{Unit}$, and make no constraints on t_2.

Going to the Formal

In general, there are two approaches to formalizing ;.

- We can define a new term for it, complete with typing and evaluation rules.
- We can define ; as a **derived form**. That is, we define it using our existing terms, requiring no new evaluation or typing rules.

A Wild New Term Appears!

Defining sequencing as a separate term, we would start with the following grammar:

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

To this we would add the following evaluation rules:

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

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

With the typing rule

$$\frac{\Gamma \vdash t_1 : \mathit{Unit} \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash (t_1; t_2) : T_2} \qquad (8)$$

Derived Form Approach

If this course has taught us one thing, it's that we should always seek to *minimize the number of terms in our language*.

- Each term we add makes it more cumbersome to prove properties of our language.

Sequencing can be captured within our existing semantics by the following expression:

$$t_1; t_2 \stackrel{def}{=} (\lambda x : Unit.t_2)\ t_1 \tag{9}$$

Which, of course, throws away t_1, once it has been evaluated to a value under call-by-value semantics, and simply yields t_2

Internal vs External Languages

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 inner language.
- This forms the distinction between:
  - The **external language**, or that of the programmer.
  - The **internal language**, or that of object code generation.

Sequencing is a Derived Form I

In order to ensure language safety, it is necessary to demonstrate an equivalency between the internal and external semantics of a language.

- In a real language, this would equate to demonstrating equivalency between a program and its compiled object code.

THEOREM [Sequencing is a Derived Form] Define $\lambda^{\mathcal{E}}$ as the **external calculus**. This language will be composed of simply typed λ-Calculus, enriched with the Unit type and term, and with the term $t_1; t_2$, E-Seq, E-SeqNext, and T-Seq.

Define $\lambda^{\mathcal{I}}$ as the **internal calculus**. This language will be composed of the simply typed λ-Calculus and Unit type and term *only*.

Sequencing is a Derived Form II

Define $e \in \lambda^{\mathcal{E}} \to \lambda^{\mathcal{I}}$ as an **elaboration function**, which translates from the external language to the internal language. It does so by replacing all instances of $t_1; t_2$ with $(\lambda x : \mathit{Unit}.t_2)\ t_1$. For each term t of $\lambda^{\mathcal{E}}$, we have:

$$t \xrightarrow{\mathcal{E}} t' \iff e(t) \xrightarrow{\mathcal{I}} e(t') \tag{10}$$

$$\Gamma \vdash^{\mathcal{E}} t : T \iff \Gamma \vdash^{\mathcal{I}} e(t) : T \tag{11}$$

- For equation 10, if a term t evaluates to t' in the external language, the internal representation of t evaluates to the internal representation of t', *and vice versa*.
- For equation 11, if a term has some type T in the external language, it's internal representation will have the same type, *and vice versa*.

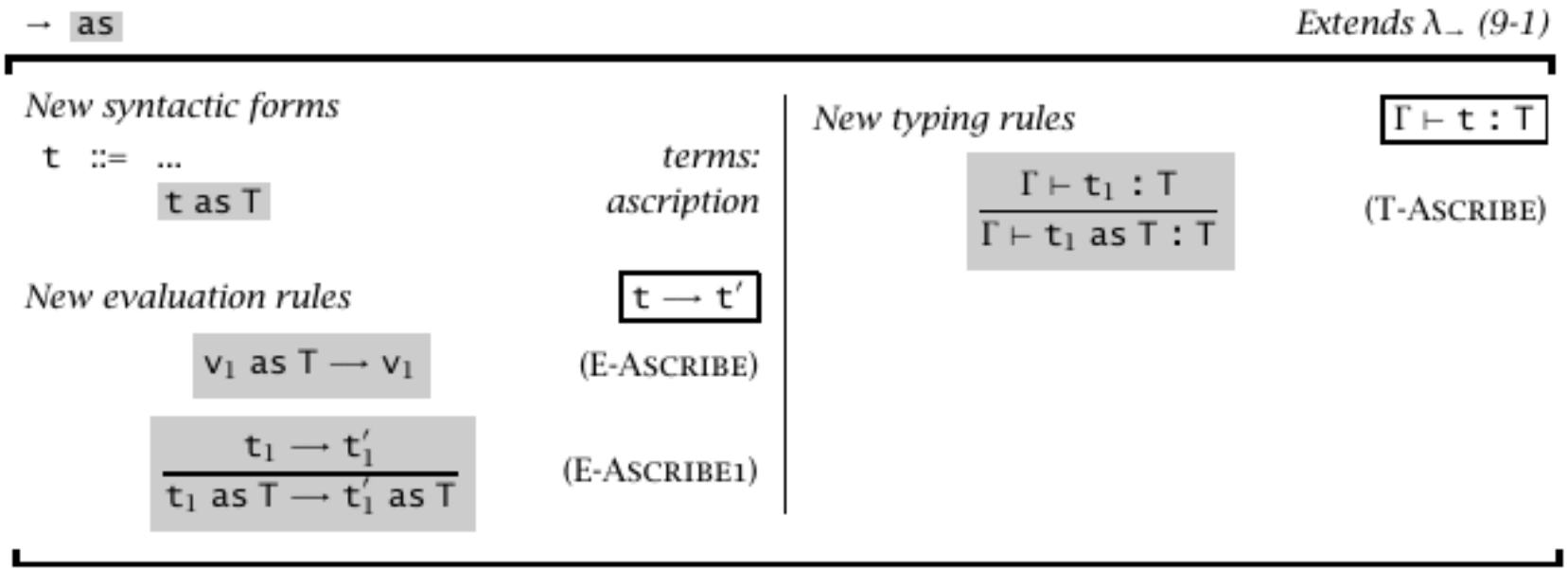
Ascription

So far, we have only given explicit typing information within λ abstractions, but most languages support the ability to **ascribe** types to existing terms.

- This is related to the idea of casting in C\C++, but works more like a type assertion.
- We want the the term $(t \; as \; T)$ to read as "the term t, to which we ascribe the type T."

We will present ascription semantics as a new term of the language, and in tutorial you will see how it can be expressed using a derived form.

Ascription Semantics



The above is an extension of the pure, typed λ-Calculus.

Ascription Description

In the above semantics, we have two evaluation rules and a typing rule:

- The congruence rule E-Ascribe1.
- An elimination rule E-Ascribe.
  - This evaluation works similarly to E-SeqNext. Once t has been fully evaluated, the next step is to throw away the ascription.
- The typing rule T-Ascribe, which types an ascription term as the type being ascribed.
  - Note the antecedent. A type ascription is only well-typed if $t_1 : T$ was *already derivable from* Γ*!*

Wait, What?

If a type ascription t *as* T gives t the type T, how could t not be well typed?

- If ascription behaved as stated above (which is what your humble professor understood when he first read it), wouldn't we add the typing information to Γ and move on?

According to the given semantics, type ascription doesn't give t type T, it **checks that** t **already has** type T.

- Ascriptions are most practical when used with **polymorphic types**. If a term may have one of a set of types, ascription can be used to select one or more types from among the set.
- Otherwise, ascription is commonly used in **documentation**.
  - It can be easy to lose track of typing in long and complicated expressions.
  - By ascribing a type and running the type checker, you can check to make sure a term is the type you think it is.

Let Bindings

When writing complex expressions, it is often useful to give names to subexpressions, so that they may be referred to by name.

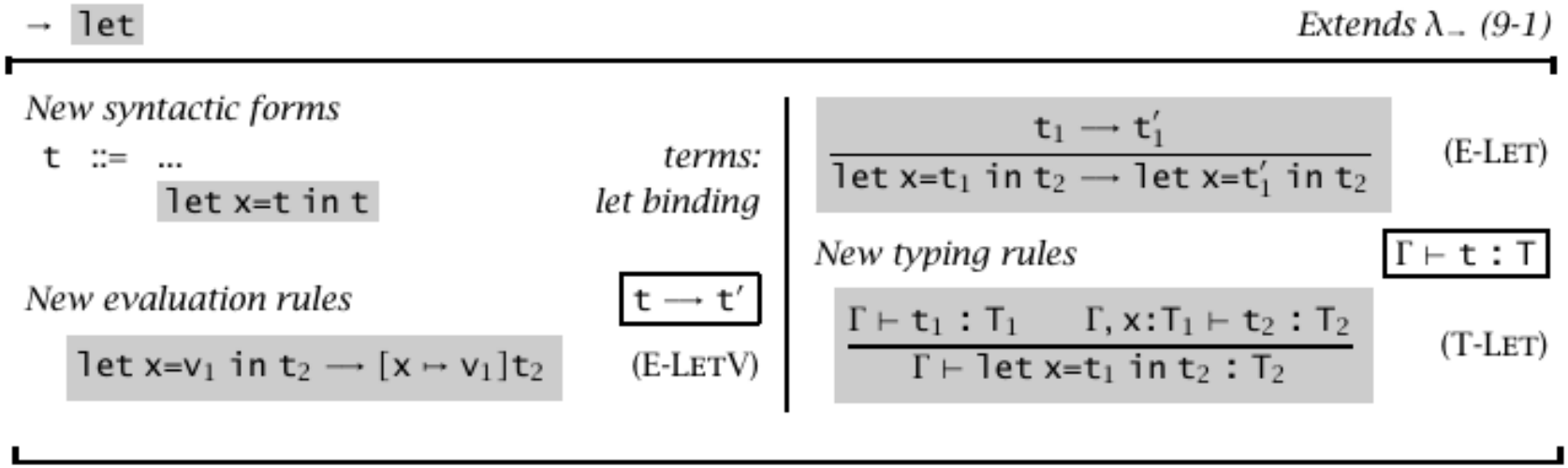
- We've been doing this informally since we introduced λ-Calculus.
- The main way of doing this in Haskell explicitly is `where` bindings, though the process is also implicit in pattern matching.

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

- This is identical to substitution in pure λ-Calculus.
- Unlike our in-class examples, we don't substitute one-at-a-time, but all at once.

Let Binding Semantics

If we were adding let bindings as a new term, we would do it as follows.



The above is an extension of purely typed λ-Calculus.

Let-ting Our Hair Down

In the above semantics, we again have two evaluation rules and a typing rule.

- The elimination rule E-LetV evaluates to a substitution over t_2.
  - Note that this substitution is *semantically identical* to the substitution produced by E-AppAbs.
- The congruence rule E-Let lets us to evaluate t_1 one step.
- Setting execution up this way (only allowing values to be substituted) is a tremendous time-saving feature!
  - If we just performed a straight substitution without fully evaluating t_1, we would multiply the amount of work the computer has to do by the number of times we sub in t_1.

Let-ting the Cat Out of the Bag

- For typing, it is obvious that the type of a let binding should be the type of t_2.
- However, we may require the type of the bound variable in order to determine this type (hence the second antecedent).
- This leaves the question, what is the type of the bound variable, considering we have not defined its type in our let binding syntax?
- Intuitively, x and t_1 should have the same type, given the nature of substitution.
  - We have proven theorems to this effect in the past.
- Hence, our first antecedent. Whatever type is derivable for t_1 will type x for the purposes of finding the typing of t_2.
  - And if t_1 is not well-typed, neither is our let binding!

Let it Go! Let it Go!

Finally, can we express let bindings using a derived form?

Intuitively, we can introduce the called for substitution easily using an abstraction-application pair:

$$\mathit{let}\ x = t_1\ \mathit{in}\ t_2 \stackrel{\mathit{def}}{=} (\lambda x : T_1.t_2)\ t_1 \tag{12}$$

But the right-hand-side contains a symbol that the left-hand-side does not!

Take a Let-ter, Maria!

From whence has this typing information been derived!?

- If we expect this typing information to be present after desugaring (which is a syntactic requirement), we need to know where this term comes from.
- In our typing rules, we derive the type of x by first deriving the type of t_1. We can use the typechecker!
- We therefore have two options:
  - Regard the derived form as a transformation on typing derivations.
  - **Decorate** the terms of our language with the results of typechecking.
- In short, the evaluation semantics of let bindings can be desugared, but the typing behaviour *must* be built into the inner language.

Last Slide Comic

11.6 Pairs

Most programming languages provide a variety of ways of building compound data structures. The simplest of these is *pairs*, or more generally *tuples*, of values. We treat pairs in this section, then do the more general cases of tuples and labeled records in §11.7 and §11.8.[3]

The formalization of pairs is almost too simple to be worth discussing—by this point in the book, it should be about as easy to read the rules in Figure 11-5 as to wade through a description in English conveying the same information. However, let's look briefly at the various parts of the definition to emphasize the common pattern.

Adding pairs to the simply typed lambda-calculus involves adding two new forms of term—pairing, written $\{t_1, t_2\}$, and projection, written `t.1` for the

3. The `fullsimple` implementation does not actually provide the pairing syntax described here, since tuples are more general anyway.