

CS421 Lecture 12: Type Derivations¹

Mark Hills

`mhills@cs.uiuc.edu`

University of Illinois at Urbana-Champaign

June 24, 2008

¹Based on slides by Mattox Beckman, as updated by Vikram Adve, Gul Agha, and Elsa Gunter

Introduction: Why Have Data Types?

Data types play a key role in:

- ▶ the design of programs (through data abstraction);
- ▶ the analysis of programs (through type checking and type inference);
- ▶ the translation and execution of programs (through compile-time code generation).

Underlying these uses are type systems that provide one way to express the meaning of a program.

Objectives

By the end of lecture, you should know

- ▶ the terminology of types and type checking
- ▶ about type rules and how to structure a proof-tree
- ▶ how to use the type rules to check the type of an expression
- ▶ how to use the type rules to infer the type of an expression
- ▶ how to write your own type rule for an expression

Type

A type t defines a set of possible data values

- ▶ E.g., `short` in C $\equiv \{i : -2^{15} \leq i \leq 2^{15} - 1\}$
- ▶ A value in this set is said to be of type t .

Type System

The rules of a language that describe how to assign a type to each expression in a program are the *type system*.

Sound Type Systems

A type system is **sound** if the type $t(e)$ it assigns to expression e is always correct, i.e., whenever e is assigned type t and evaluates to value v , the value v is in the set of values defined for type t

- ▶ SML, OCAML, and Ada have sound type systems.
- ▶ Most implementations of C and C++ do not.

Type Errors

An operation that is applied to an operand of an illegal type is a *type error*. When the operation may be applied to an operand of an illegal type, this is a *potential type error*.

- ▶ `1 +. 2.5` *in OCAML*
- ▶ `atoi();` *in C*
- ▶ `MyCls cref = (MyCls) X;` *in Java*

This last may or may not be a type error. Such an error must be detected at run time.

Strongly Typed Language

A language is **strongly typed** when no application of an operator to its arguments can lead to a run-time type error. This depends strongly on the definitions of “type” and “type system” for the language. A sound type system is required, and expensive checks may be needed.

- ▶ C++ claimed to be “strongly typed”, but
 - ▶ Union types allow creating a value at one type and using it at another
 - ▶ Type coercions may cause unexpected (undesirable) effects
 - ▶ No array bounds check (in fact, no runtime checks at all)
- ▶ SML, OCAML “strongly typed” but still must do dynamic array bounds checks, runtime type case analysis, and other checks

Weakly Typed Languages

A language is **weakly typed** when applications of operators to operands *can* lead to run-time type errors or, worse, undetectable errors that do not cause the program to halt.

- ▶ E.g., Perl, C, C++, Fortran90, assembly languages
- ▶ Advantages: more flexible programming; faster code

Statically Typed Languages

A **static type** is a type assigned to an expression at compile-time.
A **statically typed language** is one where every expression is assigned a static type.

- ▶ **Weakly Statically Typed:** Assigned types are *assumed to be correct* and not checked.
E.g., C, C++, Fortran90
- ▶ **Strongly Statically Typed:** Assigned types are checked *at compile-time or run-time*
E.g., ML, Haskell, Ada, OCAML, Java.

Dynamically Typed Language

A **dynamic type** is a type assigned to a storage location at run time. A **dynamically typed language** is one where the type of an expression may not be determined until run time.

- ▶ Safe, but may incur significant run-time overheads
- ▶ Cost: Cannot generate unique code for some operations at compile-time
 - ⇒ Generally used only in interpreted languages

Examples: Lisp, Scheme, SmallTalk

Type Checking

Type checking is a program analysis that attempts to check whether a program can generate type errors.

- ▶ Type checking assures that operations are applied to the right number of arguments of the right types
- ▶ Right type may mean same type as was specified, or may mean that there is a predefined implicit coercion that will be applied
- ▶ Used to resolve overloaded operations

Type Checking

Type checking may be done *statically* at compile time or *dynamically* at run time.

- ▶ dynamically typed languages, like Lisp and Prolog, do only dynamic type checking
- ▶ statically typed languages can do most type checking statically (at compile time)

Dynamic Type Checking

- ▶ dynamic type checking is performed at run-time before each operation is applied
- ▶ types of variables and operations are left unspecified until run-time
- ▶ variables may be used at different times with different types
- ▶ data objects must maintain type information at runtime
- ▶ type errors are not detected until the violating section of code is executed, sometimes years after being written

Dynamic Types: An Example

```
1 CL-USER(1): (defvar *list1* '(1 2 3))
2 *LIST1*
3 CL-USER(2): (defvar *list2* '("three" "short" "words"))
4 *LIST2*
5 CL-USER(3): (defvar *a* (car *list1*))
6 *A*
7 CL-USER(4): (defvar *b* (car (cdr *list1*)))
8 *B*
9 CL-USER(5): (+ *a* *b*)
10 3
```

Dynamic Types: An Example (part 2)

```
1 CL-USER(6): (setf *a* (car *list2*))  
2 "three"  
3 CL-USER(7): (setf *b* (car (cdr *list2*)))  
4 "short"  
5 CL-USER(8): (concatenate 'string *a* *b*)  
6 "threeshort"  
7 CL-USER(9): (+ *a* *b*)  
8 Error: "three" is not of the expected type 'NUMBER'
```

Static Type Checking

- ▶ performed after parsing, but before code generation – considered part of the compiler *front-end*
- ▶ type of every variable and type signature of every operator must be known at compile time
- ▶ can eliminate need to store type information with data object if no dynamic checks are needed – *type erasure*
- ▶ catches many programming errors early – before execution
- ▶ cannot check types that depend on dynamically computed values, like array bounds

Static Type Checking

Static type checking typically places some restrictions on the language:

- ▶ garbage collection instead of explicit allocation and deallocation
- ▶ references instead of pointers
- ▶ variables initialized automatically when created
- ▶ variables only used at one type (unions allow workarounds, but introduce dynamic checks)

Type Checking Costs

Type checking may incur several possible run-time overheads:

1. Run-time type checks, e.g., `MyCls cref = (MyCls) X;`
2. Run-time array bounds checks, e.g., `int X = A[i];`
3. Run-time null pointer checks, e.g., `int X = *p;`
4. Proper initialization of data values, e.g., zeroing in `new T;`
5. Garbage Collection (GC): disallow explicit `free / delete`

Static type checking may eliminate some instances of the first 3 kinds of checks. #2 and #3 are especially hard to eliminate and every widely-used, strongly typed language relies on GC!

Type Declarations

A *type declaration* is the explicit assignment of a type to a variable or function in the program source

- ▶ must be checked in a strongly typed language
- ▶ often not necessary for strong typing or even static typing

Type Inference

A program analysis that attempts to identify the type of an expression from the program context of the expression; also called *type reconstruction*

- ▶ fully static type inference first introduced by Robin Milner in ML
- ▶ Haskell, OCaml, and SML all use type inference
- ▶ As we've noticed, records are a problem for type inference

Type Equivalence and Compatibility

Two types that represent the same underlying data items are said to be *equivalent*. Equivalence comes in two forms:

- ▶ Name equivalence, where two types are equivalent if they share the same name
- ▶ Structural equivalence, where two types are equivalent if they have the same structure

Sometimes equivalence is not necessary. In these cases, operations may just need a *compatible* type.

Type Casts and Coercions

Data can often be converted from one type to another.

- ▶ *Casting* occurs when data of one type is explicitly converted to another type
- ▶ *Nonconverting type casts* reinterpret the underlying bits as a different data type, but do not perform any conversion on the data
- ▶ *Coercions* occur behind the scenes when data of one type is changed to another to make it compatible with an operation on the data (i.e. converting int to double in C floating-point addition)

Format of a Type Judgment

An *type judgment* has the following form:

$$\Gamma \vdash e : \tau$$

where

- ▶ Γ is a *type environment*, which can be seen as a list or set of pairs $x : \tau$ with x a variable or function name and τ the type;
- ▶ e is a program expression;
- ▶ and τ is the *type* to be assigned to e .

Note: the \vdash is pronounced “turnstile”, “entails”, or sometimes “satisfies”.

Examples of Valid Type Judgments

- ▶ $\vdash \text{true} \ \&\& \ \text{false} : \text{bool}$
- ▶ $[x : \text{int}] \vdash x + 3 : \text{int}$
- ▶ $[f : \text{int} \rightarrow \text{string}] \vdash f(5) : \text{string}$

Format of Typing Rules

$$\begin{array}{c}
 \text{Assumptions} \cdots \cdots \cdots \downarrow \\
 \frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \\
 \text{Conclusion} \cdots \cdots \cdots \rightarrow
 \end{array}$$

- ▶ **Core Idea:** Type of expression determined by type of components
- ▶ If a rule has no assumptions, then it is called an *axiom*
- ▶ Γ may be left out if we don't need a type environment
- ▶ Γ , e , and τ are *parameterized* – they may contain *meta-variables* used for typing

Axioms – Constants

$$\frac{}{\vdash n : \text{int}} \text{(assuming } n \text{ is an int)}$$
$$\frac{}{\vdash \text{true} : \text{bool}}$$
$$\frac{}{\vdash \text{false} : \text{bool}}$$

- ▶ These are rules that are true no matter what the typing context
- ▶ n is a *meta-variable* representing any int

Axioms – Variables

There are many ways you may see this rule formatted. If Γ is a set, it may be shown as:

$$\frac{}{\Gamma \vdash x : \tau} \text{ if } (x : \tau) \in \Gamma$$

It may also be shown as:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

If Γ is treated as a list, order is important, and we will say $\Gamma(x) = \tau$ if $x : \tau \in \Gamma$ and no other definition of x is to the left of $x : \tau$. Then, we will have:

$$\frac{}{\Gamma \vdash x : \tau} \text{ if } \Gamma(x) = \tau$$

We will use the first.

Simple Rules

- ▶ Rules for arithmetic operators (+,*,etc):

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \oplus e_2 : \text{int}}$$

- ▶ Rules for relational operators (=,<,etc) are similar:

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \sim e_2 : \text{bool}}$$

Simple Rules

Booleans look similar, except for not, which has only one operand:

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \ \&\& \ e_2 : \text{bool}}$$

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \ || \ e_2 : \text{bool}}$$

$$\frac{\Gamma \vdash e_1 : \text{bool}}{\Gamma \vdash ! \ e_1 : \text{bool}}$$

Simple Example

Suppose we want to prove that $\Gamma \vdash y \mid\mid (x + 3 > 6) : \text{bool}$.
 Assume that $\Gamma = [x : \text{int} ; y : \text{bool}]$

First thing: Write down the thing you are trying to prove, and put a bar over it. Proofs are from the bottom up.

$$\frac{?}{\Gamma \vdash y \mid\mid (x + 3 > 6) : \text{bool}}$$

Simple Example

Suppose we want to prove that $\Gamma \vdash y \mid\mid (x + 3 > 6) : \text{bool}$.
 Assume that $\Gamma = [x : \text{int} ; y : \text{bool}]$

Next, look at the *outermost* expression. What kind of expression is this? What rule has this as its conclusion?

$$\frac{?}{\Gamma \vdash y \mid\mid (x + 3 > 6) : \text{bool}}$$

Simple Example

Suppose we want to prove that $\Gamma \vdash y \parallel (x + 3 > 6) : \text{bool}$.
 Assume that $\Gamma = [x : \text{int} ; y : \text{bool}]$

Next, look at the *outermost* expression. What kind of expression is this? What rule has this as its conclusion?

?

$$\frac{}{\Gamma \vdash y \parallel (x + 3 > 6) : \text{bool}}$$

Boolean or:
$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \parallel e_2 : \text{bool}}$$

This will tell us what we need to do next.

Simple Example

Suppose we want to prove that $\Gamma \vdash y \parallel (x + 3 > 6) : \text{bool}$.
 Assume that $\Gamma = [x : \text{int} ; y : \text{bool}]$

Write parts on top and put a bar over them as well.

$$\frac{\frac{?}{\Gamma \vdash y : \text{bool}} \quad \frac{?}{\Gamma \vdash x + 3 > 6 : \text{bool}}}{\Gamma \vdash y \parallel (x + 3 > 6) : \text{bool}}$$

Simple Example

Suppose we want to prove that $\Gamma \vdash y \parallel (x + 3 > 6) : \text{bool}$.
 Assume that $\Gamma = [x : \text{int} ; y : \text{bool}]$

Now, pick an assumption. We can work left to right. Which rule has $\Gamma \vdash y : \text{bool}$ as a conclusion?

$$\frac{\frac{?}{\Gamma \vdash y : \text{bool}} \quad \frac{?}{\Gamma \vdash x + 3 > 6 : \text{bool}}}{\Gamma \vdash y \parallel (x + 3 > 6) : \text{bool}}$$

Simple Example

Suppose we want to prove that $\Gamma \vdash y \parallel (x + 3 > 6) : \text{bool}$.
 Assume that $\Gamma = [x : \text{int} ; y : \text{bool}]$

This is the variable axiom. Now, what rule has
 $\Gamma \vdash x + 3 > 6 : \text{bool}$ as a conclusion?

$$\frac{\frac{}{\Gamma \vdash y : \text{bool}} \quad \frac{}{\Gamma \vdash x + 3 > 6 : \text{bool}}}{\Gamma \vdash y \parallel (x + 3 > 6) : \text{bool}} \quad ?$$

Simple Example

Suppose we want to prove that $\Gamma \vdash y \parallel (x + 3 > 6) : \text{bool}$.
 Assume that $\Gamma = [x : \text{int} ; y : \text{bool}]$

This is the rule for the $>$ relational operator. Again, we expand this out.

$$\frac{\frac{\Gamma \vdash y : \text{bool}}{\Gamma \vdash y : \text{bool}} \quad \frac{\frac{\frac{?}{\Gamma \vdash x + 3 : \text{int}}{\Gamma \vdash x + 3 : \text{int}} \quad \frac{\frac{?}{\Gamma \vdash 6 : \text{int}}{\Gamma \vdash 6 : \text{int}}}{\Gamma \vdash x + 3 > 6 : \text{bool}}}{\Gamma \vdash x + 3 > 6 : \text{bool}}}{\Gamma \vdash y \parallel (x + 3 > 6) : \text{bool}}$$

Simple Example

Suppose we want to prove that $\Gamma \vdash y \parallel (x + 3 > 6) : \text{bool}$.
 Assume that $\Gamma = [x : \text{int} ; y : \text{bool}]$

Now, we need to select an assumption again. Which rule has $\Gamma \vdash 6 : \text{int}$ as its conclusion?

$$\frac{\Gamma \vdash y : \text{bool} \quad \frac{\frac{?}{\Gamma \vdash x + 3 : \text{int}} \quad \frac{?}{\Gamma \vdash 6 : \text{int}}}{\Gamma \vdash x + 3 > 6 : \text{bool}}}{\Gamma \vdash y \parallel (x + 3 > 6) : \text{bool}}$$

Simple Example

Suppose we want to prove that $\Gamma \vdash y \mid\mid (x + 3 > 6) : \text{bool}$.
 Assume that $\Gamma = [x : \text{int} ; y : \text{bool}]$

This is the axiom for constants. Now, we can select the other assumption. Which rule has $\Gamma \vdash x + 3 : \text{int}$ as a conclusion?

$$\begin{array}{c}
 \text{?} \\
 \frac{\Gamma \vdash x + 3 : \text{int} \quad \Gamma \vdash 6 : \text{int}}{\Gamma \vdash x + 3 > 6 : \text{bool}} \\
 \frac{\Gamma \vdash y : \text{bool} \quad \Gamma \vdash x + 3 > 6 : \text{bool}}{\Gamma \vdash y \mid\mid (x + 3 > 6) : \text{bool}}
 \end{array}$$

Simple Example

Suppose we want to prove that $\Gamma \vdash y \parallel (x + 3 > 6) : \text{bool}$.
 Assume that $\Gamma = [x : \text{int} ; y : \text{bool}]$

This is the rule for arithmetic operations.

$$\frac{\frac{\frac{?}{\Gamma \vdash x : \text{int}} \quad \frac{?}{\Gamma \vdash 3 : \text{int}}}{\Gamma \vdash x + 3 : \text{int}} \quad \frac{}{\Gamma \vdash 6 : \text{int}}}{\Gamma \vdash x + 3 > 6 : \text{bool}} \quad \frac{}{\Gamma \vdash y : \text{bool}}}{\Gamma \vdash y \parallel (x + 3 > 6) : \text{bool}}$$

Simple Example

Suppose we want to prove that $\Gamma \vdash y \mid\mid (x + 3 > 6) : \text{bool}$.
 Assume that $\Gamma = [x : \text{int} ; y : \text{bool}]$

Now, we pick an assumption to prove again. Which rule has $\Gamma \vdash 3 : \text{int}$ as a conclusion?

$$\begin{array}{c}
 \begin{array}{c}
 \frac{\quad}{\Gamma \vdash x : \text{int}} \quad ? \quad \frac{\quad}{\Gamma \vdash 3 : \text{int}} \quad ? \\
 \hline
 \Gamma \vdash x + 3 : \text{int} \qquad \Gamma \vdash 6 : \text{int} \\
 \hline
 \Gamma \vdash x + 3 > 6 : \text{bool} \\
 \hline
 \Gamma \vdash y : \text{bool} \qquad \Gamma \vdash x + 3 > 6 : \text{bool} \\
 \hline
 \Gamma \vdash y \mid\mid (x + 3 > 6) : \text{bool}
 \end{array}
 \end{array}$$

Simple Example

Suppose we want to prove that $\Gamma \vdash y \parallel (x + 3 > 6) : \text{bool}$.
 Assume that $\Gamma = [x : \text{int} ; y : \text{bool}]$

This is the axiom for constants. Which rule has $\Gamma \vdash x : \text{int}$ as a conclusion?

$$\begin{array}{c}
 \text{?} \\
 \hline
 \Gamma \vdash x : \text{int} \qquad \Gamma \vdash 3 : \text{int} \\
 \hline
 \Gamma \vdash x + 3 : \text{int} \qquad \Gamma \vdash 6 : \text{int} \\
 \hline
 \Gamma \vdash y : \text{bool} \qquad \Gamma \vdash x + 3 > 6 : \text{bool} \\
 \hline
 \Gamma \vdash y \parallel (x + 3 > 6) : \text{bool}
 \end{array}$$

Simple Example

Suppose we want to prove that $\Gamma \vdash y \mid\mid (x + 3 > 6) : \text{bool}$.
 Assume that $\Gamma = [x : \text{int} ; y : \text{bool}]$

This is the rule for variables. Now, we have no more assumptions,
 and our proof is **DONE!**

$$\frac{\frac{\frac{\Gamma \vdash x : \text{int}}{\Gamma \vdash x + 3 : \text{int}} \quad \frac{\Gamma \vdash 3 : \text{int}}{\Gamma \vdash 6 : \text{int}}}{\Gamma \vdash x + 3 > 6 : \text{bool}} \quad \Gamma \vdash y : \text{bool}}{\Gamma \vdash y \mid\mid (x + 3 > 6) : \text{bool}}$$

Type Variables in Rules

$$\text{If } \frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau}$$

- ▶ τ is a *type variable*, or *meta-variable*
- ▶ It means “any type at all”—but whatever type τ you pick it has to be the same for the three places it shows up in this rule.
- ▶ So... the `if` rule says that `if` can result in any type, as long as the `then` and `else` branches have the same type. This could even include functions.

Function Application

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2}$$

- ▶ If you have a function expression e_1 of type $\tau_1 \rightarrow \tau_2$, applied to an argument expression e_2 of type τ_1 , the resulting expression has type τ_2

Function Application Example

$$\frac{\Gamma \vdash \text{print_int} : \text{int} \rightarrow \text{unit} \quad \Gamma \vdash 5 : \text{int}}{\Gamma \vdash \text{print_int } 5 : \text{unit}}$$

- ▶ $e_1 = \text{print_int}$
- ▶ $e_2 = 5$
- ▶ $\tau_1 = \text{int}$
- ▶ $\tau_2 = \text{unit}$.

Function Application Example 2

$$\Gamma \vdash \text{map print_int} : \text{int list} \rightarrow \text{unit list} \quad \Gamma \vdash [3; 7] : \text{int list}$$

$$\Gamma \vdash \text{map print_int } [3; 7] : \text{unit list}$$

- ▶ $e_1 = \text{map print_int}$
- ▶ $e_2 = [3; 7]$
- ▶ $\tau_1 = \text{int list}$
- ▶ $\tau_2 = \text{unit list}$

Functions

- ▶ Important point: the rules describe types, but they also describe when you may change Γ .
- ▶ You may **NOT** change Γ except as described!

The rule for functions adds the bound variable to the type environment:

$$\frac{\Gamma \cup [x : \tau_1] \vdash e : \tau_2}{\Gamma \vdash \text{fun } x \rightarrow e : \tau_1 \rightarrow \tau_2}$$

Function Examples

$$\frac{\Gamma \cup [y : \text{int}] \vdash y + 3 : \text{int}}{\Gamma \vdash \text{fun } y \rightarrow y + 3 : \text{int} \rightarrow \text{int}}$$

$$\frac{\Gamma \cup [f : \text{int} \rightarrow \text{bool}] \vdash (f \ 2) :: [\text{true}] : \text{bool list}}{\Gamma \vdash (\text{fun } f \rightarrow (f \ 2) :: [\text{true}]) : (\text{int} \rightarrow \text{bool}) \rightarrow \text{bool list}}$$

Let and Letrec

- ▶ Let Rule:

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \cup [x : \tau] \vdash e_2 : \tau'}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau'}$$

- ▶ Letrec Rule:

$$\frac{\Gamma \cup [x : \tau] \vdash e_1 : \tau \quad \Gamma \cup [x : \tau] \vdash e_2 : \tau'}{\Gamma \vdash \text{let rec } x = e_1 \text{ in } e_2 : \tau'}$$

Some caveats...

The above system cannot handle polymorphism like we find in OCaml. Specifically, we have meta-variables in our proof logic, but no type variables in the type language. To handle this properly, we would need:

- ▶ Object level type variables
- ▶ Type quantification (universal types)
- ▶ **let** and **letrec** rules to introduce polymorphism
- ▶ Explicit rules to eliminate (instantiate) polymorphism

A more advanced example

```
 $\Gamma \vdash$  (let rec one = 1 :: one in  
  let x = 2 in  
    fun y  $\rightarrow$  (x :: y :: one)) : int  $\rightarrow$  int list
```

$\Gamma \vdash (\text{let rec one} = 1 :: \text{one in let } x = 2 \text{ in fun } y \rightarrow (x :: y :: \text{one})) : \text{int} \rightarrow \text{int list}$

What rule has this as the conclusion?

?

$\Gamma \vdash (\text{let rec one} = 1 :: \text{one in let } x = 2 \text{ in fun } y \rightarrow (x :: y :: \text{one})) : \text{int} \rightarrow \text{int list}$

$$\Gamma \vdash (\text{let rec one} = 1 :: \text{one in let } x = 2 \text{ in fun } y \rightarrow (x :: y :: \text{one})) : \text{int} \rightarrow \text{int list}$$

This uses the letrec rule.

#1 #2

$$\Gamma \vdash (\text{let rec one} = 1 :: \text{one in let } x = 2 \text{ in fun } y \rightarrow (x :: y :: \text{one})) : \text{int} \rightarrow \text{int list}$$

$$\#1 = \Gamma \cup [\text{one} : \text{int list}] \vdash (1 :: \text{one}) : \text{int list}$$

$$\#2 = \Gamma \cup [\text{one} : \text{int list}] \vdash \text{let } x = 2 \text{ in fun } y \rightarrow (x :: y :: \text{one}) : \text{int} \rightarrow \text{int list}$$

$\Gamma \vdash (\text{let rec one} = 1 :: \text{one in let } x = 2 \text{ in fun } y \rightarrow (x :: y :: \text{one})) : \text{int} \rightarrow \text{int list}$

Proof for #1: Which rule?

?

$\Gamma \cup [\text{one} : \text{int list}] \vdash (1 :: \text{one}) : \text{int list}$

$\Gamma \vdash (\text{let rec one} = 1 :: \text{one in let } x = 2 \text{ in fun } y \rightarrow (x :: y :: \text{one})) : \text{int} \rightarrow \text{int list}$

Proof for #1: This uses the application rule.

#3 #4

$\Gamma \cup [\text{one} : \text{int list}] \vdash (1 :: \text{one}) : \text{int list}$

#3 = $\Gamma \cup [\text{one} : \text{int list}] \vdash ((::)1) : \text{int list} \rightarrow \text{int list}$

#4 = $\Gamma \cup [\text{one} : \text{int list}] \vdash \text{one} : \text{int list}$

$\Gamma \vdash (\text{let rec one} = 1 :: \text{one in let } x = 2 \text{ in fun } y \rightarrow (x :: y :: \text{one})) : \text{int} \rightarrow \text{int list}$

Proof for #3: What rule applies here?

?

$\Gamma \cup [\text{one} : \text{int list}] \vdash ((::)1) : \text{int list} \rightarrow \text{int list}$

$\Gamma \vdash (\text{let rec one} = 1 :: \text{one in let } x = 2 \text{ in fun } y \rightarrow (x :: y :: \text{one})) : \text{int} \rightarrow \text{int list}$

Proof for #3: This uses application again.

$$\frac{\frac{\frac{?}{(\::) : \text{int} \rightarrow \text{int list} \rightarrow \text{int list}}{\quad}}{\quad} \quad \frac{?}{1 : \text{int}}}{\Gamma \cup [\text{one} : \text{int list}] \vdash ((\::)1) : \text{int list} \rightarrow \text{int list}}$$

$$\Gamma \vdash (\text{let rec one} = 1 :: \text{one in let } x = 2 \text{ in fun } y \rightarrow (x :: y :: \text{one})) : \text{int} \rightarrow \text{int list}$$

Proof for #3: Since $(::)$ is predefined, we can treat this as a constant. 1 is also a constant. Thus, this part is done.

$$\frac{\frac{}{(::) : \text{int} \rightarrow \text{int list} \rightarrow \text{int list}} \quad \frac{}{1 : \text{int}}}{\Gamma \cup [\text{one} : \text{int list}] \vdash ((::)1) : \text{int list} \rightarrow \text{int list}}$$

$\Gamma \vdash (\text{let rec one} = 1 :: \text{one in let } x = 2 \text{ in fun } y \rightarrow (x :: y :: \text{one})) : \text{int} \rightarrow \text{int list}$

Proof for #4: What rule applies here?

?

$\Gamma \cup [\text{one} : \text{int list}] \vdash \text{one} : \text{int list}$

$\Gamma \vdash (\text{let rec one} = 1 :: \text{one in let } x = 2 \text{ in fun } y \rightarrow (x :: y :: \text{one})) : \text{int} \rightarrow \text{int list}$

Proof for #4: This is the variable rule.

$\Gamma \cup [\text{one} : \text{int list}] \vdash \text{one} : \text{int list}$

$\Gamma \vdash (\text{let rec one} = 1 :: \text{one in let } x = 2 \text{ in fun } y \rightarrow (x :: y :: \text{one})) : \text{int} \rightarrow \text{int list}$

Proof for #2: What rule applies here?

?

$\Gamma \cup [\text{one} : \text{int list}] \vdash \text{let } x = 2 \text{ in fun } y \rightarrow (x :: y :: \text{one}) : \text{int} \rightarrow \text{int list}$

$$\Gamma \vdash (\text{let rec one} = 1 :: \text{one in let } x = 2 \text{ in fun } y \rightarrow (x :: y :: \text{one})) : \text{int} \rightarrow \text{int list}$$

Proof for #2: We will use the let rule.

$$\Gamma \cup [\text{one} : \text{int list}] \vdash 2 : \text{int} \quad \#5$$

$$\Gamma \cup [\text{one} : \text{int list}] \vdash \text{let } x = 2 \text{ in fun } y \rightarrow (x :: y :: \text{one}) : \text{int} \rightarrow \text{int list}$$

$$\#5 = \Gamma \cup [\text{one} : \text{int list}, x : \text{int}] \vdash \text{fun } y \rightarrow (x :: y :: \text{one}) : \text{int} \rightarrow \text{int list}$$

$\Gamma \vdash (\text{let rec one} = 1 :: \text{one in let } x = 2 \text{ in fun } y \rightarrow (x :: y :: \text{one})) : \text{int} \rightarrow \text{int list}$

Proof for #5: What rule should we use?

?

$\Gamma \cup [\text{one} : \text{int list}, x : \text{int}] \vdash \text{fun } y \rightarrow (x :: y :: \text{one}) : \text{int} \rightarrow \text{int list}$

$$\Gamma \vdash (\text{let rec one} = 1 :: \text{one in let } x = 2 \text{ in fun } y \rightarrow (x :: y :: \text{one})) : \text{int} \rightarrow \text{int list}$$

Proof for #5: Function rule

?

$$\Gamma \cup [\text{one} : \text{int list}, x : \text{int}, y : \text{int}] \vdash (x :: y :: \text{one}) : \text{int list}$$

$$\Gamma \cup [\text{one} : \text{int list}, x : \text{int}] \vdash \text{fun } y \rightarrow (x :: y :: \text{one}) : \text{int} \rightarrow \text{int list}$$

$$\Gamma \vdash (\text{let rec one} = 1 :: \text{one in let } x = 2 \text{ in fun } y \rightarrow (x :: y :: \text{one})) : \text{int} \rightarrow \text{int list}$$

Proof for #5: Now what? Two more applications...

#6 #7

$$\Gamma \cup [\text{one} : \text{int list}, x : \text{int}, y : \text{int}] \vdash (x :: y :: \text{one}) : \text{int list}$$

$$\Gamma \cup [\text{one} : \text{int list}, x : \text{int}] \vdash \text{fun } y \rightarrow (x :: y :: \text{one}) : \text{int} \rightarrow \text{int list}$$

#6 =

$$\Gamma \cup [\text{one} : \text{int list}, x : \text{int}, y : \text{int}] \vdash ((::)x) : \text{int list} \rightarrow \text{int list}$$

$$\#7 = \Gamma \cup [\text{one} : \text{int list}, x : \text{int}, y : \text{int}] \vdash (y :: \text{one}) : \text{int list}$$

$\Gamma \vdash (\text{let rec one} = 1 :: \text{one in let } x = 2 \text{ in fun } y \rightarrow (x :: y :: \text{one})) : \text{int} \rightarrow \text{int list}$

Proof for #6:

?

$\Gamma \cup [\text{one} : \text{int list}, x : \text{int}, y : \text{int}] \vdash ((::)x) : \text{int list} \rightarrow \text{int list}$

$\Gamma \vdash (\text{let rec one} = 1 :: \text{one in let } x = 2 \text{ in fun } y \rightarrow (x :: y :: \text{one})) : \text{int} \rightarrow \text{int list}$

Proof for #7:

?

$\Gamma \cup [\text{one} : \text{int list}, x : \text{int}, y : \text{int}] \vdash (y :: \text{one}) : \text{int list}$

Types as Specifications

Type systems are a major area of research, and can be very involved, moreso than we have shown here. Types can be seen as *specifications* of desired program behavior.

- ▶ Types describe properties
- ▶ Different type systems describe different properties, eg
 - ▶ Data is read-write versus read-only
 - ▶ Operation has authority to access data
 - ▶ Data came from “right” source
 - ▶ Operation might or could not raise an exception
- ▶ Common type systems focus on types describing data layout and access methods