Some instance messages and methods

```
x || ^x
y || ^y
moveDx: dx Dy: dy ||
   x <- x+dx
   y <- y+dy
```

Executing the following code

```
p moveDX:2 Y:1
```

the value of the expressions `p x` and `p y` is the object `5`.

Smalltalk

Inheritance

class name	`ColoredPoint`
super class	`Point`
class var	
instance var	`color`
class messages and methods	
`newX:xv Y:yv C:cv`	<. . . code . . .>
instance messages and methods	
`color`	`||^color`
`draw`	<. . . code . . .>

Definition of `ColoredPoint` class

- `ColoredPoint` inherits instance variables `x` and `y`, methods `x`, `y`, `moveDX:Dy:`, *etc.*
- `ColoredPoint` adds an instance variable `color` and a method `color` to return the `color` of a `ColoredPoint`.
- The `ColoredPoint draw` method *redefines* (or *overrides*) the one inherited from `Point`.
- An option available in Smalltalk is to specify that a superclass method should be undefined on a subclass.

Example: Consider

```
newX:xv Y:yv C:cv ||
   ^ self new x:xv y:yv color:cv
cp <- ColoredPoint newX:1 Y:2 C:red
cp moveDx:3 Dy:4
```

The value of `cp x` is the object `4`, and the value of the expression `cp color` is the object `red`.

Note that even though `moveDx:Dy:` is an inherited method, defined originally for points without color, the result of moving a `ColoredPoint` is again a `ColoredPoint`.

Smalltalk

Abstraction

Smalltalk rules:

- *Methods are public*.

  Any code with a pointer to an object may send any message to that object. If the corresponding method is defined in the class of the object, or any superclass, the method will be invoked. This makes all methods of an object visible to any code that can access the object.

- *Instance variables are protected*.

  The instance variables of an object are accessible only to methods of the class of the object and to methods of its subclasses.

Smalltalk

Dynamic lookup

The run-time structures used for Smalltalk classes and objects support *dynamic lookup* in two ways.

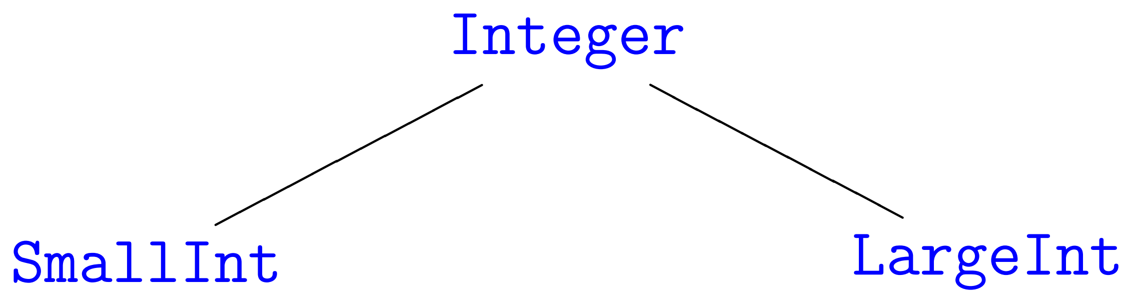
1. Methods are selected through the receiver object.

2. Method lookup starts with the method dictionary of the class of the receiver and then proceeds upwards through the class hierarchy.

Example: A factorial method

```
factorial ||
   self <= 1
      ifTrue: [^1]
      ifFalse: [^ (self-1) factorial * self]
```

in the `Integer` class for



Smalltalk

Interfaces as object types

Although Smalltalk does not use any static type checking, there is an implicit form of type that every Smalltalk programmer uses in some way. Type = interface

The *type* of an object in Smalltalk is its *interface*, *i.e.* the set of messages that can be sent to the object without receiving the error “message not understood”.

The interface of an object is determined by its class, as a class lists the messages that each object will answer. However, different classes may implement the same messages, as there are no Smalltalk rules to keep different classes from using the same selector names.

$A <: B$ iff $\text{Interface}(A) \supseteq \text{Interface}(B)$

$$\frac{a : A \quad A <: B}{a : B}$$

Smalltalk

Subtyping

> Type `A` is a *subtype* of type `B` if any context expecting an expression of type `B` may take any expression of type `A` without introducing a type error.

Semantically, in Smalltalk, it makes sense to associate *subtyping* with the *superset* relation on class interfaces.

$\boxed{?}$ Why?

- In Smalltalk, the interface of a subclass is often a subtype of the interface of its superclass. The reason being that a subclass will ordinarily inherit all of the methods of its superclass, possibly adding more methods.
- In general, however, subclassing does not always lead to subtyping in Smalltalk.
  1. Because it is possible to delete a method from a superclass in a subclass, a subclass may not produce a subtype.
  2. On the other hand, it is easy to have subtyping without inheritance.

∼ Topic VI ∼

Types in programming languages

References:

- **Chapter 6** of *Concepts in programming languages* by J. C. Mitchell. CUP, 2003.
- **Sections 4.9 and 8.6** of *Programming languages: Concepts & constructs* by R. Sethi (2ND EDITION). Addison-Wesley, 1996.

Types in programming

- A *type* is a collection of computational entities that share some common property.
- There are three main uses of types in programming languages:
  1. naming and organizing concepts,
  2. making sure that bit sequences in computer memory are interpreted consistently,
  3. providing information to the compiler about data manipulated by the program.

- Using types to organise a program makes it easier for someone to read, understand, and maintain the program. Types can serve an important purpose in documenting the design and intent of the program.
- Type information in programs can be used for many kinds of optimisations.

Type systems

A *type system* for a language is a set of rules for associating a type with phrases in the language.

Terms strong and weak refer to the effectiveness with which a type system prevents errors. A type system is *strong* if it accepts only *safe* phrases. In other words, phrases that are accepted by a strong type system are guaranteed to evaluate without type error. A type system is *weak* if it is not strong.

Type safety

A programming language is *type safe* if no program is allowed to violate its type distinctions.

Safety	**Example language**	**Explanation**
Not safe	C, C++	Type casts, pointer arithmetic
Almost safe	Pascal	Explicit deallocation; dangling pointers
Safe	LISP, SML, Smalltalk, Java	Type checking

Type checking

A *type error* occurs when a computational entity is used in a manner that is inconsistent with the concept it represents.

Type checking is used to prevent some or all type errors, ensuring that the operations in a program are applied properly.

Some questions to be asked about type checking in a language:

- Is the type system *strong* or *weak*?
- Is the checking done *statically* or *dynamically*?
- How *expressive* is the type system; that is, amongst safe programs, how many does it accept?

Static and dynamic type checking

Run-time type checking: The compiler generates code so that, when an operation is performed, the code checks to make sure that the operands have the correct types.

Examples: LISP, Smalltalk.

Compile-time type checking: The compiler checks the program text for potential type errors.

Example: SML.

NB: Most programming languages use some combination of compile-time and run-time type checking.

Static vs. dynamic type checking

Main trade-offs between compile-time and run-time checking:

Form of type checking	**Advantages**	**Disadvantages**
Run-time	Prevents type errors	Slows program execution
Compile-time	Prevents type errors Eliminates run-time tests Finds type errors before execution and run-time tests	May restrict programming because tests are *conservative*

Type checking in ML

Idea

Given a context Γ, an expression e, and a type τ, decide whether or not the expression e is of type τ in context Γ.

Type checking in ML

Idea

Given a context Γ, an expression e, and a type τ, decide whether or not the expression e is of type τ in context Γ.

Examples:

♦

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1 \; \texttt{orelse} \; e_2 : \texttt{bool}}$$

Type checking in ML

Idea

Given a context Γ, an expression e, and a type τ, decide whether or not the expression e is of type τ in context Γ.

Examples:

- $$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1 \ \texttt{orelse}\ e_2 : \texttt{bool}}$$

$$\mathsf{TC}(\Gamma, e_1 \ \texttt{orelse}\ e_2, \tau) = \begin{cases} \mathsf{TC}(\Gamma, e_1, \texttt{bool}) \wedge \mathsf{TC}(\Gamma, e_2, \texttt{bool}) & \text{, if } \tau = \texttt{bool} \\ \text{false} & \text{, otherwise} \end{cases}$$

◆

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2}$$

♦

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2}$$

$$\mathsf{TC}(\Gamma, (e_1, e_2), \tau) = \begin{cases} \mathsf{TC}(\Gamma, e_1, \tau_1) \wedge \mathsf{TC}(\Gamma, e_2, \tau_2) & \text{, if } \tau = \tau_1 * \tau_2 \\ \text{false} & \text{, otherwise} \end{cases}$$

Type equality

The question of *type equality* arises during type checking.

$\boxed{\textbf{?}}$ What does it mean for two types to be equal!?

Type equality

The question of *type equality* arises during type checking.

$\boxed{?}$ What does it mean for two types to be equal!?

Structural equality. Two type expressions are *structurally equal* if and only if they are equivalent under the following three rules.

SE1. A type name is structurally equal to itself.

SE2. Two types are structurally equal if they are formed by applying the same type constructor to structurally equal types.

SE3. After a type declaration, say `type n = T`, the type name `n` is structurally equal to `T`.

Name equality:

Pure name equality. A type name is equal to itself, but no constructed type is equal to any other constructed type.

Transitive name equality. A type name is equal to itself and can be declared equal to other type names.

Type-expression equality. A type name is equal only to itself. Two type expressions are equal if they are formed by applying the same constructor to equal expressions. In other words, the expressions have to be identical.

Examples:

- **Type equality in Pascal/Modula-2.** Type equality was left ambiguous in Pascal. Its successor, Modula-2, avoided ambiguity by defining two types to be *compatible* if
  1. they are the same name, or
  2. they are `s` and `t`, and `s = t` is a type declaration, or
  3. one is a subrange of the other, or
  4. both are subranges of the same basic type.

Type declarations

There are two basic forms of type declarations:

Transparent. An alternative name is given to a type that can also be expressed without this name.

Opaque. A new type is introduced into the program that is not equal to any other type.

Examples $1+1 : \text{int}$
$1.0 + 1.0 : \text{real}$ + is overloaded

Type inference

$\text{fn } x \Rightarrow x :$
$\text{bool} \to \text{bool}$
$\text{int} \to \text{int}$
$\vdots$

- *Type inference* is the process of determining the types of phrases based on the constructs that appear in them.
- An important language innovation.
- A cool algorithm.
- Gives some idea of how other static analysis algorithms work.

$\alpha \to \alpha$
most general type
[polymorphism]

Type inference in ML

Idea

Typing rule:

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

Type inference in ML

Idea

Typing rule:

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

Inference rule:

$$\frac{}{\Gamma \vdash x : \gamma}$$

Type inference in ML

Idea

Typing rule:

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

Inference rule:

Type constraint

$$\frac{}{\Gamma \vdash x : \gamma} \quad \boxed{\gamma \approx \alpha} \text{ if } x : \alpha \text{ in } \Gamma$$

Typing rule:

$$\frac{\Gamma \vdash f : \sigma \;{-}{>}\; \tau \qquad \Gamma \vdash e : \sigma}{\Gamma \vdash f(e) : \tau}$$

Typing rule:

$$\frac{\Gamma \vdash f : \sigma \ \texttt{->} \ \tau \qquad \Gamma \vdash e : \sigma}{\Gamma \vdash f(e) : \tau}$$

Inference rule:

$$\frac{\qquad\qquad\qquad\qquad}{\Gamma \vdash f(e) : \gamma}$$

Typing rule:

$$\frac{\Gamma \vdash f : \sigma \mathrel{-\!\!>} \tau \qquad \Gamma \vdash e : \sigma}{\Gamma \vdash f(e) : \tau}$$

Inference rule:

$$\frac{\Gamma \vdash f : \alpha \qquad\qquad\qquad}{\Gamma \vdash f(e) : \gamma}$$

Typing rule:

$$\frac{\Gamma \vdash f : \sigma \;{-}{>}\; \tau \qquad \Gamma \vdash e : \sigma}{\Gamma \vdash f(e) : \tau}$$

Inference rule:

$$\frac{\Gamma \vdash f : \alpha \qquad \Gamma \vdash e : \beta}{\Gamma \vdash f(e) : \gamma}$$

Typing rule:

$$\frac{\Gamma \vdash f : \sigma \; {-}{>} \; \tau \qquad \Gamma \vdash e : \sigma}{\Gamma \vdash f(e) : \tau}$$

Inference rule:

$$\frac{\Gamma \vdash f : \alpha \qquad \Gamma \vdash e : \beta}{\Gamma \vdash f(e) : \gamma} \quad \boxed{\alpha \approx \beta \; {-}{>} \; \gamma}$$

Typing rule:

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash (\texttt{fn}\; x \;\texttt{=>}\; e) : \sigma \;\texttt{->}\; \tau}$$

Typing rule:

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash (\mathtt{fn}\ x \Rightarrow e) : \sigma \ {-}{>}\ \tau}$$

Inference rule:

$$\frac{}{\Gamma \vdash (\mathtt{fn}\ x \Rightarrow e) : \gamma}$$

Typing rule:

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash (\texttt{fn}\ x \texttt{ => } e) : \sigma \texttt{ -> } \tau}$$

Inference rule:

$$\frac{\Gamma, x : \alpha \vdash e : \beta}{\Gamma \vdash (\texttt{fn}\ x \texttt{ => } e) : \gamma}$$

Typing rule:

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash (\mathtt{fn}\ x \Rightarrow e) : \sigma \rightarrow \tau}$$

Inference rule:

$$\frac{\Gamma, x : \alpha \vdash e : \beta}{\Gamma \vdash (\mathtt{fn}\ x \Rightarrow e) : \gamma} \quad \boxed{\gamma \approx \alpha \rightarrow \beta}$$

Example:

$$\frac{}{\vdash \texttt{fn}\ \mathsf{f} \Rightarrow \texttt{fn}\ \mathsf{x} \Rightarrow \mathsf{f}(\mathsf{f}(\mathsf{x})) : \alpha_0}$$

Example:

$$\frac{f : \alpha_1 \vdash \texttt{fn}\ x \texttt{ => } f(f(x)) : \alpha_2}{\vdash \texttt{fn}\ f \texttt{ => } \texttt{fn}\ x \texttt{ => } f(f(x)) : \alpha_0}$$

$$\boxed{\alpha_0 \approx \alpha_1 \texttt{->} \alpha_2}$$

Example:

$$\dfrac{\dfrac{\overline{f : \alpha_1, x : \alpha_3 \vdash f(f(x)) : \alpha_4}}{f : \alpha_1 \vdash \texttt{fn}\ x \Rightarrow f(f(x)) : \alpha_2}}{\vdash \texttt{fn}\ f \Rightarrow \texttt{fn}\ x \Rightarrow f(f(x)) : \alpha_0}$$

$$\boxed{\alpha_0 \approx \alpha_1 \to \alpha_2 \ , \quad \alpha_2 \approx \alpha_3 \to \alpha_4}$$

$\Rightarrow \quad \alpha_0 \approx \alpha_1 \to (\alpha_3 \to \alpha_4)$

Example:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{f : \alpha_1, x : \alpha_3 \vdash f : \alpha_5} \qquad \dfrac{}{f : \alpha_1, x : \alpha_3 \vdash f(x) : \alpha_6}}{f : \alpha_1, x : \alpha_3 \vdash f(f(x)) : \alpha_4} \ast}{f : \alpha_1 \vdash \texttt{fn}\ x \Rightarrow f(f(x)) : \alpha_2}}{\vdash \texttt{fn}\ f \Rightarrow \texttt{fn}\ x \Rightarrow f(f(x)) : \alpha_0}$$

$$\boxed{\alpha_0 \approx \alpha_1 \texttt{->} \alpha_2\ , \quad \alpha_2 \approx \alpha_3 \texttt{->} \alpha_4\ , \quad \underline{\alpha_5 \approx \alpha_6 \texttt{->} \alpha_4} \ast}$$

Example:

$$\dfrac{\dfrac{\dfrac{\dfrac{\surd}{f : \alpha_1, x : \alpha_3 \vdash f : \alpha_5} \qquad \dfrac{}{f : \alpha_1, x : \alpha_3 \vdash f(x) : \alpha_6}}{f : \alpha_1, x : \alpha_3 \vdash f(f(x)) : \alpha_4}}{f : \alpha_1 \vdash \texttt{fn}\ x \texttt{ => } f(f(x)) : \alpha_2}}{\vdash \texttt{fn}\ f \texttt{ => } \texttt{fn}\ x \texttt{ => } f(f(x)) : \alpha_0}$$

$$\boxed{\alpha_0 \approx \alpha_1 \texttt{->} \alpha_2\ , \quad \alpha_2 \approx \alpha_3 \texttt{->} \alpha_4\ , \quad \alpha_5 \approx \alpha_6 \texttt{->} \alpha_4\ , \quad \alpha_5 \approx \alpha_1}$$

$$\Downarrow \quad \alpha_0 \approx (\alpha_6 \to \alpha_4) \to (\alpha_3 \to \alpha_4)$$

Example:

$$\dfrac{\dfrac{\dfrac{\dfrac{\surd}{f : \alpha_1, x : \alpha_3 \vdash f : \alpha_5} \qquad \dfrac{\dfrac{}{f : \alpha_1, x : \alpha_3 \vdash f : \alpha_7} \qquad \dfrac{}{f : \alpha_1, x : \alpha_3 \vdash x : \alpha_8}}{f : \alpha_1, x : \alpha_3 \vdash f(x) : \alpha_6}}{f : \alpha_1, x : \alpha_3 \vdash f(f(x)) : \alpha_4}}{f : \alpha_1 \vdash \texttt{fn}\ x \texttt{=>} f(f(x)) : \alpha_2}}{\vdash \texttt{fn}\ f \texttt{=>} \texttt{fn}\ x \texttt{=>} f(f(x)) : \alpha_0}$$

$$\boxed{\begin{array}{l} \alpha_0 \approx \alpha_1 \texttt{->} \alpha_2\ , \quad \alpha_2 \approx \alpha_3 \texttt{->} \alpha_4\ , \quad \alpha_5 \approx \alpha_6 \texttt{->} \alpha_4\ , \quad \alpha_5 \approx \alpha_1 \\ \qquad\qquad \alpha_7 \approx \alpha_8 \texttt{->} \alpha_6 \end{array}}$$

Example:

$$\dfrac{\dfrac{\dfrac{\dfrac{\surd}{f:\alpha_1,x:\alpha_3\vdash f:\alpha_5}\qquad \dfrac{\dfrac{\surd}{f:\alpha_1,x:\alpha_3\vdash f:\alpha_7}\qquad\dfrac{}{f:\alpha_1,x:\alpha_3\vdash x:\alpha_8}}{f:\alpha_1,x:\alpha_3\vdash f(x):\alpha_6}}{f:\alpha_1,x:\alpha_3\vdash f(f(x)):\alpha_4}}{f:\alpha_1\vdash \texttt{fn}\ x\ \texttt{=>}\ f(f(x)):\alpha_2}}{\vdash \texttt{fn}\ f\ \texttt{=>}\ \texttt{fn}\ x\ \texttt{=>}\ f(f(x)):\alpha_0}$$

$$\boxed{\begin{array}{c}\alpha_0 \approx \alpha_1 \texttt{->} \alpha_2\ ,\quad \alpha_2 \approx \alpha_3 \texttt{->} \alpha_4\ ,\quad \alpha_5 \approx \alpha_6 \texttt{->} \alpha_4\ ,\quad \alpha_5 \approx \alpha_1 \\ \alpha_7 \approx \alpha_8 \texttt{->} \alpha_6\ ,\quad \alpha_7 \approx \alpha_1\end{array}}$$

Example:

$$\dfrac{\dfrac{\dfrac{\sqrt{}}{f:\alpha_1, x:\alpha_3 \vdash f:\alpha_5} \qquad \dfrac{\dfrac{\sqrt{}}{f:\alpha_1, x:\alpha_3 \vdash f:\alpha_7} \qquad \dfrac{\sqrt{}}{f:\alpha_1, x:\alpha_3 \vdash x:\alpha_8}}{f:\alpha_1, x:\alpha_3 \vdash f(x):\alpha_6}}{\dfrac{f:\alpha_1, x:\alpha_3 \vdash f(f(x)):\alpha_4}{f:\alpha_1 \vdash \texttt{fn}\ x \Rightarrow f(f(x)):\alpha_2}}}{\vdash \texttt{fn}\ f \Rightarrow \texttt{fn}\ x \Rightarrow f(f(x)):\alpha_0}$$

$$\boxed{\begin{array}{c} \alpha_0 \approx \alpha_1 \to \alpha_2\,, \quad \alpha_2 \approx \alpha_3 \to \alpha_4\,, \quad \alpha_5 \approx \alpha_6 \to \alpha_4\,, \quad \alpha_5 \approx \alpha_1 \\ \alpha_7 \approx \alpha_8 \to \alpha_6\,, \quad \alpha_7 \approx \alpha_1\,, \quad \alpha_8 \approx \alpha_3 \end{array}}$$

$\to$ find the most general unifier

Example:

$$\dfrac{\dfrac{\dfrac{\dfrac{\surd}{f:\alpha_1, x:\alpha_3 \vdash f:\alpha_5} \qquad \dfrac{\dfrac{\surd}{f:\alpha_1, x:\alpha_3 \vdash f:\alpha_7} \qquad \dfrac{\surd}{f:\alpha_1, x:\alpha_3 \vdash x:\alpha_8}}{f:\alpha_1, x:\alpha_3 \vdash f(x):\alpha_6}}{f:\alpha_1, x:\alpha_3 \vdash f(f(x)):\alpha_4}}{f:\alpha_1 \vdash \texttt{fn}\ x \texttt{ => } f(f(x)):\alpha_2}}{\vdash \texttt{fn}\ f \texttt{ => } \texttt{fn}\ x \texttt{ => } f(f(x)):\alpha_0}$$

$$\boxed{\begin{array}{c} \alpha_0 \approx \alpha_1 \texttt{->} \alpha_2\,, \quad \alpha_2 \approx \alpha_3 \texttt{->} \alpha_4\,, \quad \alpha_5 \approx \alpha_6 \texttt{->} \alpha_4\,, \quad \alpha_5 \approx \alpha_1 \\ \alpha_7 \approx \alpha_8 \texttt{->} \alpha_6\,, \quad \alpha_7 \approx \alpha_1\,, \quad \alpha_8 \approx \alpha_3 \end{array}}$$

$\forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha$ is the type of Church numerals

Solution: $\alpha_0 = (\alpha_3 \texttt{->} \alpha_3) \texttt{->} \alpha_3 \texttt{->} \alpha_3$

Polymorphism

Polymorphism, which literally means “having multiple forms”, refers to constructs that can take on different types as needed.

Forms of polymorphism in contemporary programming languages:

Parametric polymorphism. A function may be applied to any arguments whose types match a type expression involving type variables.

Parametric polymorphism may be:

Implicit. Programs do not need to contain types; types and instantiations of type variables are computed.

Example: SML.

Explicit. The program text contains type variables that determine the way that a construct may be treated polymorphically.

Explicit polymorphism often involves explicit instantiation or type application to indicate how type variables are replaced with specific types in the use of a polymorphic construct.

Example: C++ templates.

Ad hoc polymorphism or overloading. Two or more implementations with different types are referred to by the same name.

Subtype polymorphism. The subtype relation between types allows an expression to have many possible types.

let-polymorphism

- The standard sugaring

$$\texttt{let val } x = v \texttt{ in } e \texttt{ end} \quad \mapsto \quad (\texttt{fn } x \texttt{ => } e)(v)$$

does not respect ML type checking.

For instance

$$\texttt{let val f = fn x => x in f(f) end}$$

type checks, whilst

$$(\texttt{fn f => f(f)})(\texttt{fn x => x})$$

does not.

L Exercise Run the type inference algorithm on this expression