CS40032: Principles of Programming Languages
Module 04: Typed λ-Calculus

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

Source: Foundations of Object-Oriented Languages – Types and Semantics by Kim B. Bruce, The MIT Press, 2002

Jan 20 & 25, Feb 01: 2021

Table of Contents

$\Lambda^{\rightarrow}$: Simply-Typed λ-Calculus

Type Expression

- We start with an arbitrary collection, $\mathcal{TC}$, of type constants (which may include *Integer*, *Boolean*, etc.)
- The set, *Type*, of *type expressions* of the simply-typed λ-calculus, $\Lambda^{\rightarrow}$, is given by:

$$T \in Type ::= C \mid T_1 \rightarrow T_2 \mid (T)$$

  where $C \in \mathcal{TC}$

- Clearly this definition is parameterized by $\mathcal{TC}$, but for simplicity, we do not show this in the notation *Type*

Type Expression

$$T \in Type ::= C \mid T_1 \rightarrow T_2 \mid (T)$$

- The set *Type* is composed of
  1. Type constants C from the set $\mathcal{TC}$,
  2. Expressions of the form $T_1 \rightarrow T_2$ where $T_1, T_2 \in Type$
  3. Expressions of the form (T) where $T \in Type$
- In other words, type expressions are built up from type constants, C, by constructing function types, and using parentheses to group type expressions.
- Typical elements of *Type* include *Integer*, *Boolean*, $Integer \rightarrow Integer$, $Boolean \rightarrow (Integer \rightarrow Integer)$, and $(Boolean \rightarrow Integer) \rightarrow Integer$

Pre-Expression & Expression

- When defining the expressions of a typed programming language, we need to distinguish the **pre-expressions** from the **expressions** of the language
- The *pre-expressions* are syntactically correct, but may not be typeable, while the *expressions* are those that pass the type checker

Constant Expression

- Expressions of the typed λ-calculus can include elements from an arbitrary set of constant expressions, $\mathcal{EC}$
- Each of these constants comes with an associated type.
- For example, $\mathcal{EC}$ might include constants representing integers such as $\underline{0}$, $\underline{1}$, . . . , all with type *Integer*; booleans such as $\underline{true}$ and $\underline{false}$, with type *Boolean*; and operations such as $\underline{plus}$ and $\underline{mult}$ with type

$$Integer \rightarrow Integer \rightarrow Integer$$

  (prefix versions of "+" & "*")
- We will leave $\mathcal{EC}$ unspecified most of the time, using constant symbols freely where it enhances our examples

Pre-Expression

- The collection of **pre-expressions** of the typed λ-calculus, $\mathcal{TLCE}$ (*Typed Lambda Calculus Expressions*), are given with respect to
  - a collection of type constants, $\mathcal{TC}$,
  - a collection of expression identifiers, $\mathcal{EI}$, and
  - a collection of expression constants, $\mathcal{EC}$:

$$M, N \in \mathcal{TLCE} ::= c \mid x \mid \lambda(x : T).\ M \mid M\ N \mid (M)$$

  where $x \in \mathcal{EI}$ and $c \in \mathcal{EC}$
- As with types, this definition is parameterized by the choice of $\mathcal{TC}$, $\mathcal{EI}$, and $\mathcal{EC}$

Pre-Expression

$$M, N \in \mathcal{TLCE} ::= c \mid x \mid \lambda(x : T).\ M \mid M\ N \mid (M)$$

- Pre-expressions of $\mathcal{TLCE}$, typically written as M, N (or variants decorated with primes or subscripts), are composed of constants, c, from $\mathcal{EC}$; identifiers, x, from $\mathcal{EI}$; function definitions, $\lambda(x : T).\ M$; and function applications, $M\ N$
- Also, as with type expressions, any pre-expression, M, may be surrounded by parentheses, (M)
- All formal parameters in function definitions are associated with a type
- We treat function application as having higher precedence than λ-abstraction. Thus $\lambda(x : T).\ M\ N$ is equivalent to $\lambda(x : T).\ (M\ N)$

Pre-Expression

- In order to complete the specification of expressions of the typed λ-calculus, we need to write down type-checking rules that can be used to determine if a pre-expression is type correct
- Expressions being type checked often include identifiers, typically introduced as formal parameters along with their types
- In order to type check expressions we need to know what the type is for each identifier

Expression

- The collection of expressions of the typed λ-calculus with respect to $\mathcal{TC}$ and $\mathcal{EC}$ is the collection of pre-expressions which can be assigned a type by the type-checking rules

Free & Bound Identifiers

- **Definition**: The collection of **free identifiers** of an expression M, written $FI(M)$, is defined as follows:
  1. $FI(c) \triangleq \phi$, for $c \in \mathcal{EC}$
  2. $FI(x) \triangleq \{x\}$, for $x \in \mathcal{EI}$
  3. $FI(\lambda(x : T).\ M) \triangleq FI(M) - \{x\}$
  4. $FI(M\ N) \triangleq FI(M) \cup FI(N)$
- When an identifier is used as a formal parameter of a function, its occurrences in the function body are no longer free – we say they are **bound identifiers**
- For example

$$FI((plus\ x)\ y) = \{x, y\}$$

but

$$FI(\lambda(x : Integer).\ (plus\ x)\ y) = \{y\}$$

Static Type Environment, $\mathcal{E}$

- Bound identifiers are supplied with a type when they are declared as formal parameters, but free identifiers are not textually associated with types in the expressions containing them
- The type-checking rules require information about the type s of free identifiers
- **Static type environment**, $\mathcal{E}$ associates types with free expression identifiers
- **Definition**: A static type environment, $\mathcal{E}$, is a finite set of associations between identifiers and type expressions of the form $x : T$, where each x is unique in $\mathcal{E}$ and T is a type
- If $x : T \in \mathcal{E}$, then we sometimes write $\mathcal{E}(x) = T$

Type-checking Rules

- **Type-checking rules** can be in one of two forms
  1. A rule of the form

     $$\mathcal{E} \vdash M : T$$

     OR

     $$\overline{\mathcal{E} \vdash M : T}$$

     indicates that with the typing of free identifiers in $\mathcal{E}$, the expression M has type T
  2. A rule of the form

     $$\frac{\mathcal{E} \vdash M_1 : T_1, \cdots, \mathcal{E} \vdash M_n : T_n}{\mathcal{E} \vdash M : T}$$

     indicates that with the typing of free identifiers in $\mathcal{E}$, the expression M has type T *if the assertions above the horizontal line all hold*
- The hypotheses of a rule all occur above the horizontal line, while the conclusion is placed below the line

Type-checking Rules

Identifier	$\dfrac{}{\mathcal{E} \cup \{x{:}T\} \vdash x{:}T}$
Constant[1]	$\dfrac{}{\mathcal{E} \vdash c \in C}$
Function	$\dfrac{\mathcal{E} \cup \{x{:}T\} \vdash M{:}T'}{\mathcal{E} \vdash \lambda(x{:}T).M{:}T \rightarrow T'}$
Application	$\dfrac{\mathcal{E} \vdash M{:}T \rightarrow T',\ \mathcal{E} \vdash N{:}T}{\mathcal{E} \vdash M\ N{:}T'}$
Paren	$\dfrac{\mathcal{E} \vdash M{:}T}{\mathcal{E} \vdash (M){:}T}$

[1] $C \in \mathcal{TC}$ is the pre-assigned type for constant $c \in \mathcal{EC}$

Type-checking Rules – How to Apply?

- Read the rules from the bottom-left in a clockwise direction (example, Application Rule)

$$\frac{\mathcal{E} \ \vdash \ M : T \rightarrow T',\ \mathcal{E} \ \vdash \ N : T}{\mathcal{E} \ \vdash \ M \ N : T'}$$

- To type check $M \ N$ under $\mathcal{E}$, use Application Rule
- Proceed clockwise to the top of the rule – now we need to find the types of M and N under $\mathcal{E}$
- If M has type $T \rightarrow T'$ and N has type T, then the resulting type of $M \ N$ is T'

Type-checking Rules: Identifier Rule

Identifier Rule:

$$\frac{}{\mathcal{E} \cup \{x : T\} \vdash x : T}$$

If $\mathcal{E}$ indicates that identifier x has type T, then x has that type

Type-checking Rules: Constant Rule

Constant Rule:

$$\frac{}{\mathcal{E} \vdash c \in C}$$

A constant has whatever type is associated with it in $\mathcal{EC}$

Type-checking Rules: Function Rule

Function Rule:

$$\frac{\mathcal{E} \ \cup \ \{x : T\} \ \vdash \ M : T'}{\mathcal{E} \ \vdash \ \lambda(x : T).M : T \rightarrow T'}$$

- As the formal parameter of the function occurs in the body, the formal parameter and its type need to be added to the environment when type checking the body
- Thus if $\lambda(x : T).\ M$ is type checked in environment $\mathcal{E}$, then the body, M, should be type checked in the environment $\mathcal{E} \ \cup \ \{x : T\}$. (Recall that the environment $\mathcal{E} \ \cup \ \{x : T\}$ is legal only if x does not already occur in $\mathcal{E}$.)
- For example, in typing the function $\lambda(x : \mathit{Integer}).x + \underline{1}$, the body, $x + \underline{1}$, should be type checked in an environment in which x has type *Integer*

Type-checking Rules: Application Rule

Application Rule:

$$\frac{\mathcal{E} \vdash M : T \rightarrow T',\ \mathcal{E} \vdash N : T}{\mathcal{E} \vdash M\ N : T'}$$

A function application $M\ N$ has type T' as long as the type of the function, M, is of the form $T \rightarrow T'$, and the actual argument, N, has type T, matching the type of the domain of M

Type-checking Rules: Paren Rule

Paren Rule:

$$\frac{\mathcal{E} \ \vdash \ M : T}{\mathcal{E} \ \vdash \ (M) : T}$$

Adding parentheses has no effect on the type of an expression

Type-checking Rules: Example 1

Determine the type of

$$\lambda(x : Integer).\ (\underline{plus}\ x)\ x$$

where $\underline{plus}$ be the constant with type

$$Integer \rightarrow Integer \rightarrow Integer$$

and

$$\mathcal{E}_0 = \phi$$

Type-checking Rules: Example 1

Initially,

$$\mathcal{E}_0 \vdash \lambda(x : Integer).\ (\underline{plus}\ x)\ x :??$$

By the *Function Rule*

$$\frac{\mathcal{E} \cup \{x : T\} \vdash M : T'}{\mathcal{E} \vdash \lambda(x : T).M : T \rightarrow T'}$$

to type check this function we must check the body $(\underline{plus}\ x)\ x$ in the environment

$$\mathcal{E}_1 = \mathcal{E}_0 \cup \{x :\ Integer\} = \phi \cup \{x :\ Integer\} = \{x :\ Integer\} :$$

$$\mathcal{E}_1 \vdash (\underline{plus}\ x)\ x :??$$

Type-checking Rules: Example 1

Because the body is a function application, we type check the function and argument to make sure their types match
Type checking the argument is easy as:

$$\mathcal{E}_1 \ \vdash \ x : Integer \quad \cdots (1)$$

by *Identifier Rule*

$$\mathcal{E} \ \cup \ \{x : T\} \ \vdash \ x : T$$

Type-checking Rules: Example 1

Type-checking $\underline{plus}\ x$ is a bit more complex, because it is a function application as well. However, by *Constant Rule*,

$$\mathcal{E}_1 \ \vdash \ \underline{plus} : Integer \rightarrow Integer \rightarrow Integer \qquad \cdots(2)$$

and by *Identifier Rule*, we again get

$$\mathcal{E}_1 \ \vdash \ x : Integer$$

Because the domain of the type of $\underline{plus}$ and the type of x are the same,

$$\mathcal{E}_1 \ \vdash \ \underline{plus}\ x : Integer \rightarrow Integer \qquad \cdots(3)$$

by lines (2), (1), and the *Application rule*

$$\frac{\mathcal{E} \ \vdash \ M : T \rightarrow T', \ \mathcal{E} \ \vdash \ N : T}{\mathcal{E} \ \vdash \ M\ N : T'}$$

Type-checking Rules: Example 1

By another use of *Application rule*

$$\frac{\mathcal{E} \vdash M : T \rightarrow T', \ \mathcal{E} \vdash N : T}{\mathcal{E} \vdash M\ N : T'}$$

with (3) and (1),

$$\mathcal{E}_1 \vdash (\underline{plus}\ x)\ x : Integer \quad \cdots (4)$$

Type-checking Rules: Example 1

Finally by *Function rule*

$$\frac{\mathcal{E} \ \cup \ \{x : T\} \ \vdash \ M : T'}{\mathcal{E} \ \vdash \ \lambda(x : T).M : T \rightarrow T'}$$

and (4),

$$\mathcal{E}_0 \ \vdash \ \lambda(x : Integer).\ (\underline{plus}\ x)\ x : Integer \rightarrow Integer \quad \cdots (5)$$

$\square$

Type-checking Rules: Example 1

$$\mathcal{E}_1 \ \vdash \ x : Int \quad \cdots(1, Id)$$

$$\mathcal{E}_1 \ \vdash \ \underline{plus} : Int \rightarrow Int \rightarrow Int \quad \cdots(2, Const)$$

$$\mathcal{E}_1 \ \vdash \ \underline{plus}\ x : Int \rightarrow Int \quad \cdots(3, App)$$

$$\mathcal{E}_1 \ \vdash \ (\underline{plus}\ x)\ x : Int \quad \cdots(4, App)$$

$$\mathcal{E}_0 \ \vdash \ \lambda(x : Int).\ (\underline{plus}\ x)\ x : Int \rightarrow Int \quad \cdots(5, Func)$$

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\mathcal{E}_1 \ \vdash \ \underline{plus} : Int \rightarrow Int \rightarrow Int}(2) \quad \cfrac{}{\mathcal{E}_1 \ \vdash \ x : Int}(1)}{\mathcal{E}_1 \ \vdash \ \underline{plus}\ x : Int \rightarrow Int}(3) \quad \cfrac{}{\mathcal{E}_1 \ \vdash \ x : Int}(1)}{\mathcal{E}_1 \ \vdash \ (\underline{plus}\ x)\ x : Int}(4)}{\mathcal{E}_0 \ \vdash \ \lambda(x : Int).\ (\underline{plus}\ x)\ x : Int \rightarrow Int}(5)$$

Type-checking Rules: Example 2

Determine the type of

$$(\lambda(x : Integer).\ (\underline{plus}\ x)\ x)\underline{17}$$

where $\underline{plus}$ be the constant with type

$$Integer \rightarrow Integer \rightarrow Integer$$

and

$$\mathcal{E}_0 = \phi$$

Type-checking Rules: Example 2

From (5),

$$\mathcal{E}_0 \vdash \lambda(x : Integer).\ (\underline{plus}\ x)\ x : Integer \rightarrow Integer$$

and

$$\mathcal{E}_0 \vdash \underline{17} : Integer$$

Hence using *Application rule*

$$\frac{\mathcal{E} \vdash M : T \rightarrow T',\ \mathcal{E} \vdash N : T}{\mathcal{E} \vdash M\ N : T'}$$

we get

$$\mathcal{E}_0 \vdash (\lambda(x : Integer).\ (\underline{plus}\ x)\ x)\ \underline{17} : Integer \quad \cdots (6)$$

Type-checking Rules: Example 3

Determine the type of

$$(\lambda(x : Integer).\ x + \underline{40})\underline{2}$$

where

$$\mathcal{E}_0 = \phi$$

Type-checking Rules: Example 3

Determine the type of

$$(\lambda(x : Integer).\ x + \underline{40})\underline{2}$$

where

$$\mathcal{E}_0 = \phi$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathcal{E}_1 \vdash \underline{plus} : Int \to Int \to Int}(2) \quad \dfrac{}{\mathcal{E}_1 \vdash x : Int}(1)}{\mathcal{E}_1 \vdash \underline{plus}\ x : Int \to Int}(3) \quad \dfrac{}{\mathcal{E}_1 \vdash \underline{40} : Int}(4)}{\mathcal{E}_1 \vdash (\underline{plus}\ x)\ \underline{40} : Int}(4)}{\mathcal{E}_0 \vdash \lambda(x : Int).\ (\underline{plus}\ x)\ \underline{40} : Int \to Int}(5) \quad \dfrac{}{\mathcal{E}_1 \vdash \underline{2} : Int}(6)}{\mathcal{E}_0 \vdash (\lambda(x : Int).\ (\underline{plus}\ x)\ \underline{40})\underline{2} : Int}$$

Type-checking Rules: Example 4

Determine the type of

$$(\lambda(p : Int \rightarrow Bool).\lambda(f : Int \rightarrow Int).\lambda(x : Int)).\ p(f\ x)$$

where

$$\mathcal{E}_0 = \phi$$

Type-checking Rules: Example 4

Determine the type of

$$(\lambda(p : Int \rightarrow Bool).\lambda(f : Int \rightarrow Int).\lambda(x : Int)).\ p(f\ x)$$

where

$$\mathcal{E}_0 = \phi$$

$$\cfrac{\cfrac{\cfrac{}{\mathcal{E}_1 \ \vdash \ f : Int \rightarrow Int}(2) \quad \cfrac{}{\mathcal{E}_1 \ \vdash \ x : Int}(1)}{\mathcal{E}_1 \ \vdash \ p : Int \rightarrow Bool}(4) \quad \cfrac{}{\mathcal{E}_1 \ \vdash \ f\ x : Int}(3)}{\mathcal{E}_1 \ \vdash \ p\ (f\ x)\ x : Bool}(5)$$

Type-checking Rules: Example 5

Determine the type of

$$(\lambda(x : Bool).\ x)\ true$$

where

$$Bool \in \mathcal{TC}, true : Bool \in \mathcal{EC}, \mathcal{E}_0 = \phi$$

Type-checking Rules: Example 5

Determine the type of

$$(\lambda(x : Bool).\ x)\ true$$

where

$$Bool \in \mathcal{TC}, true : Bool \in \mathcal{EC}, \mathcal{E}_0 = \phi$$

$$\dfrac{\dfrac{\overline{\mathcal{E}_0, x : Bool \vdash x : Bool}}{\mathcal{E}_0 \vdash (\lambda(x : Bool).\ x) : Bool \rightarrow Bool} \quad \overline{\mathcal{E}_0 \vdash true : Bool}}{\mathcal{E}_0 \vdash (\lambda(x : Bool).\ x)\ true : Bool}$$

Type-checking Rules: Example 6

Show

$$\mathcal{E}_0,\ f : Bool \rightarrow Bool \vdash f\ (if\ \ false\ \ then\ \ true\ \ else\ \ false) : Bool$$

where

$$Bool \in \mathcal{TC}, true : Bool \in \mathcal{EC}, \mathcal{E}_0 = \phi$$

Type-checking Rules: Example 7

Show

$$\mathcal{E}_0,\ f : Bool \rightarrow Bool \vdash$$

$$\lambda x : Bool\ f\ (if\ \ false\ \ then\ \ true\ \ else\ \ false) : Bool \rightarrow Bool$$

where

$$Bool \in \mathcal{TC}, true : Bool \in \mathcal{EC}, \mathcal{E}_0 = \phi$$

Practice Problems

1. $$(\lambda(x : \ Float).(mult\ x)\ x\ )\ \underline{40.5}$$

   Let `mult` be a constant of type `Float` $\rightarrow$ `Float` $\rightarrow$ `Float` and let $\underline{40.5}$ be a constant of type `Float`.

   Solution

   Float

2. $$\lambda(g : \ Bool\ \ \rightarrow Char).\ \lambda(x : \ Bool).\ g\ (\ x\ \&\ \underline{true}\ )$$

   Let `&` be the constant with the type `Bool` $\rightarrow$ `Bool` $\rightarrow$ `Bool`. The type of <u>`true`</u> is `Bool`

   Solution

   (Bool $\rightarrow$ Char) $\rightarrow$ (Bool $\rightarrow$ Char)

3. $$\lambda(p : \ Float\ \ \rightarrow Integer).\ \lambda(f : \ Float\ \ \rightarrow Float).\ \lambda(y : \ Float).\ p\ (f\ (f\ y))$$

   Solution

   (Float $\rightarrow$ Integer) $\rightarrow$ ((Float $\rightarrow$ Float) $\rightarrow$ (Float $\rightarrow$ Integer))

Practice Problems (Contd...)

4. Given $+$ are type constant with the type $\phi \rightarrow \phi$.
   $\lambda(* : \phi \ \rightarrow \tau).\ \lambda(x : \phi).\ \ * \ (+x)$

   Solution

   $(\phi \rightarrow \tau) \rightarrow (\phi \rightarrow \tau)$

5. $(\lambda(x : Integer).\ (\underline{f1}\ x)\ x)x$, where
   $\underline{f1} : Integer \rightarrow Integer \rightarrow Integer \rightarrow Integer \in \mathcal{CE}$ and x is of type *integer*

   Solution

   $(Integer \rightarrow Integer)$

6. $(\lambda(S : Char).\ (\underline{\alpha}\ S)\ S)S$, where $\underline{\alpha} : Char \rightarrow Char \rightarrow Char \rightarrow Char \in \mathcal{CE}$ and S is of type *char*

   Solution

   $(char \rightarrow char)$

7. $\lambda(p : A \ \rightarrow B).\ \lambda(\phi : A \ \rightarrow A \rightarrow A).\ \lambda(\beta : A \ \rightarrow A).\ \lambda(y : A).\ \lambda(x : A).\ p\ (\beta\ (\beta\ (\beta\ (x\ \phi\ y))))$

   Solution

   $(A \ \rightarrow B) \rightarrow ((A \ \rightarrow A \rightarrow A) \rightarrow ((A \rightarrow A) \rightarrow (A \rightarrow (A \rightarrow B))))$

8. $\lambda(g : A \ \rightarrow B).\ \lambda(x : A).\ g\ x$

   Solution

   $(A \rightarrow B) \rightarrow (A \rightarrow B)$

PoPL-04

Partha Pratim Das

$\Lambda^{\rightarrow}$

Type Expression

Pre-Expression & Expression

Type-checking Rules

Example

Practice Problems

$\Lambda^{\rightarrow}_{rr}$

Types

Tuple Type

Record Type

Sum Type

Reference Type

Array Type

Type Expression

Pre-Expression

Type-checking Rules

Derived Rules

Practice Problems (Contd...)

9. $\lambda(x : Integer).\ (\underline{plus}\ x)\ x$, where $\underline{plus} : Integer \rightarrow Integer \rightarrow Integer \in \mathcal{CE}$

   Solution

   $Integer \rightarrow Integer$

10. $\lambda(f : Int \rightarrow Int).\ \lambda(y : Int).\ f\ (f\ (f\ y))$

    Solution

    $(Int \rightarrow Int) \rightarrow (Int \rightarrow Int)$

$\Lambda_{rr}^{\rightarrow}$: Extended-Typed λ-Calculus

Extensions

$\Lambda^{\rightarrow}$, Simply-Typed λ-calculus is extended with

- tuples,
- records,
- sums, and
- references (variables)

to define $\Lambda_{rr}^{\rightarrow}$

$\Lambda^{\rightarrow}_{rr}$: Tuple Type

- Ordered tuples are written in the form $\langle a_1, \cdots, a_n \rangle$ and have type $T_1 \times \cdots \times T_n$ where each T_i is the type of the corresponding a_i
- Tuple types represent the domain of functions taking several parameters
- The projection operations, $proj_i$, extract the i^{th} component of a tuple. Thus

$$proj_i(\langle a_1, \cdots, a_n \rangle) = a_i$$

$\Lambda_{rr}^{\rightarrow}$: Tuple Type: *n*-ary Functions

- We write

$$\lambda(id_1 : T_1, \cdots, id_n : T_n).\ M$$

as an abbreviation for

$$\lambda(arg : T_1 \times \cdots \times T_n).[proj_i(arg)/id_i]_{i=1,\cdots,n}M$$

- Thus an *n*-ary function is an abbreviation for a function of a single argument that takes an *n*-tuple
- When expanded, each of the individual parameters is replaced by an appropriate projection from the *n*-tuple
- For example:

$$\lambda(x : Integer, y : Integer).\ plus\ x\ y$$

abbreviates

$$\lambda(p : Integer \times Integer).\ plus\ (proj_1(p))\ (proj_2(p))$$

$\Lambda_{rr}^{\rightarrow}$: Tuple Type: *n*-ary Functions & Semantics of C / C++

- Use of tuple type has always been prevalent in programming languages for *n*-ary functions. Naturally we can see the direct parallel between:

$$\lambda(id_1 : T_1, \cdots, id_n : T_n).\ M$$

and

```
T func(T1 id1, T2 id2, ..., Tn idn);
```

except for the syntactic differences and the need for specifying a return type. In λ calculus the return type is deduced by type checker, in C/C++ it is deduced and checked with the given return type for possible needs of conversion.

- For $\lambda(id_1 : int, id_2 : real).\ M$ we have the following variants in common programming languages

Language	Function Signature	Typing

Fortran	`integer x`
	`real y`
	`int function func(x, y)`
		static

PASCAL	`function func(x: integer, y: real): integer;`	static
C/C++	`int func(int x, double y);`	static
Java	`int func(int x, double y);`	static
Python	`def func(x, y):`	dynamic

$\Lambda^{\rightarrow}_{rr}$: Tuple Type: n-ary Functions: Example in C / C++

```
#include <iostream>
using namespace std;

struct Pair {   // Pair(int, double) = int x double
                // We use 'Pair' instead of 'pair' to avoid name clash with 'std::pair'
    int i;
    double d;
    Pair(int i, double d): i(i), d(d) { }
};

double f(int i, double d) { return i + d; } // int x double -> double

double f(Pair s) { return s.i + s.d; }      // Pair(int, double) -> double

int main() {
    const int i = 5;
    const double d = 2.6;
    Pair s(i, d);

    cout << f(i, d);
    cout << f(s);

    return 0;
}
```

$\Lambda_{rr}^{\rightarrow}$: Record Type

- Records are written in the form

$$\{|l_1 : T_1 := M_1, \cdots, l_n : T_n := M_n|\}$$

- Notice that each labeled field is provided with its type
- The type of a record of this form is written as

$$\{|l_1 : T_1, \cdots, l_n : T_n|\}$$

- Dot notation is used to extract the value of a field from a record:

$$\{|l_1 : T_1 := M_1, \cdots, l_n : T_n := M_n|\}.l_i = M_i$$

$\Lambda_{rr}^{\rightarrow}$: Record Type & Semantics of C / C++

- The record type

$$\{|l_1 : T_1, \cdots, l_n : T_n|\}$$

in λ parallels `struct` in C and `struct` or `class` in C++. So

$$\{|re : real, im : real|\}$$

is

```
struct Complex {
    double re;
    double im
}
```

in C/C++ and

```
class Complex {
    double re;
    double im
}
```

in C++

$\Lambda^{\rightarrow}_{rr}$: Sum Type

- A sum type,

$$T_1 + \cdots + T_n$$

  represents a disjoint union of the types, where each element contains information to indicate which summand it comes from, even if several of the T_i's are identical

- If M is an expression from a type T_i, the expression

$$in_i^{T_1,\cdots,T_n}(M)$$

  injects the value M into the i^{th} component of the sum $T_1 + \cdots + T_n$

$\Lambda^{\rightarrow}_{rr}$: Sum Type

- If M is an expression of type $T_1 + \cdots + T_n$, then an expression of the form

$$case\ M\ of\ x_1 : T_1\ then\ E_1 \mid\mid \cdots \mid\mid x_n : T_n\ then\ E_n$$

  represents a statement listing the possible expressions to evaluate depending on which summand M is a part of
- Thus if M was created by

$$in_i^{T_1, \cdots, T_n}(M')$$

  for some M' of type T_i then evaluating the *case* statement will result in evaluating E_i using M' as the value of x_i

$\Lambda^{\rightarrow}_{rr}$: Sum Type & Semantics of C / C++

- The sum type $S = T_1 + \cdots + T_n$, is a collection of disjoint types which is severally important in all programming languages.
- In C, this need has been met by `union { T1 x1; ··· ; Tn xn}` wrapped in a `S = struct { int tag; union { ··· } }` with a *type tag* (`tag`) – an effective but weak and error-prone solution where

$$myCase \equiv case\ M\ of\ x_1 : T_1\ then\ E_1 \mid\mid \cdots \mid\mid x_n : T_n\ then\ E_n$$

  is a `switch (M.tag) {case x1: E1; ··· case xn: En;}` and the injection

$$in_i^{T_1,\cdots,T_n}(M')$$

  is achieved by setting `M.tag = i` and `xi` to `M'`.
- In C++, this is achieved by (*smarter*) dynamic dispatch where `T1, ···, Tn` are specialized from the sum type `S` as the abstract base class. With this *case M of* is implemented as a (*pure virtual*) method `myCase()` in `S` which is overridden by the implementation of `Ei` in every `class Ti`. Injection of `M'` of `Ti` is obtained by constructing an object of `Ti` and setting a `const S` reference to it. Invocation of `M.myCase()` correctly calls the corresponding computation of `Ei` by run-tine polymorphism (*virtual function*).
- We elucidate both with examples.

$\Lambda^{\rightarrow}_{rr}$: Sum Type: Example 1

- The expression

$$M_1 \equiv in_1^{Integer,Integer}(\underline{5})$$

is an expression with type

$$Integer + Integer$$

It represents injecting the number 5 into the sum type as the first component

- The expression

$$M_2 \equiv in_2^{Integer,Integer}(\underline{7})$$

is an expression with type

$$Integer + Integer$$

It represents injecting the number 7 into the sum type as the second component

$\Lambda^{\rightarrow}_{rr}$: Sum Type: Example 1

- The following function takes elements of the sum type, $Integer + Integer$, and uses *case* to compute an integer value
- The value depends on whether the element of the sum type arose by injecting an integer as the first component or the second component:

$$
\begin{aligned}
myCase = \lambda(z : Integer + Integer).\ & case\ z\ of \\
& x : Integer\ then\ x\ +\ \underline{1}\ || \\
& y : Integer\ then\ y\ *\ \underline{2}
\end{aligned}
$$

- Hence,
  - $myCase\ M_1 = \underline{6}$
  - $myCase\ M_2 = \underline{14}$

$\Lambda^{\rightarrow}_{rr}$: Sum Type: Example 1: Using union in C

$Int + Int$: $M_1 \equiv in_1^{Int,Int}(\underline{5})$, $M_2 \equiv in_2^{Int,Int}(\underline{7})$

$myCase = \lambda(z : Int + Int).\ case\ z\ of\ x : Int\ then\ x\ +\ \underline{1}\ ||y : Int\ then\ y\ *\ \underline{2}$

```
#include <iostream>

enum tag_type {field1 = 0, field2 = 1};
union union_type {
    int i1;         // field1, Int
    int i2;         // field2, Int
    union_type(enum tag_type tag, int i)
    { (tag == field1)? i1 = i: i2 = i; }
};
struct sum_type {       // Int + Int
    enum tag_type tag;  // tag to remember injection component
    union union_type u;
    sum_type(enum tag_type t, int i): tag(t), u(tag, i) { } // in1, in2
};
int myCase(struct sum_type u) {         // z: Int + Int
    switch (u.tag) {                    // case z of
        case field1: return u.u.i1 + 1; // x: Int then x + 1 ||
        case field2: return u.u.i2 * 2; // y: Int then y * 2
    }
}
int main() {
    struct sum_type M1 = {field1, 5};   // M1 = in1(5)
    struct sum_type M2 = {field2, 7};   // M2 = in2(7)

    std::cout << myCase(M1); // 6
    std::cout << myCase(M2); // 14
    return 0;
}
```

$\Lambda_{rr}^{\rightarrow}$: Sum Type: Example 1: Using Inheritance & Dynamic Dispatch in C++

$Int + Int$: $M_1 \equiv in_1^{Int,Int}(\underline{5})$, $M_2 \equiv in_2^{Int,Int}(\underline{7})$

$myCase = \lambda(z : Int + Int).\ case\ z\ of\ x : Int\ then\ x\ +\ \underline{1}\ ||y : Int\ then\ y\ *\ \underline{2}$

```
#include <iostream>
struct sum_type {                    // Int + Int
    virtual int myCase() const = 0; // myCase type switch. z: Int + Int. case z of
};
struct T1: public sum_type {         // T1 = int Wrapped
    int data;
    T1(int d): data(d) { }
    int myCase() const;              // case of T1 type action
};
struct T2: public sum_type {         // T2 = int Wrapped
    int data;
    T2(int d): data(d) { }
    int myCase() const;              // case of T2 type action
};
int T1::myCase() const {             // case of T1 type action code
    return data + 1;                 // x: Int then x + 1
}
int T2::myCase() const {             // case of T2 type action code
    return data * 2;                 // y: Int then y * 2
}
int main() {
    const sum_type& M1 = T1(5);      // M1 = in1(5)
    const sum_type& M2 = T2(7);      // M2 = in2(7)

    std::cout << M1.myCase(); // 6
    std::cout << M2.myCase(); // 14
    return 0;
}
```

$\Lambda^{\rightarrow}_{rr}$: Sum Type: Example 2

- The expression

$$M_1 \equiv in_1^{Integer, Integer \rightarrow Integer}(\underline{47})$$

is an expression with type

$$Integer + (Integer \rightarrow Integer)$$

It represents injecting the number 47 into the sum type as the first component.

- The expression

$$M_2 \equiv in_2^{Integer, Integer \rightarrow Integer}(\underline{succ})$$

is an expression with type

$$Integer + (Integer \rightarrow Integer)$$

It represents injecting the function $\underline{succ} : Integer \rightarrow Integer$ into the sum type as the second component.

$\Lambda^{\rightarrow}_{rr}$: Sum Type: Example 2

- The following function takes elements of the sum type, $Integer + (Integer \rightarrow Integer)$, and uses *case* to compute an integer value
- The value depends on whether the element of the sum type arose by injecting an integer or by injecting a function from integers to integers:

$$
\begin{aligned}
isFirst = \lambda(y : Integer + (Integer \rightarrow Integer)).\ &case\ y\ of \\
&x : Integer\ then\ x\ +\ \underline{1}\ \| \\
&f : Integer \rightarrow Integer\ then\ f\ 0
\end{aligned}
$$

- Hence,
  - $isFirst\ M_1 = \underline{48}$
  - $isFirst\ M_2 = \underline{1}$

$\Lambda^{\rightarrow}_{rr}$: Sum Type: Example 2

$$
\begin{aligned}
isFirst = \lambda(y : Integer + (Integer \rightarrow Integer)).\ & case\ y\ of \\
& x : Integer\ then\ x\ +\ \underline{1}\ || \\
& f : Integer \rightarrow Integer\ then\ f\ 0
\end{aligned}
$$

- The parameter y comes from a sum type, the first of whose summands is *Integer*, while the second is the function type, $Integer \rightarrow Integer$
- If the parameter comes from the first summand, then it must represent an integer, x, and one is added to the value
- If it comes from the second summand, then it represents a function from integers to integers, denoted f, and the function is applied to 0
- Thus the value originally injected in the sum is represented by an identifier in the appropriate branch of the case, and thus can be used in determining the value to be returned.

$\Lambda^{\rightarrow}_{rr}$: Sum Type: Example 2: Using union in C

$Int + (Int \rightarrow Int)$: $M_1 \equiv in_1^{Int, Int \rightarrow Int}(\underline{47})$, $M_2 \equiv in_2^{Int, Int \rightarrow Int}(\underline{succ})$

$isFirst = \lambda(y : Int + (Int \rightarrow Int)).\ case\ y\ of\ x : Int\ then\ x + \underline{1}\ ||\ f : Int \rightarrow Int\ then\ f\ 0$

```
#include <iostream>
enum tag_type {field1 = 0, field2 = 1};
typedef int (*int2int)(int);    // Int -> Int
union union_type {
    int i1;                     // field1, Int
    int2int i2;                 // field2, Int -> Int
    union_type(int i): i1(i) { }
    union_type(int2int i): i2(i) { }
};
struct sum_type {                   // Int + Int -> Int
    enum tag_type tag;              // tag to remember injection component
    union union_type u;
    sum_type(int i): tag(field1), u(i) { }      // in1
    sum_type(int2int i): tag(field2), u(i) { }  // in2
};
int succ(int n) { return n+1; }         // succ: Int->Int
int isFirst(struct sum_type u) {        // y: Int + (Int -> Int)
    switch (u.tag) {                    // case y of
        case field1: return u.u.i1 + 1; // x: Int then x + 1
        case field2: return u.u.i2(0);  // f: Int -> Int then f 0
    }
}
int main() {
    struct sum_type M1 = 47;    // M1 = in1(47)
    struct sum_type M2 = succ;  // M2 = in2(succ)

    std::cout << isFirst(M1); // 48
    std::cout << isFirst(M2); // 1
    return 0;
}
```

$\Lambda^{\rightarrow}_{rr}$: Sum Type: Example 2: Using Inheritance & Dynamic Dispatch in C++

$Int + (Int \rightarrow Int)$: $M_1 \equiv in_1^{Int, Int \rightarrow Int}(\underline{47})$, $M_2 \equiv in_2^{Int, Int \rightarrow Int}(\underline{succ})$
$isFirst = \lambda(y : Int + (Int \rightarrow Int)).\ case\ y\ of\ x : Int\ then\ x + \underline{1}\ ||\ f : Int \rightarrow Int\ then\ f\ 0$

```
#include <iostream>
typedef int (*int2int)(int);             // Int -> Int
int succ(int n) { return n+1; }          // succ: Int->Int
struct sum_type {                        // Int + Int -> Int
    virtual int isFirst() const = 0;     // myCase type switch. y: Int + (Int -> Int). case y of
};
struct T1: public sum_type {             // T1 = int Wrapped
    int data;
    T1(int d): data(d) { }
    int isFirst() const;                 // case of T1 type action
};
struct T2: public sum_type {             // T2 = int2int
    int2int data;
    T2(int2int d): data(d) { }
    int isFirst() const;                 // case of T2 type action
};
int T1::isFirst() const                  // case of T1 type action code
{ return data + 1; }                     // x: Int then x + 1
int T2::isFirst() const                  // case of T2 type action code
{ return data(0); }                      // f: Int -> Int then f 0
int main() {
    const sum_type& M1 = T1(47);     // M1 = in1(47)
    const sum_type& M2 = T2(succ);   // M2 = in2(succ)

    std::cout << M1.isFirst(); // 48
    std::cout << M2.isFirst(); // 1
    return 0;
}
```

$\Lambda^{\rightarrow}_{rr}$: Sum Type: Example 3

- $M_1 \equiv in_1^{Integer, Integer \rightarrow Integer, Integer \times Integer}(\underline{25})$
- $M_2 \equiv in_2^{Integer, Integer \rightarrow Integer, Integer \times Integer}(\underline{succ})$
- $M_3 \equiv in_3^{Integer, Integer \rightarrow Integer, Integer \times Integer}(< \underline{12}, \underline{21} >)$
- Type $Integer + (Integer \rightarrow Integer) + Integer \times Integer$

$$
\begin{aligned}
isFirst = \lambda(y : Integer + (Integer \rightarrow Integer) + Integer \times Integer).\ case\ y\ of\ & \\
x : Integer\ then\ \underline{plus}\ x\ \underline{1}\ || & \\
f : Integer \rightarrow Integer\ then\ f\ \underline{7}\ || & \\
t : Integer \times Integer\ then\ \underline{plus}\ proj_1(t)\ proj_2(t) &
\end{aligned}
$$

$\Lambda^{\rightarrow}_{rr}$: Sum Type: Example 3

- $M_1 \equiv in_1^{Integer, Integer \rightarrow Integer, Integer \times Integer}(\underline{25})$
- $M_2 \equiv in_2^{Integer, Integer \rightarrow Integer, Integer \times Integer}(\underline{succ})$
- $M_3 \equiv in_3^{Integer, Integer \rightarrow Integer, Integer \times Integer}(< \underline{12}, \underline{21} >)$
- Type $Integer + (Integer \rightarrow Integer) + Integer \times Integer$

$$
\begin{aligned}
isFirst = \lambda(y : Integer + (Integer \rightarrow Integer) + Integer \times Integer).\ case\ y\ of\ & \\
x : Integer\ then\ \underline{plus}\ x\ \underline{1}\ & \| \\
f : Integer \rightarrow Integer\ then\ f\ \underline{7}\ & \| \\
t : Integer \times Integer\ then\ \underline{plus}\ proj_1(t)\ proj_2(t) &
\end{aligned}
$$

- Hence,
  - $isFirst\ M_1 = \underline{26}$
  - $isFirst\ M_2 = \underline{8}$
  - $isFirst\ M_3 = \underline{33}$

$\Lambda_{rr}^{\rightarrow}$: Sum Type: Example 3: Using union in C

$Int + (Int \rightarrow Int) + Int \times Int$: $M_1 \equiv in_1^{Int, Int \rightarrow Int, Int \times Int}(\underline{25})$, $M_2 \equiv in_2^{Int, Int \rightarrow Int, Int \times Int}(\underline{succ})$,
$M_3 \equiv in_3^{Int, Int \rightarrow Int, Int \times Int}(< \underline{12}, \underline{21} >)$
$isFirst = \lambda(y : Int + (Int \rightarrow Int) + Int \times Int).\ case\ y\ of\ x : Int\ then\ \underline{plus}\ x\ \underline{1}\ ||$
$f : Int \rightarrow Int\ then\ f\ \underline{7}\ ||\ t : Int \times Int\ then\ \underline{plus}\ proj_1(t)\ proj_2(t)$

```
#include <iostream>
enum tag_type {field1 = 0, field2 = 1, field3 = 2};
typedef int (*int2int)(int);    // Int -> Int
struct pair {                   // Int x Int
    int i1;
    int i2;
    pair(int i1_, int i2_): i1(i1_), i2(i2_) { }
};
union union_type {
    int i1;                     // field1, Int
    int2int i2;                 // field2, Int -> Int
    pair i3;                    // field3, Int x Int
    union_type(int i): i1(i) { }
    union_type(int2int i): i2(i) { }
    union_type(pair i): i3(i) { }
};
struct sum_type {                   // Int + Int -> Int + Int x Int
    enum tag_type tag;              // tag to remember injection component
    union union_type u;
    sum_type(int i): tag(field1), u(i) { }      // in1
    sum_type(int2int i): tag(field2), u(i) { }  // in2
    sum_type(pair i): tag(field3), u(i) { }     // in3
};
int succ(int n) { return n+1; }         // succ: Int->Int
```

$\Lambda_{rr}^{\rightarrow}$: Sum Type: Example 3: Using union in C

PoPL-04

Partha Pratim Das

$Int + (Int \rightarrow Int) + Int \times Int$: $M_1 \equiv in_1^{Int, Int \rightarrow Int, Int \times Int}(\underline{25})$, $M_2 \equiv in_2^{Int, Int \rightarrow Int, Int \times Int}(\underline{succ})$,
$M_3 \equiv in_3^{Int, Int \rightarrow Int, Int \times Int}(< \underline{12}, \underline{21} >)$
$isFirst = \lambda(y : Int + (Int \rightarrow Int) + Int \times Int).\ case\ y\ of\ x : Int\ then\ \underline{plus}\ x\ \underline{1}\ ||$
$f : Int \rightarrow Int\ then\ f\ \underline{7}\ ||\ t : Int \times Int\ then\ \underline{plus}\ proj_1(t)\ proj_2(t)$

```
// enum tag_type {field1 = 0, field2 = 1, field3 = 2};
// typedef int (*int2int)(int); // Int -> Int
// struct pair;                 // Int x Int
// union union_type;
// struct sum_type;             // Int + Int -> Int + Int x Int

int isFirst(struct sum_type u) {        // y: Int + (Int -> Int) + Int x Int
    switch (u.tag) {                    // case y of
        case field1: return u.u.i1 + 1; // x: Int then x + 1
        case field2: return u.u.i2(7);  // f: Int -> Int then f 7
        case field3: return u.u.i3.i1 + u.u.i3.i2;  // t: Int x Int then proj1(t) + proj2(t)
    }
}
int main() {
    struct sum_type M1 = 25;    // M1 = in1(25)
    struct sum_type M2 = succ;  // M2 = in2(succ)
    struct sum_type M3 = pair(12, 21);  // M3 = in2(<12, 21>)

    std::cout << isFirst(M1); // 26
    std::cout << isFirst(M2); // 8
    std::cout << isFirst(M3); // 33
    return 0;
}
```

$\Lambda^{\rightarrow}_{rr}$: Sum Type: Example 3: Using Inheritance & Dynamic Dispatch in C++

$Int + (Int \rightarrow Int) + Int \times Int$: $M_1 \equiv in_1^{Int, Int \rightarrow Int, Int \times Int}(\underline{25})$, $M_2 \equiv in_2^{Int, Int \rightarrow Int, Int \times Int}(\underline{succ})$,
$M_3 \equiv in_3^{Int, Int \rightarrow Int, Int \times Int}(< \underline{12}, \underline{21} >)$
$isFirst = \lambda(y : Int + (Int \rightarrow Int) + Int \times Int).\ case\ y\ of\ x : Int\ then\ \underline{plus}\ x\ \underline{1}\ ||$
$f : Int \rightarrow Int\ then\ f\ \underline{7}\ ||\ t : Int \times Int\ then\ \underline{plus}\ proj_1(t)\ proj_2(t)$

```
#include <iostream>
typedef int (*int2int)(int);             // Int -> Int
struct pair {                            // Int x Int
    int i1, i2;
    pair(int i1_, int i2_): i1(i1_), i2(i2_) { }
};
int succ(int n) { return n+1; }          // succ: Int->Int
struct sum_type {                        // Int + (Int -> Int) + Int x Int
    virtual int isFirst() const = 0;     // myCase type switch.
                                         // y: Int + (Int -> Int) + Int x Int. case y of
};
struct T1: public sum_type {             // T1 = int Wrapped
    int data;
    T1(int d): data(d) { }
    int isFirst() const;                 // case of T1 type action
};
struct T2: public sum_type {             // T2 = int2int
    int2int data;
    T2(int2int d): data(d) { }
    int isFirst() const;                 // case of T2 type action
};
struct T3: public sum_type {             // T3 = int x int
    pair data;
    T3(pair d): data(d) { }
    int isFirst() const;                 // case of T3 type action
};
```

$\Lambda^{\rightarrow}_{rr}$: Sum Type: Example 3: Using Inheritance & Dynamic Dispatch in C++

$Int + (Int \rightarrow Int) + Int \times Int$: $M_1 \equiv in_1^{Int, Int \rightarrow Int, Int \times Int}(\underline{25})$, $M_2 \equiv in_2^{Int, Int \rightarrow Int, Int \times Int}(\underline{succ})$, $M_3 \equiv in_3^{Int, Int \rightarrow Int, Int \times Int}(< \underline{12}, \underline{21} >)$

$isFirst = \lambda(y : Int + (Int \rightarrow Int) + Int \times Int).\ case\ y\ of\ x : Int\ then\ \underline{plus}\ x\ \underline{1}\ ||$
$f : Int \rightarrow Int\ then\ f\ \underline{7}\ ||\ t : Int \times Int\ then\ \underline{plus}\ proj_1(t)\ proj_2(t)$

```
#include <iostream>
// typedef int (*int2int)(int); // Int -> Int
// struct pair;                 // Int x Int
// int succ(int n);             // succ: Int->Int
// struct sum_type;             // Int + (Int -> Int) + Int x Int
// struct T1: public sum_type;  // T1 = int Wrapped
// struct T2: public sum_type;  // T2 = int2int
// struct T3: public sum_type;  // T3 = int x int

int T1::isFirst() const         // case of T1 type action code
{ return data + 1; }            // x: Int then x + 1
int T2::isFirst() const         // case of T2 type action code
{ return data(7); }             // f: Int -> Int then f 7
int T3::isFirst() const         // case of T3 type action code
{ return data.i1 + data.i2; }   // t: Int x Int then proj1(t) + proj2(t)
int main() {
    const sum_type&  M1 = T1(25);            // M1 = in1(25)
    const sum_type&  M2 = T2(succ);          // M2 = in2(succ)
    const sum_type&  M3 = T3(pair(12,21));   // M3 = in3(pair(12,21))

    std::cout << M1.isFirst(); // 48
    std::cout << M2.isFirst(); // 8
    std::cout << M3.isFirst(); // 33
    return 0;
}
```

$\Lambda_{rr}^{\rightarrow}$: Sum Type: Example 4

- $M_1 \equiv in_1^{Integer, Integer+Integer}(\underline{10})$
- $M_2 \equiv in_2^{Integer, Integer+Integer}(in_1^{Integer, Integer}(\underline{12}))$
- $M_3 \equiv in_2^{Integer, Integer+Integer}(in_2^{Integer, Integer}(\underline{14}))$

$\Lambda^{\rightarrow}_{rr}$: Sum Type: Example 4

- $M_1 \equiv in_1^{Integer, Integer+Integer}(\underline{10})$
- $M_2 \equiv in_2^{Integer, Integer+Integer}(in_1^{Integer, Integer}(\underline{12}))$
- $M_3 \equiv in_2^{Integer, Integer+Integer}(in_2^{Integer, Integer}(\underline{14}))$
- Type $Integer + (Integer + Integer)$

$$
\begin{aligned}
isFirst = \lambda(y : Integer + (Integer + Integer)).\ & case\ y\ of \\
& x : Integer\ then\ \underline{plus}\ x\ \underline{1} \ || \\
& s : Integer + Integer\ then \\
& \quad \lambda(z : Integer + Integer).\ case\ z\ of \\
& \quad\quad a : Integer\ then\ \underline{plus}\ a\ \underline{2} \ || \\
& \quad\quad b : Integer\ then\ \underline{plus}\ b\ \underline{3}
\end{aligned}
$$

$\Lambda^{\rightarrow}_{rr}$: Sum Type: Example 4

- $M_1 \equiv in_1^{Integer, Integer+Integer}(\underline{10})$
- $M_2 \equiv in_2^{Integer, Integer+Integer}(in_1^{Integer, Integer}(\underline{12}))$
- $M_3 \equiv in_2^{Integer, Integer+Integer}(in_2^{Integer, Integer}(\underline{14}))$
- Type $Integer + (Integer + Integer)$

$$
\begin{aligned}
isFirst = \lambda(y : Integer + (Integer + Integer)).\ & case\ y\ of \\
& x : Integer\ then\ \underline{plus}\ x\ \underline{1}\ || \\
& s : Integer + Integer\ then \\
& \quad \lambda(z : Integer + Integer).\ case\ z\ of \\
& \quad\quad a : Integer\ then\ \underline{plus}\ a\ \underline{2}\ || \\
& \quad\quad b : Integer\ then\ \underline{plus}\ b\ \underline{3}
\end{aligned}
$$

- Hence,
  - $isFirst\ M_1 = \underline{11}$
  - $isFirst\ M_2 = \underline{14}$
  - $isFirst\ M_3 = \underline{17}$

$\Lambda^{\rightarrow}_{rr}$: Sum Type: Example 4: Using union in C

$Int{+}(Int{+}Int) : M_1 \equiv in_1^{Int, Int+Int}(\underline{10}),\ M_2 \equiv in_2^{Int, Int+Int}(in_1^{Int, Int}(\underline{12})),\ M_3 \equiv in_2^{Int, Int+Int}(in_2^{Int, Int}(\underline{14}))$

$isFirst = \lambda(y : Int + (Int + Int)).\ case\ y\ of\ x : Int\ then\ \underline{plus}\ x\ \underline{1}\ ||\ s : Int + Int\ then$
$\lambda(z : Int + Int).\ case\ z\ of\ a : Int\ then\ \underline{plus}\ a\ \underline{2}\ ||\ b : Int\ then\ \underline{plus}\ b\ \underline{3}$

```
#include <iostream>

enum tag_type {field1 = 0, field2 = 1};
union union_type_inner {
    int i1; // field1, Int
    int i2; // field2, Int
    union_type_inner(enum tag_type tag, int i)
    { (tag == field1)? i1 = i: i2 = i; }
};
struct sum_type_inner {          // Int + Int
    enum tag_type tag;           // tag to remember injection component
    union union_type_inner u;
    sum_type_inner(enum tag_type t, int i): tag(t), u(tag, i) { } // in1, in2
};
union union_type {
    int i1;                  // field1, Int
    sum_type_inner i2;   // field2, Int + Int
    union_type(int i): i1(i) { }
    union_type(sum_type_inner i): i2(i) { }
};
struct sum_type {        // Int + (Int + Int)
    enum tag_type tag;   // tag to remember injection component
    union union_type u;
    sum_type(int i): tag(field1), u(i) { }
    sum_type(sum_type_inner i): tag(field2), u(i) { }
};
```

$\Lambda_{rr}^{\rightarrow}$: Sum Type: Example 4: Using union in C

$Int{+}(Int{+}Int) : M_1 \equiv in_1^{Int,Int+Int}(\underline{10}),\ M_2 \equiv in_2^{Int,Int+Int}(in_1^{Int,Int}(\underline{12})),\ M_3 \equiv in_2^{Int,Int+Int}(in_2^{Int,Int}(\underline{14}))$

$isFirst = \lambda(y : Int + (Int + Int)).\ case\ y\ of\ x : Int\ then\ \underline{plus}\ x\ \underline{1} \mid\mid s : Int + Int\ then$
$\lambda(z : Int + Int).\ case\ z\ of\ a : Int\ then\ \underline{plus}\ a\ \underline{2} \mid\mid b : Int\ then\ \underline{plus}\ b\ \underline{3}$

```
// enum tag_type {field1 = 0, field2 = 1};
// union union_type_inner;
// struct sum_type_inner;   // Int + Int
// union union_type;
// struct sum_type;         // Int + (Int + Int)

int isFirst(struct sum_type u) {                        // y: Int + (Int + Int)
    switch (u.tag) {                                    // case y of
        case field1: return u.u.i1 + 1;                 // x: Int then x + 1 ||
        case field2:                                    // s: Int + Int, z: Int + Int
            switch (u.u.i2.tag) {                       // case z of
                case field1: return u.u.i2.u.i1 + 2;    // a: Int then a + 2 ||
                case field2: return u.u.i2.u.i2 + 3;    // b: Int then b + 2
            }
    }
}
int main() {
    struct sum_type M1 = 10;                       // M1 = in1(10)
    struct sum_type M2 = { { field1, 12 } };       // M2 = in2(in1(12))
    struct sum_type M3 = { { field2, 14 } };       // M3 = in2(in2(14))

    std::cout << isFirst(M1); // 11
    std::cout << isFirst(M2); // 12
    std::cout << isFirst(M3); // 14
    return 0;
}
```

$\Lambda^{\rightarrow}_{rr}$: Sum Type: Example 4: Using Inheritance & Dynamic Dispatch in C++

$Int+(Int+Int) : M_1 \equiv in_1^{Int,Int+Int}(\underline{10}),\ M_2 \equiv in_2^{Int,Int+Int}(in_1^{Int,Int}(\underline{12})),\ M_3 \equiv in_2^{Int,Int+Int}(in_2^{Int,Int}(\underline{14}))$

$isFirst = \lambda(y : Int + (Int + Int)).\ case\ y\ of\ x : Int\ then\ \underline{plus}\ x\ \underline{1}\ ||\ s : Int + Int\ then$
$\lambda(z : Int + Int).\ case\ z\ of\ a : Int\ then\ \underline{plus}\ a\ \underline{2}\ ||\ b : Int\ then\ \underline{plus}\ b\ \underline{3}$

```
#include <iostream>
struct sum_type {                              // Int + (Int + Int)
    virtual int myCase() const = 0;            // myCase type switch
                                               // y: Int + (Int + Int). case y of
};
struct sum_type_inner: public sum_type {       // Int + Int
    virtual int myCase() const = 0;            // myCase type switch
                                               // s: Int + Int. case s of
};
struct T1: public sum_type {                   // T1 = int Wrapped
    int data; T1(int d): data(d) { }
    int myCase() const;                        // case of T1 type action
};
struct T2: public sum_type_inner {             // T2 = int Wrapped
    int data; T2(int d): data(d) { }
    int myCase() const;                        // case of T2 type action
};
struct T3: public sum_type_inner {             // T3 = int Wrapped
    int data; T3(int d): data(d) { }
    int myCase() const;                        // case of T2 type action
};
int T1::myCase() const                         // case of T1 type action code
{ return data + 1; }                           // x: Int then x + 1
int T2::myCase() const                         // case of T2 type action code
{ return data + 2; }                           // a: Int then a + 2
int T3::myCase() const                         // case of T2 type action code
{ return data + 3; }                           // b: Int then b + 3
```

$\Lambda^{\rightarrow}_{rr}$: Sum Type: Example 4: Using Inheritance & Dynamic Dispatch in C++

$Int{+}(Int{+}Int) : M_1 \equiv in_1^{Int, Int+Int}(\underline{10}),\ M_2 \equiv in_2^{Int, Int+Int}(in_1^{Int, Int}(\underline{12})),\ M_3 \equiv in_2^{Int, Int+Int}(in_2^{Int, Int}(\underline{14}))$

$isFirst = \lambda(y : Int + (Int + Int)).\ case\ y\ of\ x : Int\ then\ \underline{plus}\ x\ \underline{1}\ ||\ s : Int + Int\ then$
$\lambda(z : Int + Int).\ case\ z\ of\ a : Int\ then\ \underline{plus}\ a\ \underline{2}\ ||\ b : Int\ then\ \underline{plus}\ b\ \underline{3}$

```
// struct sum_type;                               // Int + (Int + Int)
// struct sum_type_inner: public sum_type;        // Int + Int
// struct T1: public sum_type;                    // T1 = int Wrapped
// struct T2: public sum_type_inner;              // T2 = int Wrapped
// struct T3: public sum_type_inner;              // T3 = int Wrapped

int main() {
    const sum_type& M1 = T1(10);                   // M1 = in1(10)
    const sum_type_inner& M2_inner = T2(12);       // M2_inner = in1(12)
    const sum_type_inner& M3_inner = T3(14);       // M3_inner = in2(14)
    const sum_type& M2 = M2_inner;                 // M2 = in2(M2_inner) = in2(in1(12))
    const sum_type& M3 = M3_inner;                 // M3 = in2(M3_inner) = in2(in2(14))

    // Since Int + (Int + Int) = Int + Int + Int, we can inject the components of
    // the inner sum type directly too. So commenting the above declarations of
    // M2_inner, M2, M3_inner & M3 and un-commenting the following declarations of
    // M2 & M3 with direct injection in the hierarchy will also work

    // const sum_type& M2 = T2(12);                // M2 = in2(12)
    // const sum_type& M3 = T3(14);                // M3 = in3(14)

    std::cout << M1.myCase(); // 11
    std::cout << M2.myCase(); // 14
    std::cout << M3.myCase(); // 17
    return 0;
}
```

$\Lambda_{rr}^{\rightarrow}$: Sum Type: Example 4: Using Inheritance & Dynamic Dispatch in C++

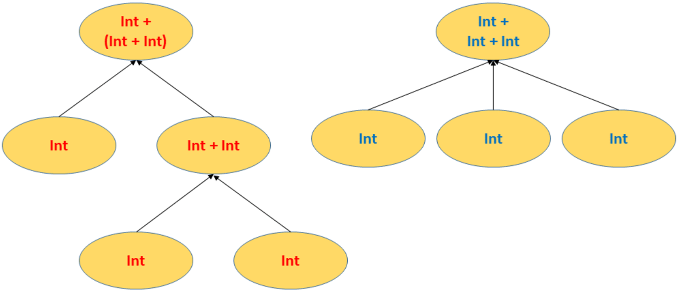
Since $Int + (Int + Int) \equiv Int + Int + Int$, the following

$Int{+}(Int{+}Int) : M_1 \equiv in_1^{Int,Int+Int}(\underline{10}),\ M_2 \equiv in_2^{Int,Int+Int}(in_1^{Int,Int}(\underline{12})),\ M_3 \equiv in_2^{Int,Int+Int}(in_2^{Int,Int}(\underline{14}))$
$isFirst = \lambda(y : Int + (Int + Int)).\ case\ y\ of\ x : Int\ then\ \underline{plus}\ x\ \underline{1}\ ||\ s : Int + Int\ then$
$\lambda(z : Int + Int).\ case\ z\ of\ a : Int\ then\ \underline{plus}\ a\ \underline{2}\ ||\ b : Int\ then\ \underline{plus}\ b\ \underline{3}$

can be simplified to:

$Int + Int + Int : M_1 \equiv in_1^{Int,Int,Int}(\underline{10}),\ M_2 \equiv in_2^{Int,Int,Int}(\underline{12}),\ M_3 \equiv in_3^{Int,Int,Int}(\underline{14})$
$isFirst = \lambda(y : Int + Int + Int).\ case\ y\ of\ x : Int\ then\ \underline{plus}\ x\ \underline{1}\ ||$
$a : Int\ then\ \underline{plus}\ a\ \underline{2}\ ||\ b : Int\ then\ \underline{plus}\ b\ \underline{3}$

This changes the type by construction, but gives a simpler equivalent type for injection and case. We illustrate with a flattened hierarchy below:



$\Lambda^{\rightarrow}_{rr}$: Sum Type: Example 4: Using Inheritance & Dynamic Dispatch in C++

$Int + Int + Int : M_1 \equiv in_1^{Int,Int,Int}(\underline{10}),\ M_2 \equiv in_2^{Int,Int,Int}(\underline{12}),\ M_3 \equiv in_3^{Int,Int,Int}(\underline{14})$

$isFirst = \lambda(y : Int + Int + Int).\ case\ y\ of\ x : Int\ then\ \underline{plus}\ x\ \underline{1}\ ||$
$a : Int\ then\ \underline{plus}\ a\ \underline{2}\ ||\ b : Int\ then\ \underline{plus}\ b\ \underline{3}$

```
#include <iostream>
struct sum_type {                        // Int + Int + Int
    virtual int myCase() const = 0;      // myCase type switch
                                         // y: Int + Int + Int. case y of
};
struct T1: public sum_type {             // T1 = int Wrapped
    int data;
    T1(int d): data(d) { }
    int myCase() const;                  // case of T1 type action
};
struct T2: public sum_type {             // T2 = int Wrapped
    int data;
    T2(int d): data(d) { }
    int myCase() const;                  // case of T2 type action
};
struct T3: public sum_type {             // T3 = int Wrapped
    int data;
    T3(int d): data(d) { }
    int myCase() const;                  // case of T3 type action
};
```

$\Lambda^{\rightarrow}_{rr}$: Sum Type: Example 4: Using Inheritance & Dynamic Dispatch in C++

$Int + Int + Int : M_1 \equiv in_1^{Int,Int,Int}(\underline{10}),\ M_2 \equiv in_2^{Int,Int,Int}(\underline{12}),\ M_3 \equiv in_3^{Int,Int,Int}(\underline{14})$
$isFirst = \lambda(y : Int + Int + Int).\ case\ y\ of\ x : Int\ then\ \underline{plus}\ x\ \underline{1}\ ||$
$a : Int\ then\ \underline{plus}\ a\ \underline{2}\ ||\ b : Int\ then\ \underline{plus}\ b\ \underline{3}$

```
// struct sum_type;                 // Int + Int + Int
// struct T1: public sum_type;      // T1 = int Wrappe
// struct T2: public sum_type;      // T2 = int Wrapped
// struct T3: public sum_type;      // T3 = int Wrapped

int T1::myCase() const              // case of T1 type action code
{   return data + 1; }              // x: Int then x + 1
int T2::myCase() const              // case of T2 type action code
{   return data + 2; }              // a: Int then a + 2
int T3::myCase() const              // case of T2 type action code
{   return data + 3; }              // b: Int then b + 3

int main() {
    const sum_type& M1 = T1(10);    // M1 = in1(10)
    const sum_type& M2 = T2(12);    // M2 = in2(12)
    const sum_type& M3 = T3(14);    // M3 = in2(14)

    std::cout << M1.myCase(); // 11
    std::cout << M2.myCase(); // 14
    std::cout << M3.myCase(); // 17
    return 0;
}
```

$\Lambda_{rr}^{\rightarrow}$: Reference Type

- Reference types represent updatable variables
- If M has type

$$Ref \;\; T$$

  we can think of it as denoting a location which can hold a value of type T
- The expression

$$val \;\; M$$

  will denote the value stored at that location

$\Lambda^{\rightarrow}_{rr}$: Reference Type: Remarks

- In most procedural languages, programmers are not required to distinguish between variables (representing locations) and the values they denote
- The compiler or interpreter automatically selects the appropriate attribute (location or l-value versus value or r-value) based on context without requiring the programmer to annotate the variable
  - In C, for
    $$x = y;$$
    x denotes l-value while y denotes r-value
  - In ML (which supports references)
    - Write $!x$ when the value stored in x is required, while x alone always denotes the location
  - In C
    - Write $\&x$ when the location of x is required in an r-value context, where x alone denotes the value stored in x

$\Lambda^{\rightarrow}_{rr}$: Reference Type: *null* Expression & Assignment

- The *null* expression is a special constant representing the null reference, a reference that does not point to anything
- Evaluating the expression

$$val\ \ null$$

will always result in an error

- If M is a reference, with type $Ref\ \ T$, and N has type T, then the expression

$$M := N$$

denotes the assignment of N to M – the value of N is stored in the location denoted by M

$\Lambda_{rr}^{\rightarrow}$: Array Type

- We ignore the array type because an array is as good as its index function (when the memory is not considered)
- For example,

```
int a[3] = {5, 3, 8};
```

may be modeled as an index function:

```
int aIndex(int i) {
    switch (i) {
        case 0: return 5;
        case 1: return 3;
        case 2: return 8;
    }
}
```

$\Lambda_{rr}^{\rightarrow}$: Array Type

- Of course, for making assignments to array elements, we need more tricks. In the context of `a[1] = 4;`, `aIndex()` changes from
  `((0, 5), (1, 3), (2, 8))`
  to
  `((0, 5), (1, 4), (2, 8))`
- So every assignment needs a higher order function
  `aAssign(aIndex(), indexToItem, value)`
  that returns function `aIndex()`
- Also, we need to support subtype (subrange of Integer) for the index type
- We need further work (including modeling memory) to ensure contiguous locations for an array

Type Expression

- Let $\mathcal{L}$ be an infinite collection of labels, and let $\mathcal{TC}$ be a collection of type constants. The type expressions of $\Lambda_{rr}^{\rightarrow}$ are given by the following grammar:

$$
\begin{array}{rl}
T \in Type ::= & C \mid \\
& Void \mid \\
& T_1 \rightarrow T_2 \mid \\
& T_1 \times \cdots \times T_n \mid \\
& \{| l_1 : T_1, l_2 : T_2, \cdots, l_n : T_n |\} \mid \\
& T_1 + \cdots + T_n \mid \\
& Ref\ \ T \mid \\
& Command
\end{array}
$$

where $l_i \in \mathcal{L}$, and, as before, $C \in \mathcal{TC}$ represents type constants (like *Integer*, *Double*, etc.)

Type Expression: Types

- *Void* Type
  - The type *Void* is the type of zero tuples used as the return type of commands or statements and as the parameter type for parameterless functions
  - It has only one (trivial) value, $\langle\rangle$
- Function Types
- Product (tuple) Types
- Record Types
- Sum (or disjoint union) Types
- Reference types (the types of variables)
- *Command* Type
  - Represents the type of statements, expressions like assignments that are evaluated simply for their side effects

Pre-Expression

- The collection of **pre-expressions** of $\Lambda^{\rightarrow}_{rr}$, $\mathcal{RLCE}$ are:

$$
\begin{aligned}
M \in \mathcal{RLCE} ::= {} & c \mid x \mid \langle\rangle \mid \lambda(x:T).\ M \mid M\ N \mid (\ M\ ) \mid \\
& \langle M_1, \cdots, M_n \rangle \mid proj_i(M) \mid \\
& \{| l_1 : T_1 := M_1, \cdots, l_n : T_n := M_n |\} \mid M.l_i \mid \\
& in_i^{T_1, \cdots, T_n}(M) \mid \\
& case\ M\ of\ x_1 : T_1\ then\ E_1 \mid\mid \cdots \mid\mid x_n : T_n\ then\ E_n \mid \\
& ref\ M \mid null \mid val\ M \mid \\
& if\ B\ then\ \{\ M\ \}\ else\ \{\ N\ \} \mid \\
& nop \mid N := M \mid M;\ N
\end{aligned}
$$

where $x \in \mathcal{EI}$, $c \in \mathcal{EC}$, and $l_i \in \mathcal{L}$

Pre-Expression: Command / Statements

- Statements of a typical programming language are added here as expressions of *Command* type:
  - *if* B *then* $\{\ M\ \}$ *else* $\{\ N\ \}$ is conditional statement
  - *nop*, a constant, represents a statement that has no effect
  - $N := M$ represents an assignment statement
  - $M;\ N$ indicates the sequencing of the two statements. Do the first statement for the side effect and then return the value of the second
- *We ignore various loop constructs – we shall add recursion later for completing the expressive power – hence, loops are not needed*

Type-checking Rules

Identifier	$\dfrac{}{\mathcal{E} \cup \{x{:}T\} \vdash x{:}T}$
Constant	$\dfrac{}{\mathcal{E} \vdash c \in C}$
Void	$\dfrac{}{\mathcal{E} \vdash \langle\rangle{:}Void}$
Function	$\dfrac{\mathcal{E} \cup \{x{:}S\} \vdash M{:}T}{\mathcal{E} \vdash \lambda(x{:}S).\ M{:}\ S \rightarrow T}$
Application	$\dfrac{\mathcal{E} \vdash M{:}S \rightarrow T,\ \mathcal{E} \vdash N{:}S}{\mathcal{E} \vdash M\ \ N{:}T}$
Paren	$\dfrac{\mathcal{E} \vdash M{:}T}{\mathcal{E} \vdash (M){:}T}$

Type-checking Rules

Tuple
$$\frac{\mathcal{E} \vdash M_i : T_i,\ \forall i,\ 1 \le i \le n}{\mathcal{E} \vdash \langle M_1, \cdots, M_n \rangle : T_1 \times \cdots \times T_n}$$

Projection
$$\frac{\mathcal{E} \vdash M : T_1 \times \cdots \times T_n}{\mathcal{E} \vdash proj_i(M) : T_i},\ \forall i,\ 1 \le i \le n$$

Record
$$\frac{\mathcal{E} \vdash M_i : T_i,\ \forall i,\ 1 \le i \le n}{\mathcal{E} \vdash \{| l_1 : T_1 := M_1, \cdots, l_n : T_n := M_n |\} : \{| l_1 : T_1, \cdots, l_n : T_n |\}}$$

Selection
$$\frac{\mathcal{E} \vdash M : \{| l_1 : T_1, \cdots, l_n : T_n |\}}{\mathcal{E} \vdash M.l_i : T_i},\ \forall i,\ 1 \le i \le n$$

Sum
$$\frac{\mathcal{E} \vdash M : T_i,\ \exists i,\ 1 \le i \le n}{\mathcal{E} \vdash in_i^{T_1, \cdots, T_n}(M) : T_1 + \cdots + T_n}$$

Case
$$\frac{\mathcal{E} \vdash M : T_1 + \cdots + T_n,\ \mathcal{E} \cup \{x_i : T_i\} \vdash E_i : U}{\mathcal{E} \vdash case\ \ M\ \ of\ \ x_1 : T_1\ \ then\ \ E_1\ \ || \cdots ||\ \ x_n : T_n\ \ then\ \ E_n : U},\ \forall i,\ 1 \le i \le n$$

The case expressions require that the types of the branches all be the same type. This way a result of the same type is returned no matter which branch is selected.

Type-checking Rules

Reference
$$\frac{\mathcal{E} \vdash M{:}T}{\mathcal{E} \vdash ref\ \ M{:}Ref\ \ T}$$

Null
$$\frac{}{\mathcal{E} \vdash null{:}Ref\ \ T}, \textit{ for any type } T$$

Value
$$\frac{\mathcal{E} \vdash M{:}Ref\ \ T}{\mathcal{E} \vdash val\ \ M{:}T}$$

No op
$$\frac{}{\mathcal{E} \vdash nop{:}Command}$$

Assignment
$$\frac{\mathcal{E} \vdash N{:}Ref\ \ T,\ \mathcal{E} \vdash M{:}T}{\mathcal{E} \vdash N{:=}M{:}Command}$$

Conditional
$$\frac{\mathcal{E} \vdash B{:}Boolean,\ \mathcal{E} \vdash M{:}T,\ \mathcal{E} \vdash N{:}T}{\mathcal{E} \vdash if\ \ B\ \ then\ \{\ M\ \}\ else\ \{\ N\ \}{:}T}$$

The if-then-else expressions require that the types of the branches all be the same type. This way a result of the same type is returned no matter which branch is selected.

Sequencing
$$\frac{\mathcal{E} \vdash M{:}S,\ \mathcal{E} \vdash N{:}T}{\mathcal{E} \vdash M;\ \ N{:}T}$$

Type-checking Rules: n-ary Functions

- We write

$$\lambda(id_1 : T_1, \cdots, id_n : T_n).\ M$$

as an abbreviation for

$$\lambda(arg : T_1 \times \cdots \times T_n).[proj_i(arg)/id_i]_{i=1,\cdots,n}M$$

- Thus an n-ary function is an abbreviation for a function of a single argument that takes an n-tuple
- When expanded, each of the individual parameters is replaced by an appropriate projection from the n-tuple
- The derived typing rule for n-ary functions is:

$$n\text{-ary function} \quad \frac{\mathcal{E} \cup \{id_1{:}T_1,\cdots,id_n{:}T_n\} \vdash M{:}U}{\mathcal{E} \vdash \lambda(\{id_1{:}T_1,\cdots,id_n{:}T_n\}).\ M{:}T_1{\times}\cdots{\times}T_n{\rightarrow}U}$$

Type-checking Rules: *let* expressions

- We write

$$let\ x : T \ = \ M\ in\ N\ end$$

as an abbreviation for

$$(\lambda(x : T).\ N)\ M$$

- Thus, introducing an identifier for an expression is modeled by writing a function with that identifier as the parameter, and then applying the function to the intended value for the identifier
- The derived typing rule for *let* expressions is:

$$\textit{let}\text{ expression} \quad \frac{\mathcal{E} \cup \{x{:}T\} \vdash N{:}S,\ \mathcal{E} \vdash \{M{:}T\}}{\mathcal{E} \vdash let\ \ x{:}T \ = \ M\ \ in\ \ N\ \ end{:}S}$$