Last updated: 2025-07-25 19:09:00-07:00

1 Abstract Syntax

The abstract syntax of LAMP expressions is described by the following grammar. Syntactic sugars are colored in blue.

$$
\begin{array}{rcll}
\mathit{prog} & ::= & (\mathit{def} \mid \mathit{eqn})^* \ \mathit{main} & \text{program} \\
\mathit{def} & ::= & \mathsf{def}\ X : t = e & \text{abbreviation} \\
\mathit{eqn} & ::= & \mathsf{type}\ X = t & \text{type equation} \\
\mathit{main} & ::= & \mathsf{main} : t = e & \text{main expression} \\
e \in \mathit{expr} & ::= & n & \text{natural number} \\
& \mid & e_1 \oplus e_2 & \text{arithmetic} \\
& \mid & e_1 == e_2 & \text{equality (between nats only)} \\
& \mid & b & \text{boolean} \\
& \mid & \mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 & \text{if-then-else} \\
& \mid & x & \text{variable} \\
& \mid & \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 & \text{let-binding} \\
& \mid & \lambda x.\ e & \text{lambda function} \\
& \mid & e_1\ e_2 & \text{application} \\
& \mid & (e_1, \ldots, e_n) & \text{packing (an } n\text{-tuple, } n \geq 0) \\
& \mid & \mathsf{let}\ (x_1, \ldots, x_n) = e_1\, \mathsf{in}\, e_2 & \text{unpacking (an } n\text{-tuple, } n \geq 0) \\
& \mid & X & \text{abbreviations (global variable)} \\
& \mid & e : t & \text{type annotation} \\
& \mid & \underline{l}\ e & \text{injection} \\
& \mid & \mathsf{switch}\ e\ \{(\underline{l}\ x : e)*\} & \text{switch} \\
& \mid & \mathsf{match}\ e\ \{(\mathit{branch})*\} & \text{pattern matching} \\
\mathit{branch} & ::= & p : e & \text{branch} \\
p \in \mathit{pattern} & ::= & \_ & \text{wildcard} \\
& \mid & x & \text{variable pattern} \\
& \mid & \underline{l}\ p & \text{injection pattern} \\
& \mid & (p_1, \ldots, p_n) & \text{tuple pattern } (n \geq 0) \\
n \in \mathbb{N} & = & \{0, 1, 2, 3, \ldots\} & \\
b \in \mathbb{B} & = & \{\mathsf{True}, \mathsf{False}\} & \\
\oplus & \in & \{+, -, *\} & \\
l & \in & \mathsf{Labels} &
\end{array}
$$

Note that the empty tuple () is allowed (you can take n to be 0 in an n-tuple), and is called the *unit*.

We define *values* using the following grammar:

$$
\begin{array}{rcll}
v \in \mathit{value} & ::= & n \in \mathbb{N} & \text{natural number} \\
& \mid & b \in \mathbb{B} & \text{boolean} \\
& \mid & \lambda x.\ e & \text{lambda function} \\
& \mid & (v_1, \ldots, v_n) & \text{tuple of } n \text{ value } (n \geq 0) \\
& \mid & \underline{l}\ v & \text{injection value}
\end{array}
$$

We define *types* (abstract values) using the following grammar:

$$
\begin{array}{rcll}
t \in \mathit{type} & ::= & \mathsf{Nat} & \text{natural number type} \\
& \mid & \mathsf{Bool} & \text{boolean type} \\
& \mid & t_1 \to t_2 & \text{function type} \\
& \mid & (v_1, \ldots, v_n) & \text{product type } (n \geq 0) \\
& \mid & +\{\underline{l_1} : t_1,\ \underline{l_2} : t_2,\ \ldots\} & \text{sum type } (n \geq 0) \\
& \mid & X & \text{type name}
\end{array}
$$

2 Concrete Syntax

2.1 Associativity and Precedence

The operator associativity for LAMP is given by the following table. Higher up = higher precedence. For example, the

Operation	Associativity
Application	Left
$*$	Left
$+, -$	Left
$==$	Non-associative
$\to$	Right

expression $f\ x\ y$ will be parsed as $(f\ x)\ y$, and $f\ 1 + 2 * 3$ will be parsed as $(f\ 1) + (2 * 3)$. Just remember that function applications always bind more strongly than *anything else*.

2.2 Identifiers

As shown in the abstract syntax, LAMP has two kinds of identifiers:

1. A *variable* is like your usual, lexically-scoped variable. In the concrete syntax, the name of the variable must start with a lower case letter, followed by any number of digits, lower or upper case letters, or underscores "_". A variable can also be a single underscore.
2. An *abbreviation* must start with a capital letter, followed by any number of digits, lower- or upper-case letters, or underscores "_". The meaning of abbreviations is explained in the next subsection.

2.3 Lambdas

Since the greek letter λ is difficult to type in a text editor, we'll use the backslack symbol "\" in place of "λ" when writing LAMP program in concrete syntax. For example, the lambda function $\lambda x.\ \lambda y.\ x + y$ will be written as:

```
\x. \y. x + y
```

2.4 Sum Injections

To inject an expression e into a sum type with label l, we use the notation "$\underline{l}\ e$" in this manual, but in the concrete syntax, we prefix the label with a single quote, like this:

```
def SafeDiv: Nat -> Nat -> +{'divByZero: (), 'success: Nat} = \x. \y.
    if y == 0 then 'divByZero
    else 'success (x / y)
```

Note that if the injected expression is a 0-tuple (aka a unit), then the unit can be omitted, and you only need to write the label, as the first branch of the above example shows.

2.5 Pattern Matching

The concrete syntax for pattern matching is the same as the abstract syntax. Here're a couple of syntactic sugars that you may found useful.

Injection patterns of the form `'label ()` can be simply written as `'label` in the concrete syntax.

Functions that immediately pattern match on their argument can be written as follows:

```
\{ ... }
```

which is desugared into:

```
\x. match x {
    ...
}
```

Tuple unpacking (the standard elimination form for products) is desugared into pattern matching as follows:

```
let (x1, x2, ..., xn) = e1 in e2
```

is desugared into

```
match e1 {
    (x1, x2, ..., xn): e2
}
```

where the pattern $(x_1, x_2, \ldots, x_n)$ is a tuple pattern that matches a tuple value of size n.

Switch (the standard elimination form for sums) is desugared into pattern matching as follows:

```
switch e {
    'l1 x: e1,
    'l2 x: e2,
    ...,
    'ln x: en
}
```

is desugared into

```
match e {
    'l1 x: e1,
    'l2 x: e2,
    ...,
    'ln x: en
}
```

where each `li x: ei` is a branch that matches an injection value with label `li` and binds the inner value to the variable `x`.

2.6 Programs

A Lamp program is a (possibly empty) list of abbreviations or type equations followed by the entry point.

An abbreviation has the form

```
def X : t = e
```

where X is the name of the abbreviation, t is a type, and e is an expression. Abbreviations are implicitly unfolded during evaluation, and can be (mutually) recursive.

A type equation has the form

```
type X = t
```

where X is a type name (a global variable), and t is a type. Type equations are used to define (mutually) recursive types. For example, the following type equation defines types for (user-defined) natural numbers and linked lists:

```
type MyNat = +{
    'zero: (),
    'succ: MyNat
}
type List = +{
    'nil: (),
    'cons: MyNat, List
}
```

Note that type equations do not have to be recursive, as the following example shows:

```
type MyBool = +{
    'true: (),
    'false: ()
}
```

The entry point should appear after all abbreviations and must have the form:

```
main : t = e
```

where `main` is a reserved keyword and e is an expression.

For example, the following is a valid LAMP program:

```
def FibHelper: (Nat, Nat) -> Nat -> Nat = \p. \n.
    let (x, y) = p in
    if n == 0 then x
    else FibHelper (y, x+y) (n-1)

def Fib: Nat -> Nat = FibHelper (0, 1)

type MyBool = +{
    'true: (),
    'false: ()
}

def Negate: MyBool -> MyBool = \{
    'true: 'false,
    'false: 'true
}

main: Nat = Fib 10
```

If we remove all syntactic sugars, the above program becomes:

```
def FibHelper: (Nat, Nat) -> Nat -> Nat =
    \p. \n. match p {
        (x, y):
            if n == 0 then x
            else FibHelper (y, x+y) (n-1)
    }

def Fib: Nat -> Nat = FibHelper (0, 1)

type MyBool = +{
    'true: (),
    'false: ()
}

def Negate: MyBool -> MyBool = \x. match x {
    'true (): 'false (),
    'false (): 'true ()
}

main: Nat = Fib 10
```

Make sure you understand how each syntactic sugar in the former program is desugared into the latter program.

2.7 Operational Semantics (Dynamics)

Judgment: $\boxed{e \Rightarrow v}$

$\sigma \in \textit{substitution}$	$::=$	$\cdot$	empty substitution
	$\mid$	$\sigma,\ v/x$	substitution with x replaced with v

$$\frac{}{n \Rightarrow n}\ \textsc{Nat} \qquad \frac{e_1 \Rightarrow n_1 \quad e_2 \Rightarrow n_2 \quad (\oplus \in \{+, -, *\}) \quad (n_1 \oplus n_2 = n_3)}{e_1 \oplus e_2 \Rightarrow n_3}\ \textsc{Arith}$$

$$\frac{e_1 \Rightarrow n_1 \quad e_2 \Rightarrow n_2 \quad (n_1 = n_2)}{e_1 == e_2 \Rightarrow \mathsf{True}}\ \textsc{EqTrue} \qquad \frac{e_1 \Rightarrow n_1 \quad e_2 \Rightarrow n_2 \quad (n_1 \neq n_2)}{e_1 == e_2 \Rightarrow \mathsf{False}}\ \textsc{EqFalse}$$

$$\frac{}{b \Rightarrow b}\ \textsc{Bool} \qquad \frac{e_1 \Rightarrow \mathsf{True} \quad e_2 \Rightarrow v}{\mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 \Rightarrow v}\ \textsc{IfTrue} \qquad \frac{e_1 \Rightarrow \mathsf{False} \quad e_3 \Rightarrow v}{\mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 \Rightarrow v}\ \textsc{IfFalse}$$

(No rule for variables) $$\frac{(\Delta(X) = e) \quad e \Rightarrow v}{X \Rightarrow v}\ \textsc{Abbrev} \qquad \frac{e_1 \Rightarrow v_1 \quad ([v_1/x]e_2 = e_2') \quad e_2' \Rightarrow v_2}{\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \Rightarrow v_2}\ \textsc{Let}$$

$$\frac{}{\lambda x.\ e \Rightarrow \lambda x.\ e}\ \textsc{Lambda} \qquad \frac{e_1 \Rightarrow \lambda x.\ e \quad e_2 \Rightarrow v \quad ([v/x]e = e') \quad e' \Rightarrow v'}{e_1\ e_2 \Rightarrow v'}\ \textsc{App}$$

$$\frac{(\text{for } 1 \leq i \leq n)\ e_i \Rightarrow v_i}{(e_1, \ldots, e_n) \Rightarrow (v_1, \ldots, v_n)}\ \textsc{Pack} \qquad \frac{e \Rightarrow v}{\underline{l}\ e \Rightarrow \underline{l}\ v}\ \textsc{Inj}$$

$$\frac{e \Rightarrow v \quad \text{least } i \text{ s.t. } v \bowtie p_i \hookrightarrow \sigma \quad ([\sigma]e_i = e_i') \quad e_i' \Rightarrow v'}{\mathsf{match}\ e\ \{p_1 : e_1, p_2 : e_2, \ldots\} \Rightarrow v'}\ \textsc{Match}$$

Judgment: $\boxed{v \bowtie p \hookrightarrow \sigma}$

$\sigma \in \textit{substitution}$	$::=$	$\cdot$	empty substitution
	$\mid$	$\sigma,\ v/x$	substitution with x replaced with v

$$\frac{}{v \bowtie \_ \hookrightarrow \cdot}\ \textsc{Match-Wildcard} \qquad \frac{}{v \bowtie x \hookrightarrow v/x}\ \textsc{Match-Var} \qquad \frac{v \bowtie p \hookrightarrow \sigma}{\underline{l}\ v \bowtie \underline{l}\ p \hookrightarrow \sigma}\ \textsc{Match-Inj}$$

$$\frac{(\text{for } 1 \leq i \leq n)\ v_i \bowtie p_i \hookrightarrow \sigma_1 \quad \sigma = \sigma_1, \sigma_2, \ldots, \sigma_n}{(v_1, \ldots, v_n) \bowtie (p_1, \ldots, p_n) \hookrightarrow \sigma}\ \textsc{Match-Tuple}$$

2.8 Type System (Statics)

Recall that Δ holds a list of all abbreviations defined in a LAMP program. Since every abbreviation has a declared type, we overload the notation $\Delta(X) = t$ to denote that the declared type of abbreviation X is t, i.e., the program contains a definition like

```
def X : t = ...
```

Let ξ be the list of type equations defined in a LAMP program. We use the notation $\xi(X) = t$ to denote that the type equation for X is t, i.e., the program contains a definition like

```
type X = t
```

The type system is defined by the judgment $\boxed{\Gamma \vdash e : t}$, where e is an expression, t is the type of e, and Γ is a list of pairs of variable and type:

$$
\begin{array}{rcll}
t \in \mathit{type} & ::= & \mathsf{Nat} & \text{natural number type} \\
& | & \mathsf{Bool} & \text{boolean type} \\
& | & t_1 \to t_2 & \text{function type} \\
& | & (t_1, \ldots, t_n) & \text{product type } (n \geq 0)
\end{array}
\qquad
\begin{array}{rcll}
\Gamma & ::= & \cdot & \text{empty} \\
& | & \Gamma, x : t & \text{binding}
\end{array}
$$

The typing judgment is defined by the following abstract semantics rules (aka typing rules):

$$
\frac{}{\Gamma, x : t \vdash x : t}\ \text{T-ID}
\qquad
\frac{\Gamma \vdash x : t_1 \qquad (x \neq y)}{\Gamma, y : t \vdash x_2 : t_1}\ \text{T-WEAKEN}
$$

$$
\frac{}{\Gamma \vdash n : \mathsf{Nat}}\ \text{T-NAT}
\qquad
\frac{\Gamma \vdash e_1 : \mathsf{Nat} \qquad \Gamma \vdash e_2 : \mathsf{Nat} \qquad (\oplus \in \{+, -, *\})}{\Gamma \vdash e_1 \oplus e_2 : \mathsf{Nat}}\ \text{T-ARITH}
\qquad
\frac{\Gamma \vdash e_1 : \mathsf{Nat} \qquad \Gamma \vdash e_2 : \mathsf{Nat}}{\Gamma \vdash e_1 == e_2 : \mathsf{Bool}}\ \text{T-EQ}
$$

$$
\frac{}{\Gamma \vdash b : \mathsf{Bool}}\ \text{T-BOOL}
\qquad
\frac{\Gamma \vdash e_1 : \mathsf{Bool} \qquad \Gamma \vdash e_2 : t_2 \qquad \Gamma \vdash e_3 : t_3 \qquad (t_2 = t_3)}{\Gamma \vdash \mathsf{if}\, e_1 \,\mathsf{then}\, e_2 \,\mathsf{else}\, e_3 : t_2}\ \text{T-IF}
$$

$$
\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma, x : t_1 \vdash e_2 : t_2}{\Gamma \vdash \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 : t_2}\ \text{T-LET}
\qquad
\frac{\Gamma, x : t_i \vdash e : t_o}{\Gamma \vdash \lambda x.\ e : t_i \to t_o}\ \text{T-LAMBDA}
\qquad
\frac{\Gamma \vdash e_1 : t_i \to t_o \qquad \Gamma \vdash e_2 : t_i}{\Gamma \vdash e_1\ e_2 : t_o}\ \text{T-APP}
$$

$$
\frac{(\text{for } 1 \leq i \leq n)\ \Gamma \vdash e_i : t_i}{\Gamma \vdash (e_1, \ldots, e_n) : (t_1, \ldots, t_n)}\ \text{T-PACK}
\qquad
\frac{(j \in L) \qquad \Gamma \vdash e : t_j}{\Gamma \vdash \underline{j}\ e : +\{\underline{l} : t_l\}_{l \in L}}\ \text{T-INJ}
$$

$$
\frac{???}{\Gamma \vdash \mathsf{match}\ e\ \{p_1 : e_1,\ p_2 : e_2,\ \ldots\} : t}\ \text{T-MATCH}
$$

$$
\frac{(\Delta(X) = t)}{\Gamma \vdash X : t}\ \text{T-ABBREV}
\qquad
\frac{\Gamma \vdash e : t}{\Gamma \vdash (e : t) : t}\ \text{T-ANN}
$$

2.9 Bidirectional Type System

Type synthesis: $\boxed{\Gamma \vdash e \downarrow t}$ Type checking: $\boxed{\Gamma \vdash e \uparrow t}$

$$\frac{\Gamma \vdash e \downarrow t_2 \qquad (t_1 = t_2)}{\Gamma \vdash e \uparrow t_1} \downarrow\uparrow$$

$$\frac{}{\Gamma, x : t \vdash x \downarrow t} \text{ T-ID} \qquad \frac{\Gamma \vdash x \downarrow t_1 \qquad (x \neq y)}{\Gamma, y : t_2 \vdash x \downarrow t_1} \text{ T-WEAKEN}$$

$$\frac{}{\Gamma \vdash n \downarrow \mathsf{Nat}} \text{ T-NAT} \qquad \frac{\Gamma \vdash e_1 \uparrow \mathsf{Nat} \qquad \Gamma \vdash e_2 \uparrow \mathsf{Nat} \qquad (\oplus \in \{+, -, *\})}{\Gamma \vdash e_1 \oplus e_2 \downarrow \mathsf{Nat}} \text{ T-ARITH} \qquad \frac{\Gamma \vdash e_1 \uparrow \mathsf{Nat} \qquad \Gamma \vdash e_2 \uparrow \mathsf{Nat}}{\Gamma \vdash e_1 == e_2 \downarrow \mathsf{Bool}} \text{ T-EQ}$$

$$\frac{}{\Gamma \vdash b \downarrow \mathsf{Bool}} \text{ T-BOOL} \qquad \frac{\Gamma \vdash e_1 \uparrow \mathsf{Bool} \qquad \Gamma \vdash e_2 \uparrow t \qquad \Gamma \vdash e_3 \uparrow t}{\Gamma \vdash \mathsf{if}\, e_1 \,\mathsf{then}\, e_2 \,\mathsf{else}\, e_3 \uparrow t} \text{ T-IF}$$

$$\frac{\Gamma \vdash e_1 \downarrow t_1 \qquad \Gamma, x : t_1 \vdash e_2 \uparrow t_2}{\Gamma \vdash \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 \uparrow t_2} \text{ T-LET} \qquad \frac{\Gamma, x : t_i \vdash e \uparrow t_o}{\Gamma \vdash \lambda x.\ e \uparrow t_i \to t_o} \text{ T-LAMBDA} \qquad \frac{\Gamma \vdash e_1 \downarrow t_i \to t_o \qquad \Gamma \vdash e_2 \uparrow t_i}{\Gamma \vdash e_1\ e_2 \downarrow t_o} \text{ T-APP}$$

$$\frac{(\text{for all } 1 \leq i \leq n)\ \Gamma \vdash e_i \uparrow t_i}{\Gamma \vdash (e_1, \ldots, e_n) \uparrow (t_1, \ldots, t_n)} \text{ T-PACK} \qquad \frac{(j \in L) \qquad \Gamma \vdash e \uparrow t_j}{\Gamma \vdash \underline{j}\ e \uparrow +\{\underline{l} : t_l\}_{l \in L}} \text{ T-INJ}$$

$$\frac{???}{\Gamma \vdash \mathsf{match}\ e\ \{p_1 : e_1,\ p_2 : e_2,\ \ldots\} \uparrow t} \text{ T-MATCH}$$

$$\frac{(\Delta(X) = t)}{\Gamma \vdash X \downarrow t} \text{ T-ABBREV} \qquad \frac{\Gamma \vdash e \uparrow t}{\Gamma \vdash (e : t) \downarrow t} \text{ T-ANN}$$