LambdaScript Syntax and Semantics

Alex Kozik

July 18, 2023

Contents

1 Syntax

1.1 Metavariables

Below is a list of meta-variables for different fundamental langauge constructs

$x \in Var$	Variable indentifier
$b \in \{true, false\}$	Boolean
$n \in \mathbb{N}$	Natural number
$s \in \Sigma^*$	String
$\oplus \ \in \ \{+, -, *, /, \%, <, >, <=, >=, ==$	Binary operator
$, ! =\}$	Unary operator

1.2 Expressions

$\langle e \rangle$ ::= n	Integer
$\mid$ b	Boolean
$\mid$ s	String
$\mid$ ()	Nothing
$\mid$ x	Identifier
$\mid$ $e_1 \oplus e_2$	Binary Operation
$\mid$ $(e_1,\ e_2,\ ...,\ e_n)$	Vector
$\mid$ [ ]	Nil (empty list)
$\mid$ e_1 :: e_2	Cons (nonempty list)
$\mid$ **fn** $p \rightarrow e$	Function
$\mid$ **bind** $p \leftarrow e_1$ **in** e_2	Bind expression
$\mid$ **bind** $p\ \ p_1\ ...\ p_n \leftarrow e_1$ **in** e_2	Bind expression
$\mid$ **bind rec** $f \leftarrow$ **fn** p $\rightarrow e_1$ **in** e_2	Recursive function
$\mid$ **bind rec** $f\ \ p_1\ ...\ p_n \leftarrow e_1$ **in** e_2	Recursive function
$\mid$ $e_1\ \ e_2$	Function application
$\mid$ **if** e_1 **then** e_2 **else** e_3	Ternary expressions
$\mid$ **switch** e_0 => $\mid p_1 \rightarrow e_1\ ...\ \mid p_n \rightarrow e_n$ **end**	Switch expression

1.3 Patterns

$\langle p \rangle$ ::=	_	Wildcard pattern*
|	x	Identifier pattern**
|	()	Nothing pattern
|	b	Boolean pattern
|	n	Integer pattern
|	s	String pattern
|	$(p_1, p_2, ..., p_n)$	Vector pattern
|	[ ]	Nil pattern
|	$p_1 :: p_2$	Cons pattern***

* The wildcard pattern matches any value

** The identifier pattern matches any value and produces a binding to it

*** The cons pattern matches a non empty list, but only p_1 matches the head of the list and p_2 matches the remainder of the list

1.4 Values

$\langle v \rangle$ ::=	n	Integer value
|	s	String value
|	b	Boolean value
|	()	Nothing value
|	[ ]	Nil value
|	$v_1 :: v_2$	Cons value
|	(Δ, p, e)	Function Closure

1.5 Types

$\langle t \rangle$ ::=	int	Integer type
|	bool	Boolean type
|	str	String type
|	ng	Nothing type
|	t_i	Type variable
|	$t_1 \rightarrow t_2$	Function type*
|	[t]	List type
|	$(t_1, t_2, ..., t_n)$	Vector type
|	(t)	Parenthesized type*

* The function type operator $\rightarrow$ associates to the right

For example, the type $t_1 \rightarrow t_2 \rightarrow t_3$ is parsed as $t_1 \rightarrow (t_2 \rightarrow t_3)$

Parentheses are the highest precedence operator in the type grammar, and they can be used to counter the right associativity of the arrrow operator.

For example

$$\text{fn f} \rightarrow \text{fn x} \rightarrow \text{f x} : (t_1 \rightarrow t_2) \rightarrow t_1 \rightarrow t_2$$

2 Dynamic Semantics

In order to discuss the dynamic semantics of the programming language, we first need to define a few things.

2.1 Dynamic Environment

LambdaScript uses an environment model to make substitutions in function bodies. The environment is an object defined as follows

$$\Delta \in Var \rightarrow Value$$

It is essentally a function from a set of variable identifiers to a set of values. Note that it is a partial function because its domain will be a subset of Var

- $\Delta(x)$ represents the value x maps to in environment Δ
- $\{\}$ is the empty environment
- $\Delta[x \rightarrow v]$ represents the environment where $\Delta(y) = v$ if $y = x$, and $\Delta(y)$ otherwise
- $D(\Delta)$ is the domain of Δ
- $\Delta_1 \circ \Delta_2$ represents the environment Δ where $\forall y \in D(\Delta_2)$, $\Delta(y) = \Delta_2(y)$, $\forall y \in D(\Delta_1) - D(\Delta_2)$, $\Delta(y) = \Delta_1(y)$. Otherwise, $\Delta(y)$ is not defined.

2.2 Evaluation Relation

The evaluation relation is what describes how an expression is evaluated to a value under a certain environment

Define it as follows

$$(\Delta, e) \Rightarrow v$$

It means the following: Under environment Δ, expression e evaluates to value v

2.3 Pattern Matching Relation

In order to model a value matching some pattern, and producing some bindings, we will use the following relation

$$v \in p \rightarrow \Delta$$

This can be read as "value v matches pattern p and produces bindings Δ"

We will also use the following relation

$$v \notin p$$

This can be read as "value v does not patch pattern p"

2.4 Dynamic Semantics For Patterns

2.4.1 Wildcard Pattern

$$v \in \_ \rightarrow \{\}$$

2.4.2 Variable Identifier

$$v \in x \rightarrow \{\}[x \rightarrow v]$$

2.4.3 Nothing Pattern

$$() \in () \rightarrow \{\}$$

2.4.4 Boolean Pattern

$$b \in b \rightarrow \{\}$$

2.4.5 Integer Pattern

$$i \in i \rightarrow \{\}$$

2.4.6 String Pattern

$$s \in s \rightarrow \{\}$$

2.4.7 Nil Pattern

$$[] \in [] \rightarrow \{\}$$

2.4.8 Vector Pattern

$$\frac{(v_1, v_2, ..., v_n) \in (p_1, p_2, ..., p_n) \rightarrow \Delta_1 \circ \Delta_2 \circ ... \circ ...\Delta_n}{\begin{array}{c} v_1 \in p_1 \rightarrow \Delta_1 \\ v_2 \in p_2 \rightarrow \Delta_2 \\ \cdots \\ v_n \in p_n \rightarrow \Delta_n \end{array}}$$

2.4.9 Cons Pattern

$$\frac{v_1 :: v_2 \in p_1 :: p_2 \to \Delta_1 \circ \Delta_2}{\begin{array}{c} v_1 \in p_1 \to \Delta_1 \\ v_2 \in p_2 \to \Delta_2 \end{array}}$$

2.5 Basic Dynamic Semantics

2.5.1 Value

$$(\Delta, v) \Rightarrow v$$

A value always evaluates to itself

2.5.2 Variable Identifiers

$$(\Delta, x) \Rightarrow \Delta(x)$$

To evaluate an identifier x, it is simply looked up in the environment Δ

2.5.3 Vector

$$\frac{(\Delta, (e_1, e_2, ..., e_n)) \Rightarrow (v_1, v_2, ..., v_n)}{\begin{array}{c} (\Delta, e_1) \Rightarrow v_1 \\ (\Delta, e_2) \Rightarrow v_2 \\ \cdots \\ (\Delta, e_n) \Rightarrow v_n \end{array}}$$

To evaluate a vector, evaluate each sub expression, then construct a new vector with the values

2.5.4 Cons

$$\frac{(\Delta, e_1 :: e_2) \Rightarrow v_1 :: v_2}{\begin{array}{c} (\Delta, e_1) \Rightarrow v_1 \\ (\Delta, e_2) \Rightarrow v_2 \end{array}}$$

To evaluate a cons expression, evaluate the two operands, then return the first argument prepended to the second

2.6 Switch Expression

A switch expression uses an expression, call it e_0 and a list of branches. Each branch consists of a pattern and a body.

First, e_0 is evaluted to a value v_0 using the current environment Δ

Starting from the first branch, v_0 is compared to its pattern. If it matches, certain bindings are produced, which are used to evaluate its body. That value is then returned.

This process of comparing v_0 to the pattern of a branch continues until a match is made.

$$(\Delta, \text{switch e => } |p_1 \to e_1 ... |p_n \to e_n \text{ end}) \implies v'$$

$$(\Delta, e) \implies v$$

$$v \notin p_i \text{ for } i < m$$

$$v \in p_m \to \Delta_m \text{ where } 1 \leq m \leq n$$

$$(\Delta \circ \Delta_m, e_m) \implies v'$$

Let's go through those statements one by one

1. $(\Delta, e) \implies v$ shows that e evalutes to v under environment Δ
2. $v \notin p_i$ for $i < m$ shows that v doesn't match the first $m - 1$ patterns
3. $v \in p_m \to \Delta_m$ where $1 \leq m \leq n$ shows that v matches the m^{th} pattern and produces bindings Δ_m
4. $(\Delta \circ \Delta_m, e_m) \implies v'$ shows that the body of the m^{th} branch evaluates to a value v' under the external environment Δ composed with the new bindings Δ_m. v' is what the entire switch expression evaluates to.

2.7 Ternary Expression

There are two rules regarding the dynamic semantics of ternary expressions. There is one for when the predicate is true and one for when the predicate is false.

$$(\Delta, \text{if } e_1 \text{ then } e_2 \text{ else } e_3) \implies v$$

$$(\Delta, e_1) \implies \text{true}$$

$$(\Delta, e_2) \implies v$$

$$\overline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}$$

$$(\Delta, \text{if } e_1 \text{ then } e_2 \text{ else } e_3) \implies v$$

$$\overline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad}$$

bro

$$(\Delta, e_1) \implies \text{false}$$

$$(\Delta, e_3) \implies v$$

2.8 Function

A function in LambdaScript evaluates to a function closure, which consists of three parts: the environment, the pattern, and the body.

$$(\Delta, \text{fn } p \to e) \implies (\Delta, p, e)$$

2.9 Function Application

$$(\Delta, e_1\ e_2) \implies v$$

$$\overline{\qquad\qquad\qquad\qquad}$$

$$(\Delta, e_1) \implies (\Delta_c, p_c, e_c)$$

$$(\Delta, e_2) \implies v_2$$

$$v_2 \in p_c \to \Delta_n$$

$$(\Delta_c \circ \Delta_n, e_c) \implies v$$

Let's go through these statements one by one

1. $(\Delta, e_1) \implies (\Delta_c, p_c, e_c)$ states that e_1 evaluates to a function closure. This is crucial because if e_1 is not a function closure, it cannot be applied.
2. $(\Delta, e_2) \implies v_n$ states that e_2, the argument, evaluates to a value v_2
3. $v_2 \in p_c \implies \Delta_2$ states that v_2, the value of the argument, matches the pattern of the function closure p_c and produces bindings Δ_n
4. $(\Delta_c \circ \Delta_n, e_c) \to v$ states that e_2, the body of the function closure, evaluates to a value v under the environment of the function closure, Δ_c composed with the new bindings Δ_n, which result from matching the value of the argument v against the pattern p_c. The entire expression evalutes to v.

2.10 Bind Expression

A bind expression is just syntactic sugar for the application of an anonymous function to an argument

$$\text{bind } p \leftarrow e_1 \text{ in } e_2$$

Is equivalent to

$$(\text{fn } p \rightarrow e_2)\ e_1$$

Bind expressions also have syntactic sugar for function arguments as follows

$$\text{bind } p\ x_1\ x_2\ ...\ x_n \leftarrow e_1 \text{ in } e_2$$

Is equivalent to

$$(\text{fn } p \rightarrow e_2)\ (\text{fn } x_1 \rightarrow\ \text{fn } x_2 \rightarrow \ldots\ \text{fn } x_n \rightarrow e_1)$$

Consequently, the dynamic semantics of the bind expression is already defined by the dynamic semantics of the function and function application

2.11 Recursive Bind Expression

Moreover, a recursive function can be defined as follows

$$\text{bind rec } f\ x_1\ x_2\ ...\ x_n \leftarrow e_1 \text{ in } e_2$$

The only difference between this and the standard bind expression is the following rule:
Suppose we have a recursive bind expression of the following form

$$\text{bind rec } f\ x_1\ x_2\ ...\ x_n \leftarrow e_1 \text{ in } e_2$$

De-sugar that to yield the following

$$\text{bind rec } f \leftarrow\ \text{fn } x_1 \rightarrow\ \text{fn } x_2 \rightarrow \ldots\ \text{fn } x_n \rightarrow e_1 \text{ in } e_2$$

Now, suppose for the purposes of intuition that we are discussing a non-recursive bind expression, such at this

$$\text{bind } f \leftarrow\ \text{fn } x_1 \rightarrow\ \text{fn } x_2 \rightarrow \ldots\ \text{fn } x_n \rightarrow e_1 \text{ in } e_2$$

Evaluating the expression between $\leftarrow$ and "in" would yield the following

$$(\Delta_{\text{standard}}, x_1,\ \text{fn } x_2 \rightarrow \ldots\ \text{fn } x_n \rightarrow e_1)$$

Now, create a special type of dynamic environment that satisfies the following two properties

1.

$$\forall g \in Var - \{f\},\ \Delta_{\text{recursive}}(g) = \Delta_{\text{standard}}(g)$$

2.

$$\Delta_{\text{recursive}}(f) = (\Delta_{\text{recursive}}, x_1,\ \text{fn } x_2 \to \ldots\ \text{fn } x_n \to e_1)$$

Now we go back to discussing the recursive bind expression

$$\text{bind rec } f \leftarrow\ \text{fn } x_1 \to\ \text{fn } x_2 \to \ldots\ \text{fn } x_n \to e_1 \text{ in } e_2$$

The expression between ← and "in" evaluates to

$$(\Delta_{\text{recursive}}, x_1,\ \text{fn } x_2 \to \ldots\ \text{fn } x_n \to e_1)$$

So we now have the following. Note that this is a theoretical representation, not a syntactic one.

$$\text{bind rec } f \leftarrow (\Delta_{\text{recursive}}, x_1,\ \text{fn } x_2 \to \ldots\ \text{fn } x_n \to e_1) \text{ in } e_2$$

Desugar this to yield the following

$$(\text{fn } f \to e_2)(\Delta_{\text{recursive}}, x_1,\ \text{fn } x_2 \to \ldots\ \text{fn } x_n \to e_1)$$

This expression can now be further evaluated using the dynamic semantics regarding function application.

3 Static Semantics

3.1 Static Environment

Similar to the dynamic environment, which represents mappings from identifiers to values, we can define a static environment, which represents mappings from identifiers to types.

The static environment is defined as follows

$$\Sigma \in Var \to \mathit{Type}$$

It is a partial function from identifiers to types

It has the following operations

- $\Sigma(x)$ represents the type x maps to in environment Σ
- $\{\}$ is the empty environment
- $\Sigma[x \to v]$ represents the environment where $\Sigma(y) = v$ if $y = x$, and $\Sigma(y)$ otherwise
- $D(\Sigma)$ is the domain of Σ
- $\Sigma_1 \circ \Sigma_2$ represents the environment Σ where $\forall y \in D(\Sigma_2)$, $\Sigma(y) = \Sigma_2(y)$, $\forall y \in D(\Sigma_1) - D(\Sigma_2)$, $\Sigma(y) = \Sigma_1(y)$. Otherwise, $\Sigma(y)$ is not defined.

3.2 Type Equations

A type equation can be thought of intuitively as an equality between two types.

Formally, a system of type equations is defined as follows.

$$\mathcal{E} \in \mathit{Type} \times \mathit{Type}$$

Mathematically, it is nothing more than a set of pairs, where each pair consists of two types that are thought to be equal.

1. The empty system is denoted $\{\}$
2. The union of two systems is denoted $\mathcal{E}_1 \cup \mathcal{E}_2$

3.3 Equality of Type Equations

Define $\cong$ to be an equivalence relation on $Type \times Type$ as follows

$$\cong = \{((t_1, t_2), (t_2, t_1)) \mid \forall (t_1, t_2) \in Type \times Type\}$$

If can also be defined as

$$(t_1, t_2) \cong (t_3, t_4) = ((t_1, t_2) = (t_3, t_4) \vee (t_1, t_2) = (t_4, t_3))$$

$\cong$ captures the notion of two type equations containing the exact same information

For example

$$(t_1 = int) \cong (int, t_1)$$
$$(t_1, t_2) \cong (t_1, t_2)$$

3.4 Universal Quantification of Type Variables

There is a scenario later on in which universal quantification of type variables is required for a sound type system.

A type variable of the form t_n, where $n \in \mathbb{N}$, is a standard type variable with "a single identity"

This means that, for example, the following system would be inconsistent

$$t_1 = int$$
$$t_1 = bool$$
$$\Longrightarrow$$
$$int = bool$$

In order to allow polymorphism, universally quantified variables are required. They are denoted u_n, where $n \in \mathbb{N}$.

3.5 Reduction of Type Equations

The reduction of type equations is analogus to Gaussian elimination of linear systems of equations in linear algebra. The purpose of reduction is to take a system of type equations and, if possible, change it into a form that makes it easier to deduce the solution.

The reduction function is defined as follows

$$R : P(Type) \to P(Type)$$

- $R(\{\}) = \{\}$
- If $\mathcal{E} \neq \{\}$, pick an arbitrary type equation in $\mathcal{E}$, call it $\epsilon = (t_1, t_2)$
- Let $\mathcal{E}' = \{\epsilon' \in \mathcal{E} \mid \text{not } \epsilon' \cong \epsilon\}$ be the remainder of the type equations
- If $t_1 = t_2$, $R(\mathcal{E}) = R(\mathcal{E}')$
- If $t_1 = i_1 \to o_1$ and $t_2 = i_2 \to o_2$ are function types, $R(\mathcal{E}) = R(\{(i_1, i_2), (o_1, o_2)\} \cup \mathcal{E}')$
- If $t_1 = [l_1]$ and $t_2 = [l_2]$ are both list types, $R(\mathcal{E}) = R(\{(l_1, l_2)\} \cup \mathcal{E}')$
- If $t_1 = (t_{11}, t_{12}, \ldots, t_{1n})$ and $t_2 = (t_{21}, t_{22}, \ldots, t_{2n})$ are vector types, $R(\mathcal{E}) = R(\bigcup_{i=1}^{n}\{(t_{1i}, t_{2i})\} \cup \mathcal{E}')$
-

3.6 Instantiation of Types

The instantiation function is a function from types to types. It takes a type and does the following:

1. For each unique univerally quantified type variable, generate a fresh type variable. Create a 1 to 1 mapping from the universal variables to the concrete ones.
2. Using the mapping, replace the universally quantified variables with the concrete ones.
3. Return this new type.

For example

$$I(t_1 \to u_1 \to u_2) = (t_1 \to t_2 \to t_3)$$

3.7 Generalization of Types

3.8 Type Relation

In order to discuss the relationships between the expressions of LambdaScript and the corresponding types, we must first define a type relation.

The type relation is defined as follows

$$\Sigma \to e : t \to \mathcal{E}$$

It can be read as follows: "under static environment Σ, expression e is of type t and produces a set of type equations $\mathcal{E}$"

3.9 Type Inference relation for Patterns

Similar to dynamic semantics, patterns produce bindings from identifiers to Types

The type inference relation for patterns is defined as follows

$$p : t \to \Sigma$$

This can be read as "pattern p is of type t and creates static bindings Σ"

3.10 Type Inference for Patterns

3.10.1 Integer Pattern

$$i : int \to \{\}$$

3.10.2 Boolean Pattern

$$b : bool \to \{\}$$

3.10.3 String Pattern

$$s : str \to \{\}$$

3.10.4 Nothing Pattern

$$() : ng \to \{\}$$

3.10.5 Wildcard Pattern

$$\_ : t \to \{\}$$

3.10.6 Identifier Pattern

$$x : t \to \{x : t\}$$

3.10.7 Nil Pattern

$$[\,] : [t] \to \{\}$$

3.10.8 Cons Pattern

$$\frac{p_1 :: p_2 : [t] \to \mathcal{E}_1 \cup \mathcal{E}_1}{\begin{array}{c} p_1 : t \to \mathcal{E}_1 \\ p_2 : [t] \to \mathcal{E}_2 \end{array}}$$

3.10.9 Vector Pattern

$$\frac{(p_1, p_2, \ldots, p_n) : (t_1, t_2, \ldots, t_n) \to \bigcup_{i=1}^{n} \mathcal{E}_i}{\begin{array}{c} p_1 : t_1 \to \mathcal{E}_1 \\ p_2 : t_2 \to \mathcal{E}_2 \\ \ldots \\ p_n : t_n \to \mathcal{E}_n \end{array}}$$

3.11 Type Inference of Basic Expressions

3.11.1 Integer

$$\Sigma \to i : int \to \{\}$$

3.11.2 Boolean

$$\Sigma \to b : bool \to \{\}$$

3.11.3 String

$$\Sigma \to s : str \to \{\}$$

3.11.4 Nothing

$$\Sigma \to () : ng \to \{\}$$

3.11.5 Identifier

$$\Sigma \to x : I(\Sigma(x)) \to \{\}$$

Notice that I is used to instantiate the type of x is Σ. This is required since $\Sigma(x)$ may consist of universally quantified type variables, which must be replaced with concrete ones before insertion into type equations.

3.11.6 Nil

$$\Sigma \to [\,] : [t] \to \{\}$$

3.11.7 Vector

$$\frac{\Sigma \to (e_1, e_2, \ldots, e_n) : (t_1, t_2, \ldots, t_n) \to \bigcup_{i=1}^{n} \mathcal{E}_i}{\begin{array}{c} \Sigma \to e_1 : t_1 \to \mathcal{E}_1 \\ \Sigma \to e_2 : t_2 \to \mathcal{E}_2 \\ \ldots \\ \Sigma \to e_n : t_n \to \mathcal{E}_n \end{array}}$$

3.11.8 Cons

$$\frac{\Sigma \to e_1 :: e_2 : t_2 \to \{[t_1] = t_2\} \cup \mathcal{E}_1 \cup \mathcal{E}_2}{\begin{array}{c} \Sigma \to e_1 : t_1 \to \mathcal{E}_1 \\ \Sigma \to e_2 : t_2 \to \mathcal{E}_2 \end{array}}$$

3.12 Switch Expression

$$
\frac{\begin{array}{c}
p_1 : t_{n+1} \to \Sigma_1 \\
p_2 : t_{n+2} \to \Sigma_1 \\
\dots \\
p_n : t_{2n} \to \Sigma_1 \\
\Sigma \circ \Sigma_1 : e_1 : t_1 \to \mathcal{E}_1 \\
\Sigma \circ \Sigma_2 : e_2 : t_2 \to \mathcal{E}_2 \\
\dots \\
\Sigma \circ \Sigma_n : e_n : t_n \to \mathcal{E}_n \\
\mathcal{E}_{\text{types equal}} = \bigcup_{i=1}^{n} \{t = t_i\} \\
\mathcal{E}_{\text{branches}} = \bigcup_{i=1}^{n} \mathcal{E}_i \\
\mathcal{E}_{\text{patterns}} = \bigcup_{i=n+1}^{2n} \{t_p = t_i\}
\end{array}}{\Sigma \to \text{switch } e => |p_1 \to e_1|...|p_n \to e_n \text{ end} : t \to \mathcal{E}_{\text{branches}} \cup \mathcal{E}_{\text{types equal}} \cup \mathcal{E}_{\text{patterns}}}
$$

The type of a switch expression is essentially the type of any given branch. The types of the branch bodies must all be the same.

3.13 Ternary Expression

$$
\frac{\begin{array}{c}
\Sigma \to e_1 : t_1 \to \mathcal{E}_1 \\
\Sigma \to e_2 : t_2 \to \mathcal{E}_2 \\
\Sigma \to e_3 : t_3 \to \mathcal{E}_3 \\
\mathcal{E}_0 = \{t_1 = bool, t = t_1, t = t_2\}
\end{array}}{\Sigma \to \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t \to \mathcal{E}_0 \cup \mathcal{E}_1 \cup \mathcal{E}_2 \cup \mathcal{E}_3}
$$

3.14 Function

$$\frac{\Sigma \to (\text{fn } p \to e) : (t_1 \to t_2) \to \mathcal{E}}{\begin{array}{c} p : t_1 \to \Sigma_p \\ \Sigma \circ \Sigma_p \to e : t_2 \to \mathcal{E} \end{array}}$$

1. $p : t_1 \to \Sigma_b$ states that the functions pattern p is of type t_1 and produces static bindings Σ_p
2. $\Sigma \circ \Sigma_p \to e : t_2 \to \mathcal{E}$ states that the function's body e is of type t_2 when evaluated under the external static environment Σ composed with the new bindings produced by p, which are Σ_p. Additionally, type equations $\mathcal{E}$ are produced.
3. The input type of the function is t_1, and the output type of the function is t_2. Therefore, the type of the entire expression is $t_1 \to t_2$. Additionally, the equations resulting from inference of the function body are returned.

3.15 Function Application

$$\frac{\Sigma \to e_1\ e_2 : t \to \mathcal{E}_1 \cup \mathcal{E}_2 \cup \mathcal{E}_3}{\begin{array}{c} \Sigma \to e_1 : t_1 \to \mathcal{E}_1 \\ \Sigma \to e_2 : t_2 \to \mathcal{E}_2 \\ \mathcal{E}_3 = \{t_1 = t_2 \to t\} \end{array}}$$