L1 Syntax and Documentation

Arthur Giesel Vedana

January 31, 2017

Contents

Introduction

The *L*1 programming language is a functional language with eager left-to-right evaluation. It has a simple I/O system supporting only direct string operations. It is a trait based strongly and statically typed language supporting both explicit and implicit typing.

This document both specifies the *L*1 language and shows its implementation in F#. It is divided into 4 categories:

1. Abstract Syntax and Semantics

   This defines the abstract syntax and semantics of the language. It only contains the bare minimum for the language to function, without any syntactic sugar.

2. Concrete Syntax

   This is the actual syntax when programming for *L*1. This defines all operators, syntactic sugar and other aspects of the language.

3. Implementation

   Technical aspects on how *L*1 is implemented in F#, showing the interpreter, evaluator and type inference.

4. Changelog

   A chronological list of changes made both to the language definition and its implementation.

1 Abstract Syntax and Semantics

1.1 Abstract Syntax

1.1.1 Expressions

Programs in $L1$ are expressions. Each expression is a member of the abstract syntax tree defined below. The syntax tree will be constructed in parts, with an explanation of what each expression means and their uses. The full syntax tree can be obtained by simply joining all the separate sections.

Constants and Variables $L1$ has support for a few basic constants and variables.

$$
\begin{array}{rcl}
e & ::= & n \\
  & | & b \\
  & | & c \\
  & | & x
\end{array}
$$

$$
\begin{array}{rcl}
x & ::= & \{x_0, x_1, \ldots\}
\end{array}
$$

$$
\begin{array}{rcl}
b & ::= & true \mid false \\
n & ::= & \mathbb{Z} \\
c & ::= & \text{`}char\text{'} \\
char & ::= & \texttt{ASCII characters}
\end{array}
$$

The constants available for the language are:

- Booleans
- Integers
- Characters

For variables, one can define any number of identifiers to be used throughout the program. These variables must be associated with other terms in some way (functions, `let` declarations, etc) as they cannot be evaluated on their own.

Conditional Like most functional languages, $L1$ provides a conditional expression. This expression, like all others, always returns a value.

$$
\begin{array}{rcl}
e & ::= & \cdots \\
  & | & \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3
\end{array}
$$

Binary Operations $L1$ provides a series of binary operations built-in into the language.

$$
\begin{array}{lll}
e & ::= & \cdots \\
 & | & e_1 \; op \; e_2
\end{array}
$$

$$
\begin{array}{lll}
op & ::= & opNum \mid opEq \mid opIneq \mid opBool \\
opNum & ::= & + \mid - \mid * \mid \div \\
opEq & ::= & = \mid \neq \\
opIneq & ::= & < \mid \leq \mid > \mid \geq \\
boolOp & ::= & \wedge \mid \vee
\end{array}
$$

They are divided into separate categories, each one requiring terms of specific types. The following list describes the requirements and meaning of each category of built-in operators:

- Numerical

  These operators require numbers and also return numbers.

- Equality

  These operators $(=, \neq)$ compare two different values for equality, returning a boolean. The values type must conform to the Equatable trait.

- Inequality

  These operators compare two different values for order, returning a boolean. The values type must conform to the Equatable and Orderable trait.

- Boolean Operators

  These operators perform logical operations on boolean values.

Functions   The expressions below all relate to function and function application in the $L1$ language.

$$
\begin{array}{lll}
e & ::= & \cdots \\
 & | & \mathbf{fn} \; x : T \Rightarrow e \\
 & | & \mathbf{fn} \; x \Rightarrow e \\
 & | & \mathbf{rec} \; x_1 : T_1 \rightarrow T_2 \;\; x_2 : T_1 \Rightarrow e \\
 & | & \mathbf{rec} \; x_1 \; x_2 \Rightarrow e \\
 & | & e_1 \; e_2
\end{array}
$$

The first two expressions define simple unnamed functions that take exactly one parameter, x. The first of these explicitly says what the type of this parameter is, while the second one leaves the job of inferring the type to the compiler. When a value v is applied to a function, all occurrences of x in e are replaced by v and then the expression is evaluated. (In reality, this replacement only occurs as needed to be more efficient (see 1.2), but the result is the same).

The following two expressions define recursive functions that also take one parameter. In these expressions, x_1 is the name of the function that can be recursively called inside e. x_2 is the identifier of the single parameter for the function. As with unnamed

functions, recursive functions have two variations: explicitly and implicitly typed. In the explicitly typed version, x_1 has to be typed as a function that takes the type of x_2 (that is, T_1) and returns another type T_2.

The last expression is the application of e_2 to a function e_1.

Let declarations   The expressions below are used to declare identifiers for sub-expressions. This both helps increase readability and reduce repetition when writing programs.

$$
\begin{array}{lll}
e & ::= & \cdots \\
  & | & \texttt{let } x : T = e_1 \texttt{ in } e_2 \\
  & | & \texttt{let } x = e_1 \texttt{ in } e_2
\end{array}
$$

There are two versions of the `let` expression: one that is explicitly typed and one that is implicitly typed, just as with function expressions. Again, similar to functions, this expression works by replacing all occurrences of x by the value of e_1 in e_2 and then evaluating the resulting expression.

Exceptions   Exceptions are used when an expression cannot be evaluated correctly because of reasons outside of the normal. These reasons include division by zero, accessing an empty list, etc. Since these situations cannot be known before evaluating a program, there is no way to defend against them in a type system. To deal with this, we must have an expression to deal with exceptions. It is also a good idea to be able to create exceptions inside a program, so we have an expression to do just that.

$$
\begin{array}{lll}
e & ::= & \cdots \\
  & | & \textit{raise} \\
  & | & \texttt{try } e_1 \texttt{ with } e_2
\end{array}
$$

The first expression simply evaluates to an exception that will be propagated to its parent expression.

The second expression evaluated e_1 and, if an exception is encountered, evaluates e_2. If the evaluation of e_1 does not encounter an exception, the resulting value is used, and e_2 is discarded.

Lists   The $L1$ language has built-in lists. Each list is a homogeneous finite-length collection of values, meaning that a list contains only elements of the same type. There are also basic operations on lists, such as appending, obtaining the first element of a list, etc.

$$
\begin{array}{lll}
e & ::= & \cdots \\
  & | & \textit{nil} \\
  & | & e_1 :: e_2 \\
  & | & \texttt{isempty } e \\
  & | & \texttt{hd } e \\
  & | & \texttt{tl } e
\end{array}
$$

The first expression is the empty list. It is the only zero-length list possible, and all other lists are constructed on top of it.

The second expression is the append operation, adding e_1 to the front of the list e_2.

The last three expressions are operations on lists.

The first one tests whether e is the empty list (i.e. *nil*), returning true if positive and false if negative.

The second one returns the first element of a non-empty list. If the list is empty, an exception is raised.

The third one returns the list obtained by removing the first element of a non-empty list. If the list is empty, an exception is raised.

Input and Ouput   These expressions allow a program to interact with a user. Because of this, they are inherently non-deterministic.

$$
\begin{array}{rcl}
e & ::= & \cdots \\
  & | & \texttt{input} \\
  & | & \texttt{output } e
\end{array}
$$

The `input` expression receives a line of text from the user. It always evaluates to a list of characters, but the length of this list is only known at run-time.

The `output` expression prints a list of characters to the user. It is only evaluated for its side-effect of printing the list, resulting in a useless value. This is dealt with by the next set of expressions in the language.

Sequence   Up until now, no expressions in $L1$ had side effects, so all expressions were written to obtain their resulting value. With the introduction of input and (especially) output expressions, now there exist expressions that are evaluated only for their side effect. Because of this, it is import to enable a programmer to ignore the resulting value and continue evaluating other expressions.

$$
\begin{array}{rcl}
e & ::= & \cdots \\
  & | & skip \\
  & | & e_1 \; ; \; e_2
\end{array}
$$

The first new expression (*skip*) is an empty value, always meant to be ignored. It is the result of evaluating an `output` expression and any other expression that is only evaluated for its side-effects.

The second expression evaluates e_1, ignores its results and evaluates e_2. It requires that e_1 evaluates to no useful value (i.e. evaluates to *skip*).

1.1.2   Types

Since $L1$ is strongly typed, every (valid) expression has exactly one type associated with it. Some expressions require the programmer to explicitly declare types of identifiers, such as `let` declarations and functions. Other expressions, such as $e_1 = e_2$, or even constants, such as 1 or *true*, have types implicitly associated with them. These

types are used by the type system (see 1.3) to check whether an expression is valid or not, avoiding run-time errors that can be detected at compile time.

Base Types   These are all of the base types available in *L*1. They are considered constant, and cannot be deconstructed or replaced by other types.

$$T \quad ::= \quad \text{Int}$$
$$\quad | \quad \text{Bool}$$
$$\quad | \quad \text{Char}$$
$$\quad | \quad \text{Unit}$$

Parametric Types   These types are composed of 1 or more other types. If the argument types of a parametric type are constant, we can say that the parametric type itself is also constant. This means that 'Int *list*' is a constant type, while '$VarType_1 \rightarrow Int$' is not.

$$T \quad ::= \quad \cdots$$
$$\quad | \quad T_1 \rightarrow T_2$$
$$\quad | \quad T \ list$$

Variable Types   These types represent an unknown constant type. Explicitly typed expressions cannot be given variable types, but they are used by the type system for implicitly typed expressions. In the course of the type inference, the type system can replace variable types for their corresponding parametric or constant type.

It is important to realize that variable types already represent a unique type with an unknown identity. This means that a variable type may only be replaced by the specific type which it represents and not any other type. This distinction becomes important when talking about polymorphism, which uses variable types, along with universal quantifiers, to represent a placeholder for any possible type (this is discussed in greater detail in 1.3.1).

$$T \quad ::= \quad \cdots$$
$$\quad | \quad X^{Traits}$$

$$X \quad ::= \quad X_1, X_2, \dots$$

1.1.3   Traits

Types can possess traits, which define certain behaviors that are expected of said type. Constant types always have their trait information implicitly defined, since this information is included in the language. Variable types, on the other hand, can explicitly state which traits they possess, restricting the set of possible constant types they can represent.

$$Traits \quad ::= \quad \varnothing$$
$$\quad | \quad \{Trait\} \cup Traits$$

$$
\begin{array}{lll}
Trait & ::= & Equatable \\
& | & Orderable
\end{array}
$$

Equatable If a type T is *Equatable*, expressions of type T can use the equality operators ($=, \neq$).

To define the set of types that belong to *Equatable*, the following rules are used:

$\{\text{Int, Bool, Char}\} \subset Equatable$
$T \in Equatable \implies T\ list \in Equatable$
$X^{Traits} \in Equatable \implies Equatable \in Traits$

Orderable If a type T is *Orderable*, expressions of type T can use the inequality operators ($<, \leq, >, \geq$). Any type that is *Orderable* is also *Equatable*. In practice, the only difference between *Orderable* and *Equatable* is that the base type Bool is not in *Orderable*.

To define the set of types that belong to *Orderable*, the following rules are used:

$\{\text{Int, Char}\} \subset Orderable$
$T \in Orderable \implies T\ list \in Orderable$
$X^{Traits} \in Orderable \implies Orderable \in Traits$

1.2 Operational Semantics

The $L1$ language is evaluated using a big-step evaluation with environments. This evaluation reduces an expression into a value directly, not necessarily having a rule of evaluation for every possible expression. To stop programmers from creating programs that cannot be evaluated, a type inference system will be specified later.

Value A value is the result of the evaluation of an expression in big-step. This set of values is different from the set of expressions of $L1$, even though they share many similarities.

Environment An environment is a mapping of identifiers to values that is extended each time a *let* declaration in encountered. Every expression must be evaluated before being stored in the environment, which means that $L1$ has eager evaluation.

Below are the definitions of both values and environments:

$$env \quad ::= \quad \{\} \mid \{x \to v\} \cup env$$

$$
\begin{aligned}
v \quad ::= \quad & n \\
\mid \quad & b \\
\mid \quad & c \\
\mid \quad & nil \\
\mid \quad & v_1 :: v_2 \\
\mid \quad & raise \\
\mid \quad & skip \\
\mid \quad & \langle x, e, env \rangle \\
\mid \quad & \langle x_1, x_2, e, env \rangle
\end{aligned}
$$

The values $\langle x, e, env \rangle$ and $\langle x_1, x_2, e, env \rangle$ are closures and recursive closures, respectively. They represent the result of evaluating functions and recursive functions, both and store the environment at the moment of evaluation. This means that $L1$ has static scope, since closures capture the environment at the moment of evaluation and $L1$ has eager evaluation.

Closures also store the identifier for the parameter of the respective function (as x), along with the function body (as e). Recursive closures, besides storing the identifier for the parameter (as x_2) and the function body, also store the name of the function (as x_1). This allows the function to be called inside its own body, something that the simple closure does not allow.

1.2.1 Big-Step Rules

$$\text{env} \vdash n \Downarrow n \quad \text{(BS-Num)}$$

$$\text{env} \vdash b \Downarrow b \quad \text{(BS-Bool)}$$

7

$$\text{env} \vdash c \Downarrow c \qquad\qquad\qquad\text{(BS-CHAR)}$$

$$\frac{\text{env}(x) = v}{\text{env} \vdash x \Downarrow v} \qquad\qquad\qquad\text{(BS-IDENT)}$$

Numerical Operations The $L1$ language only supports integers, so all operations are done on integer numbers. This means that the division always results in a whole number, truncated towards zero.

$$\frac{\text{env} \vdash e_1 \Downarrow n_1 \qquad \text{env} \vdash e_2 \Downarrow n_2 \qquad \|n\| = \|n_1\| + \|n_2\|}{\text{env} \vdash e_1 + e_2 \Downarrow n} \qquad\text{(BS-+)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow n_1 \qquad \text{env} \vdash e_2 \Downarrow n_2 \qquad \|n\| = \|n_1\| - \|n_2\|}{\text{env} \vdash e_1 - e_2 \Downarrow n} \qquad\text{(BS--)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow n_1 \qquad \text{env} \vdash e_2 \Downarrow n_2 \qquad \|n\| = \|n_1\| * \|n_2\|}{\text{env} \vdash e_1 * e_2 \Downarrow n} \qquad\text{(BS-*)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow n_1 \qquad \text{env} \vdash e_2 \Downarrow 0}{\text{env} \vdash e_1 \div e_2 \Downarrow raise} \qquad\text{(BS-}\div\text{ZERO)}$$

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow n_1 \qquad \text{env} \vdash e_2 \Downarrow n_2 \\ \|n_2\| \neq 0 \qquad \|n\| = \|n_1\| \div \|n_2\|\end{array}}{\text{env} \vdash e_1 \div e_2 \Downarrow n} \qquad\text{(BS-}\div\text{)}$$

Equality Operations The equality operators (= and ≠) allow comparison of certain expressions with other expressions of the same kind. In this way, it is a polymorphic operator, being usable in different contexts. Even so, it is important to realize that it only compares values of the same kind (numbers with numbers, characters with characters, etc).

$$\frac{\text{env} \vdash e_1 \Downarrow n_1 \qquad \text{env} \vdash e_2 \Downarrow n_2 \qquad \|n_1\| = \|n_2\|}{\text{env} \vdash e_1 = e_2 \Downarrow true} \quad\text{(BS-=NUMTRUE)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow n_1 \qquad \text{env} \vdash e_2 \Downarrow n_2 \qquad \|n_1\| \neq \|n_2\|}{\text{env} \vdash e_1 = e_2 \Downarrow false} \quad\text{(BS-=NUMFALSE)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow c_1 \qquad \text{env} \vdash e_2 \Downarrow c_2 \qquad \|c_1\| = \|c_2\|}{\text{env} \vdash e_1 = e_2 \Downarrow true} \quad\text{(BS-=CHARTRUE)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow c_1 \qquad \text{env} \vdash e_2 \Downarrow c_2 \qquad \|c_1\| \neq \|c_2\|}{\text{env} \vdash e_1 = e_2 \Downarrow false} \text{ (BS-=CharFalse)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow b_1 \qquad \text{env} \vdash e_2 \Downarrow b_2 \qquad \|b_1\| = \|b_2\|}{\text{env} \vdash e_1 = e_2 \Downarrow true} \text{ (BS-=BoolTrue)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow b_1 \qquad \text{env} \vdash e_2 \Downarrow b_2 \qquad \|b_1\| \neq \|b_2\|}{\text{env} \vdash e_1 = e_2 \Downarrow false} \text{ (BS-=BoolFalse)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow nil \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 = e_2 \Downarrow true} \text{ (BS-=NilTrue)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow v_1 :: v_2 \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 = e_2 \Downarrow false} \text{ (BS-=NilFalse1)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow nil \qquad \text{env} \vdash e_2 \Downarrow v_1 :: v_2}{\text{env} \vdash e_1 = e_2 \Downarrow false} \text{ (BS-=NilFalse2)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow v_1 :: v_2 \qquad \text{env} \vdash e_2 \Downarrow v_3 :: v_4 \qquad \text{env} \vdash v_1 = v_3 \Downarrow false}{\text{env} \vdash e_1 = e_2 \Downarrow false} \text{ (BS-=ListFalse)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow v_1 :: v_2 \qquad \text{env} \vdash e_2 \Downarrow v_3 :: v_4 \qquad \text{env} \vdash v_1 = v_3 \Downarrow true \qquad \text{env} \vdash v_2 = v_4 \Downarrow b}{\text{env} \vdash e_1 = e_2 \Downarrow b} \text{ (BS-=ListTrue)}$$

$$\frac{\text{env} \vdash e_1 = e_2 \Downarrow false}{\text{env} \vdash e_1 \neq e_2 \Downarrow true} \text{ (BS-}\neq\text{True)}$$

$$\frac{\text{env} \vdash e_1 = e_2 \Downarrow true}{\text{env} \vdash e_1 \neq e_2 \Downarrow false} \text{ (BS-}\neq\text{False)}$$

Inequality Operations   The inequality operators function much in the same way as the equality operators. The only difference is that they do not allow comparison of certain kinds of expressions (such as booleans) when such expressions do not have a clear ordering to them.

To reduce the number of rules, some rules are condensed for all inequality operators $(<, \leq, >, \geq)$. The comparison done on numbers is the ordinary numerical comparison. For characters, the ASCII values are compared numerically.

$$\frac{\text{env} \vdash e_1 \Downarrow n_1 \qquad \text{env} \vdash e_2 \Downarrow n_2 \qquad \|n_1\| \, opIneq \, \|n_2\|}{\text{env} \vdash e_1 \, opIneq \, e_2 \Downarrow true} \text{ (BS-INEQNUMTRUE)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow n_1 \qquad \text{env} \vdash e_2 \Downarrow n_2 \qquad \neg \, \|n_1\| \, opIneq \, \|n_2\|}{\text{env} \vdash e_1 \, opIneq \, e_2 \Downarrow true} \text{ (BS-INEQNUMFALSE)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow c_1 \qquad \text{env} \vdash e_2 \Downarrow c_2 \qquad \|c_1\| \, opIneq \, \|c_2\|}{\text{env} \vdash e_1 \, opIneq \, e_2 \Downarrow true} \text{ (BS-INEQCHARTRUE)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow c_1 \qquad \text{env} \vdash e_2 \Downarrow c_2 \qquad \neg \, \|c_1\| \, opIneq \, \|c_2\|}{\text{env} \vdash e_1 \, opIneq \, e_2 \Downarrow true} \text{ (BS-INEQCHARFALSE)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow nil \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 < e_2 \Downarrow false} \text{ (BS-<NIL)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow nil \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 \leq e_2 \Downarrow true} \text{ (BS-}\leq\text{NIL)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow nil \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 > e_2 \Downarrow false} \text{ (BS->NIL)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow nil \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 \geq e_2 \Downarrow true} \text{ (BS-}\geq\text{NIL)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow v_1 :: v_2 \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 < e_2 \Downarrow false} \text{ (BS-<LISTNIL)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow v_1 :: v_2 \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 \leq e_2 \Downarrow false} \text{ (BS-}\leq\text{LISTNIL)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow v_1 :: v_2 \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 > e_2 \Downarrow true} \quad \text{(BS->ListNil)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow v_1 :: v_2 \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 \geq e_2 \Downarrow true} \quad \text{(BS-}\geq\text{ListNil)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow nil \qquad \text{env} \vdash e_2 \Downarrow v_1 :: v_2}{\text{env} \vdash e_1 < e_2 \Downarrow true} \quad \text{(BS-<NilList)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow nil \qquad \text{env} \vdash e_2 \Downarrow v_1 :: v_2}{\text{env} \vdash e_1 \leq e_2 \Downarrow true} \quad \text{(BS-}\leq\text{NilList)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow nil \qquad \text{env} \vdash e_2 \Downarrow v_1 :: v_2}{\text{env} \vdash e_1 > e_2 \Downarrow false} \quad \text{(BS->NilList)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow nil \qquad \text{env} \vdash e_2 \Downarrow v_1 :: v_2}{\text{env} \vdash e_1 \geq e_2 \Downarrow false} \quad \text{(BS-}\geq\text{NilList)}$$

$$\frac{\begin{array}{cc} \text{env} \vdash e_1 \Downarrow v_1 :: v_2 & \text{env} \vdash e_2 \Downarrow v_3 :: v_4 \\ \text{env} \vdash v_1 = v_3 \Downarrow false & \text{env} \vdash v_1 \; opIneq \; v_3 \Downarrow b \end{array}}{\text{env} \vdash e_1 \; opIneq \; e_2 \Downarrow b} \quad \text{(BS-IneqListHead)}$$

$$\frac{\begin{array}{cc} \text{env} \vdash e_1 \Downarrow v_1 :: v_2 & \text{env} \vdash e_2 \Downarrow v_3 :: v_4 \\ \text{env} \vdash v_1 = v_3 \Downarrow true & \text{env} \vdash v_2 \; opIneq \; v_4 \Downarrow b \end{array}}{\text{env} \vdash e_1 \; opIneq \; e_2 \Downarrow b} \quad \text{(BS-IneqListTail)}$$

Logical Operations The logical operators ∧ (AND) and ∨ (OR) both have a short-circuit evaluation. This means that, if the result of the operation can be determined from the first operand, the second one is not evaluated.

$$\frac{\text{env} \vdash e_1 \Downarrow true}{\text{env} \vdash e_1 \vee e_2 \Downarrow true} \quad \text{(BS-}\vee\text{Short)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow false \qquad \text{env} \vdash e_2 \Downarrow b}{\text{env} \vdash e_1 \vee e_2 \Downarrow b} \quad \text{(BS-}\vee\text{)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow false}{\text{env} \vdash e_1 \wedge e_2 \Downarrow false} \quad \text{(BS-}\wedge\text{Short)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow true \qquad \text{env} \vdash e_2 \Downarrow b}{\text{env} \vdash e_1 \wedge e_2 \Downarrow b} \quad \text{(BS-}\wedge\text{)}$$

Conditional Expression $L1$ supports conditional expressions, which always return a value, but not conditional statements. Because of this, all conditional expressions must have both a *then* and an *else* branch. Evaluation is done only on the condition and the proper branch, avoiding the evaluation of the unused branch.

$$\frac{\text{env} \vdash e_1 \Downarrow \textit{true} \qquad \text{env} \vdash e_2 \Downarrow v}{\text{env} \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \Downarrow v} \qquad \text{(BS-IfTrue)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow \textit{false} \qquad \text{env} \vdash e_3 \Downarrow v}{\text{env} \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \Downarrow v} \qquad \text{(BS-IfFalse)}$$

Function Expressions There are two types of function expressions, each having two variants, with implicit or explicit typing.

The first is a simple unnamed function that takes exactly one parameter. This parameter can occur anywhere inside the function body (e) and will be replaced by the argument when the function is called.

The typed variant specifies only the type of the input, not of the output.

$$\text{env} \vdash \texttt{fn } x : T \Rightarrow e \Downarrow \langle x, e, env \rangle \qquad \text{(BS-Fn)}$$

$$\text{env} \vdash \texttt{fn } x \Rightarrow e \Downarrow \langle x, e, env \rangle \qquad \text{(BS-Fn2)}$$

The second type of function is a recursive function that also takes exactly one parameter (x_2). Unlike the unnamed function, a recursive function also specifies its own name (x_1), such that it can be called within the function body.

The typed variant must specify the types of both the input and output. The type T_1 is the type of the input, and T_2 is the type of the output. In L_1, the programmer specifies the type of the function (that is, $T_1 \rightarrow T_2$), instead of specifying the output type directly.

$$\text{env} \vdash \texttt{rec } x_1 : T_1 \rightarrow T_2 \ \ x_2 : T_1 \Rightarrow e \Downarrow \langle x_1, x_2, e, env \rangle \qquad \text{(BS-Rec)}$$

$$\text{env} \vdash \texttt{rec } x_1 \ x_2 \Rightarrow e \Downarrow \langle x_1, x_2, e, env \rangle \qquad \text{(BS-Rec2)}$$

Application An application expression requires either a closure or a recursive closure for its left-hand operand. The right-hand operand (argument) is always evaluated using the current environment, resulting in a value v_2.

In the case of a simple closure, the body of the function (e) is evaluated using the stored closure, adding an association between the parameter identifier (x) and the argument (v_2).

$$\frac{\text{env} \vdash e_1 \Downarrow \langle x, e, env \rangle \qquad \text{env} \vdash e_2 \Downarrow v_2}{\begin{array}{c} \{x \rightarrow v_2\} \cup \text{env} \vdash e \Downarrow v \\ \hline \text{env} \vdash e_1 \ e_2 \Downarrow v \end{array}} \qquad \text{(BS-AppFn)}$$

In the case of a recursive closure, there are two new associations added to the stored closure. The first is, as with a simple closure, the parameter identifier (x_2) and the argument (v_2). The second is the function identifier (x_1) and the closure itself. This ensures that the function body can call the recursive function again since its closure is included in the environment.

$$\frac{\text{env} \vdash e_1 \Downarrow \langle x_1, x_2, e, env \rangle \qquad \text{env} \vdash e_2 \Downarrow v_2}{\{x_2 \to v_2, x_1 \to \langle x_1, x_2, e, env \rangle\} \cup \text{env} \vdash e \Downarrow v}{\text{env} \vdash e_1 \ e_2 \Downarrow v} \qquad \text{(BS-APPREC)}$$

Let Expressions   These expressions are used to associate an identifier with a specific value, allowing the value to be reused throughout the program. Since $L1$ is a functional language, these are not variables, and the values assigned to an identifier will be constant (unless the same identifier is used in a new *let* expression).

After evaluating the expression that is to be associated to the identifier (that is, e_1), resulting in v, the *let* expression evaluates e_2. For this evaluation, the association of x to v is added to the environment. The result of this evaluation (that is, v_2) is the final result of the evaluation of the entire *let* expression.

There are two variants of the *let* expression, one with explicit typing and one with implicit typing. For the purposes of evaluation, both have the same behavior.

$$\frac{\text{env} \vdash e_1 \Downarrow v \qquad \{x \to v\} \cup \text{env} \vdash e_2 \Downarrow v_2}{\text{env} \vdash \texttt{let } x : T = e_1 \texttt{ in } e_2 \Downarrow v_2} \qquad \text{(BS-LET)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow v \qquad \{x \to v\} \cup \text{env} \vdash e_2 \Downarrow v_2}{\text{env} \vdash \texttt{let } x = e_1 \texttt{ in } e_2 \Downarrow v_2} \qquad \text{(BS-LET2)}$$

Lists   The expression *nil* always evaluates to the value *nil*, which represents an empty list. The append operation (::) accepts any value as its first operand (e_1), but the second operand (e_2) must evaluate to either the empty list (*nil*) or a non-empty list (represented by the value $v_1 :: v_2$).

$$\text{env} \vdash nil \Downarrow nil \qquad \text{(BS-NIL)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow v \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 :: e_2 \Downarrow v :: nil} \qquad \text{(BS-LIST)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow v \qquad \text{env} \vdash e_2 \Downarrow v_1 :: v_2}{\text{env} \vdash e_1 :: e_2 \Downarrow v :: (v_1 :: v_2)} \qquad \text{(BS-LIST2)}$$

The *empty* expression returns *true* if and only if its operand (e_1) evaluates to the empty list (*nil*).

$$\frac{\text{env} \vdash e_1 \Downarrow nil}{\text{env} \vdash \texttt{isempty } e_1 \Downarrow true} \qquad \text{(BS-EMPTYTRUE)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow v_1 :: v_2}{\text{env} \vdash \texttt{isempty } e_1 \Downarrow false} \qquad \text{(BS-EMPTYFALSE)}$$

The *head* expression attempts to obtain the first element of a non-empty list. The *tail* expression is the mirror of the *head* expression, removing the first element of the list and returning the remaining list. If the list evaluates to *nil*, a run-time exception is thrown for both of these expressions.

$$\frac{\text{env} \vdash e_1 \Downarrow v_1 :: v_2}{\text{env} \vdash \texttt{hd } e_1 \Downarrow v_1} \qquad \text{(BS-HEAD)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow nil}{\text{env} \vdash \texttt{hd } e_1 \Downarrow raise} \qquad \text{(BS-HEADEMPTY)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow v_1 :: v_2}{\text{env} \vdash \texttt{tl } e_1 \Downarrow v_2} \qquad \text{(BS-TAIL)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow nil}{\text{env} \vdash \texttt{tl } e_1 \Downarrow raise} \qquad \text{(BS-TAILEMPTY)}$$

Exceptions   Some programs can be syntactically correct but still violate the semantics of the $L1$ language, such as a dividing by zero or trying to access the head of an empty list. In these scenarios, the expression is evaluated as the *raise* value.

Besides violation of semantic rules, the only other expression that evaluates to the *raise* value is the *raise* expression, using the following rule:

$$\text{env} \vdash raise \Downarrow raise \qquad \text{(BS-RAISE)}$$

This value is propagated by (almost) all expressions, climbing up the evaluation tree. This means that, for every evaluation rule, there is an alternative rule that, when a subexpression evaluates to raise, evaluates the whole expression to raise. To avoid cluttering this document with the repetition of rules, these are not shown in their entirety. Below are a few examples of these propagation rules:

$$\frac{\text{env} \vdash e_1 \Downarrow raise}{\text{env} \vdash \texttt{let } x : T = e_1 \texttt{ in } e_2 \Downarrow raise} \qquad \text{(BS-LETRAISE)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow n_1 \qquad \text{env} \vdash e_2 \Downarrow raise}{\text{env} \vdash e_1 + e_2 \Downarrow raise} \qquad \text{(BS-+RAISE)}$$

The only expression that does not propagate the *raise* value is the *try* exception. Its evaluation rules are the following:

$$\frac{\text{env} \vdash e_1 \Downarrow raise \qquad \text{env} \vdash e_2 \Downarrow v}{\text{env} \vdash \texttt{try } e_1 \texttt{ with } e_2 \Downarrow v} \qquad \text{(BS-TRYRAISE)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow v}{\text{env} \vdash \texttt{try } e_1 \texttt{ with } e_2 \Downarrow v} \qquad \text{(BS-TRY)}$$

Sequential Evaluation   The expression $e_1 \, ; \, e_2$ is used when the expression e_1 is only evaluated for its side effects, without any regard for its resulting value.

$$\frac{\text{env} \vdash e_1 \Downarrow skip \qquad \text{env} \vdash e_2 \Downarrow v}{\text{env} \vdash e_1 \, ; \, e_2 \Downarrow v} \qquad \text{(BS-SEQUENTIAL)}$$

The value *skip* can be obtained either by the expression *skip*

$$\text{env} \vdash skip \Downarrow skip \qquad \text{(BS-SKIP)}$$

or by evaluating other expressions (such as *output e*, as shown below).

Input and Output   Since both input and output deal with side effects, specifying their evaluation rules is tricky.

The *output* expression does not evaluate to any significant value, so we represent it with the value *skip*.

$$\frac{\text{env} \vdash e \Downarrow nil}{\text{env} \vdash \texttt{output } e \Downarrow skip} \qquad \text{(BS-OUTPUTEMPTY)}$$

$$\frac{\text{env} \vdash e \Downarrow c :: v_1 \qquad \text{env} \vdash output \, v_1 \Downarrow skip}{\text{env} \vdash \texttt{output } e \Downarrow skip} \qquad \text{(BS-OUTPUT)}$$

This cannot be represented in the semantic rules, but the value *c* that results from evaluating *e* in rule **BS-Output** is printed on the output stream (typically the console). The rule **BS-OutputEmpty** prints a newline character on the output stream, effectively making *output* similar to "'printLine'" found in other languages.

The *input* expression does evaluate to a significant value, but the value is non-deterministic since it depends on the input of a user. The only guarantees that exist are that it will be either *nil* or $c :: v_2$, making this similar to "'readLine'" found in other languages.

$$\text{env} \vdash \texttt{input} \Downarrow v \qquad \text{(BS-INPUT)}$$

1.3 Type System

*L*1 has a strong and static type inference system that checks a program to decide whether or not it is "'well-typed'". If a program is considered to be well-typed, the type system guarantees that the program will be able to be properly evaluated according to the operational semantics of *L*1. As a side-effect of checking the validity of a program, the type system can also provide the actual type of any implicitly typed expression down to its basic types, be those concrete types or variable types.

1.3.1 Polymorphism

*L*1 has support for parametric Damas-Milner polymorphism. This means that functions can have their types be defined with universal quantifiers, allowing their use with any type.

For instance, take the function *count*, which counts the number of elements in a list. This function can be defined as follows:

```
let count = rec count x ⇒ if isempty x then 0 else 1 + count (tl x) in
count (3::4::nil)
```

In this situation, *count* can be used with a list of any type, not only Int. To allow this, its identifier (*count*) must have a universal association in the environment, defined as so:

$\forall x.\ x\ list \rightarrow \text{Int}$

The universal quantifier $\forall x$ allows the type variable x to be substituted for any concrete type when the function is called. When creating a polymorphic type, the type system must identify which type variables are free in the function type and which are bound in the environment. This process guarantees that a polymorphic type only universally quantifies those type variables that are not already bound, while still allowing all free variables to be instantiated when the function is called.

1.3.2 Traits

Traits are characteristics that a type can have, defining behaviors expected of that type. Some expressions are polymorphic in a sense that they accept certain types for their operators, but not any type.

1.3.3 Type Inference System

The type inference system is composed of two basic parts:

- Constraint Collection

- Constraint Unification

Constraints are equations between type expressions, which can have both constant types and variable types. To infer the type of a program, the type system recursively collects a set of constraints for every subexpression in that program. This is done in a static way across the expression tree from the nodes to the root, without having

to evaluate any of the expressions. To create a valid set of constraints, the system must contain an environment, built from the root to the nodes, to ensure identifiers are properly typed.

Environment   Just like the operational semantics, the type system also uses an environment to store information about identifiers. In this case, the environment maps identifiers to type associations. These can be either simple associations or universal associations, which are used for polymorphic functions.

Simple Associations   These associate an identifier with a unique type, which can be either constant or a variable type. When the association is called, the type is returned as-is, even if it is a variable type.

Universal Associations   This association, also called a type scheme, stores a type which contains at least one variable type bound by a "for all" quantifier ($\forall$). When called, this association creates a new variable type for each bound variable and returns a new instance of the type scheme. Universal associations are used exclusively for polymorphic functions.

To create this type of association, the type system must generate a list of "free variables" present in the type that is to be universalized. These are the variable types that are not present in the environment when the identifier is declared. When these free variables are found, they are universally bound. This ensures that only those variable types that are unbound in the environment become universally bound in the resulting association.

Constraint Unification   After collecting every type constraint for the program, the type inference system attempts to unify these constraints and find a solution for them. This solution comes in the form of type substitutions, which associate variable types to other types, and type traits, which associate variable types to sets of traits.

If the constraints cannot be unified - that is, if a conflict is found -, the program is deemed not well-typed. Because of how the collection and unification process works, little information is given about where the problem ocurred.

Unification Application   After obtaining a valid solution to the set of constraints, the type inference system applies the substitution to the type of the program. This is done recursively until no more substitutions are found, resulting in what is called the principal type. If there are any variable types in the principal type, the traits are applied to them, restricting the set of types that the variable types can represent.

Constraint Collection Rules   Every expression in $L1$ has a rule for constraint collection, similar to how every expression has a rule for its semantic evaluation.

$$\Gamma \vdash n : \text{Int} \mid \{\} \qquad \text{(T-Num)}$$

$$\Gamma \vdash b : \text{Bool} \mid \{\} \qquad \text{(T-Bool)}$$

17

$$\Gamma \vdash c : \text{Char} \mid \{\} \qquad\qquad\qquad (\text{T-CHAR})$$

$$\Gamma \vdash skip : \text{Unit} \mid \{\} \qquad\qquad\qquad (\text{T-SKIP})$$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T \mid \{\}} \qquad\qquad\qquad (\text{T-IDENT})$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2}{\Gamma \vdash e_1 + e_2 : \text{Int} \mid C_1 \cup C_2 \cup \{T_1 = \text{Int}; T_2 = \text{Int}\}} \qquad (\text{T-}+)$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2}{\Gamma \vdash e_1 - e_2 : \text{Int} \mid C_1 \cup C_2 \cup \{T_1 = \text{Int}; T_2 = \text{Int}\}} \qquad (\text{T-}-)$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2}{\Gamma \vdash e_1 * e_2 : \text{Int} \mid C_1 \cup C_2 \cup \{T_1 = \text{Int}; T_2 = \text{Int}\}} \qquad (\text{T-}*)$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2}{\Gamma \vdash e_1 \div e_2 : \text{Int} \mid C_1 \cup C_2 \cup \{T_1 = \text{Int}; T_2 = \text{Int}\}} \qquad (\text{T-}\div)$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2 \qquad X_1^{\{Equatable\}} \text{ is new}}{\Gamma \vdash e_1 = e_2 : \text{Bool} \mid C_1 \cup C_2 \cup \{T_1 = T_2; X_1^{\{Equatable\}} = T_2\}} \qquad (\text{T-}=)$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2 \qquad X_1^{\{Equatable\}} \text{ is new}}{\Gamma \vdash e_1 \neq e_2 : \text{Bool} \mid C_1 \cup C_2 \cup \{T_1 = T_2; X_1^{\{Equatable\}} = T_2\}} \qquad (\text{T-}\neq)$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2 \qquad X_1^{\{Orderable\}} \text{ is new}}{\Gamma \vdash e_1 < e_2 : \text{Bool} \mid C_1 \cup C_2 \cup \{T_1 = T_2; X_1^{\{Orderable\}} = T_2\}} \qquad (\text{T-}<)$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2 \qquad X_1^{\{Orderable\}} \text{ is new}}{\Gamma \vdash e_1 \leq e_2 : \text{Bool} \mid C_1 \cup C_2 \cup \{T_1 = T_2; X_1^{\{Orderable\}} = T_2\}} \qquad (\text{T-}\leq)$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2 \qquad X_1^{\{Orderable\}} \text{ is new}}{\Gamma \vdash e_1 > e_2 : \text{Bool} \mid C_1 \cup C_2 \cup \{T_1 = T_2; X_1^{\{Orderable\}} = T_2\}} \qquad (\text{T-}>)$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2 \qquad X_1^{\{Orderable\}} \text{ is new}}{\Gamma \vdash e_1 \geq e_2 : \text{Bool} \mid C_1 \cup C_2 \cup \{T_1 = T_2; X_1^{\{Orderable\}} = T_2\}} \qquad (\text{T-}\geq)$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2}{\Gamma \vdash e_1 \vee e_2 : \text{Bool} \mid C_1 \cup C_2 \cup \{T_1 = \text{Bool}; T_2 = \text{Bool}\}} \quad \text{(T-}\vee\text{)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2}{\Gamma \vdash e_1 \wedge e_2 : \text{Bool} \mid C_1 \cup C_2 \cup \{T_1 = \text{Bool}; T_2 = \text{Bool}\}} \quad \text{(T-}\wedge\text{)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2 \qquad \Gamma \vdash e_3 : T_3 \mid C_3}{\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \text{Bool} \mid C_1 \cup C_2 \cup C_3 \cup \{T1 = \text{Bool}; T_2 = T_3\}} \quad \text{(T-I\textsc{f})}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2 \qquad X_1 \text{ is new}}{\Gamma \vdash e_1 \; e_2 : X \mid C_1 \cup C_2 \cup \{T1 = T_2 \to X_1} \quad \text{(T-A\textsc{pp})}$$

$$\frac{\{x \to T\} \cup \Gamma \vdash e : T_1 \mid C_1}{\Gamma \vdash \texttt{fn } x : T \Rightarrow e : T \to T_1 \mid C_1} \quad \text{(T-F\textsc{n})}$$

$$\frac{X_1 \text{ is new} \qquad \{x \to X_1\} \cup \Gamma \vdash e : T_1 \mid C_1}{\Gamma \vdash \texttt{fn } x \Rightarrow e : X_1 \to T_1 \mid C_1} \quad \text{(T-F\textsc{n}2)}$$

$$\frac{\{x_1 \to (T_1 \to T_2), x_2 \to T_1\} \cup \Gamma \vdash e : T \mid C}{\Gamma \vdash \texttt{rec } x_1 : T_1 \to T_2 \; x_2 : T_1 \Rightarrow e : T_1 \to T_2 \mid C \cup \{T = T_2\}} \quad \text{(T-R\textsc{ec})}$$

$$\frac{X_1 \text{ is new} \qquad X_2 \text{ is new} \qquad \{x_1 \to X_1, x_2 \to X_2\} \cup \Gamma \vdash e : T \mid C}{\Gamma \vdash \texttt{rec } x_1 \; x_2 \Rightarrow e : Y \to T \mid C \cup \{X_1 = X_2 \to T\}} \quad \text{(T-R\textsc{ec}2)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \{x \to T\} \cup \Gamma \vdash e_2 : T_2 \mid C_2}{\Gamma \vdash \texttt{let } x : T = e_1 \texttt{ in } e_2 : T_2 \mid C_1 \cup C_2 \cup \{T = T_1\}} \quad \text{(T-L\textsc{et})}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \{x \to T_1\} \cup \Gamma \vdash e_2 : T_2 \mid C_2}{\Gamma \vdash \texttt{let } x = e_1 \texttt{ in } e_2 : T_2 \mid [\,]} \quad \text{(T-L\textsc{et}2)}$$

$$\frac{X_1 \text{ is new}}{\Gamma \vdash \mathit{nil} : X_1 \; \mathit{list} \mid \{\}} \quad \text{(T-N\textsc{il})}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2}{\Gamma \vdash e_1 :: e_2 : T_1 \ list \mid C_1 \cup C_2 \cup \{T_1 \ list = T_2\}} \qquad \text{(T-List)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad X_1 \ is \ new}{\Gamma \vdash \texttt{isempty} \ e_1 : \text{Bool} \mid C_1 \cup \{T_1 = X_1 \ list\}} \qquad \text{(T-Empty)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad X_1 \ is \ new}{\Gamma \vdash \texttt{hd} \ e_1 : X_1 \mid C_1 \cup \{T_1 = X_1 \ list\}} \qquad \text{(T-Head)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad X_1 \ is \ new}{\Gamma \vdash \texttt{tl} \ e_1 : X_1 \ list \mid C_1 \cup \{T_1 = X_1 \ list\}} \qquad \text{(T-Tail)}$$

$$\frac{X_1 \ is \ new}{\Gamma \vdash raise : X_1 \mid \{\}} \qquad \text{(T-Raise)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2}{\Gamma \vdash \texttt{try} \ e_1 \ \texttt{with} \ e_2 : T_2 \mid C_1 \cup C_2 \cup \{T_1 = T_2\}} \qquad \text{(T-Try)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2}{\Gamma \vdash e_1 \ ; \ e_2 : T_2 \mid C_1 \cup C_2 \cup \{T_1 = \text{Unit}\}} \qquad \text{(T-Sequential)}$$

$$\frac{\Gamma \vdash e : T \mid C}{\Gamma \vdash \texttt{output} \ e : Unit \mid C \cup \{T = \text{Char} \ list\}} \qquad \text{(T-Output)}$$

$$\Gamma \vdash \texttt{input} : \text{Char} \ list \mid \{\} \qquad \text{(T-Input)}$$