Oliver Graf

# The

# Dromedar

## Programming Language

# Contents

# 1 Introduction

## 1.1 Hello, World!

```
fn main -> void
    print_str("Hello, World!")
```

# 2 Typing System

Dromedar uses a static, strong and sound typing system – typing errors cannot happen at runtime. It knows primitive and reference types.

## 2.1 Primitives

There are four primitive types:

- **`int`** represents 64-bit 2's complement (signed) integers.
- **`flt`** represents 64-bit IEEE-754 standard floating point numbers.
- **`char`** represents 8-bit UTF-8 characters.
- **`bool`** represents a 1-bit value: **`true`** and **`false`**.

## 2.2 Reference Types

Reference types represent data structures that are laid over pointers to objects stored in the heap. References come in two types: Maybe-**`null`** and Definitely-not-**`null`** types. Operations like array subscript access are only possible with non-**`null`** types to ensure **`null`** safety.

With a non-null-type `t`, the type `t?` represents a reference type that allows **`null`** values. Primitives are non-nullable.

The following are reference types:

- **`string`** represents lists of characters.
- `[t]` represents an array with elements of type `t`.

Thus, e.g. `[[int]?]` represents a two-dimensional array of integers which is definitely non-**`null`**, whereas its rows may be **`null`**.

## 2.3 Mutability

In general, variables declared with the **`let`** keyword are immutable, whereas **`mut`** declarations create mutable variables.

For references, Dromedar uses a different notion of mutability: Allowing the object to point to new objects is handled with the **`mut`** declaration. However, even immutable objects are allowed to call methods which alter their internal state – for example changing an array's element.

## 2.4 Subtyping

Generally, variables and objects can be assigned values of *subtypes*. Every type is a subtype of itself, and e.g. `t` is a subtype of `t?`. Generally, a subtype is a restricted value set of its supertype – any subtype expression can be assigned to a variable of its supertype.

# 3 Expressions

Because Dromedar has a strong type system, it generally disallows any operations with operands of a non-specified type unless they are explicitly cast to the correct type before. This means that integer and floating point numbers cannot be added, multiplied, etc.

The following table describes precedence and type of all operators:

<table>
<tr><th>Operator</th><th>Name</th><th>Prec.</th><th>Assoc.</th><th>Types</th></tr>
<tr><td>-<br><br>!</td><td>Unary Negation<br><br>Logical Negation</td><td>100</td><td>non-assoc.</td><td>int -> int<br>flt -> flt<br>bool -> bool</td></tr>
<tr><td>**</td><td>Exponentiation</td><td>90</td><td>right</td><td>int,int -> int<br>flt,flt -> flt<br>int,flt -> flt<br>flt,int -> flt</td></tr>
<tr><td>*</td><td>Multiplication</td><td>80</td><td>left</td><td>int,int -> int<br>flt,flt -> flt<br>int,flt -> flt<br>flt,int -> flt</td></tr>
<tr><td>+</td><td>String Addition</td><td>70</td><td>left</td><td>string,string -> string</td></tr>
<tr><td>+,-</td><td>Addition<br>Subtraction</td><td>70</td><td>left</td><td>int,int -> int<br>flt,flt -> flt<br>int,flt -> flt<br>flt,int -> flt<br>char,int -> char<br>int,char -> char</td></tr>
<tr><td><<,>>,>>></td><td>Left Shift<br>Logical Right Shift<br>Arithmetic Right Shift</td><td>60</td><td>left</td><td>int,int -> int</td></tr>
<tr><td>&</td><td>Bitwise And</td><td>60</td><td>left</td><td>int,int -> int</td></tr>
<tr><td>^</td><td>Bitwise Xor</td><td>50</td><td>left</td><td>int,int -> int</td></tr>
<tr><td>|</td><td>Bitwise Or</td><td>40</td><td>left</td><td>int,int -> int</td></tr>
<tr><td>=,!=,>,<,>=,<=</td><td>Comparison</td><td>30</td><td>non-assoc.</td><td>[int/flt] -> bool<br>[char] -> bool</td></tr>
<tr><td>=,!=,>,<,>=,<=</td><td>Structural Comparison</td><td>30</td><td>non-assoc.</td><td>[string]->bool</td></tr>
<tr><td>==,!==</td><td>Reference Comparison</td><td>30</td><td>non-assoc.</td><td>[rt1...rtn]->bool</td></tr>
<tr><td>&&</td><td>Logical And</td><td>20</td><td>left</td><td>bool,bool -> bool</td></tr>
<tr><td>||</td><td>Logical Or</td><td>10</td><td>left</td><td>bool,bool -> bool</td></tr>
</table>

Consider the following example:

The expression `1+18-18+'a'` is well-typed (of type **`char`**) and gets parsed as `((1+18)-18)+'a'` and evaluated to `'b'`. `10 - 0.0` on the other hand is not well-typed as `-` cannot take an **`int`** and a **`flt`** operand as arguments.

Comparison operators work differently to other (binary) operators: Instead of comparing just two expressions, Dromedar allows chaining expressions to create one final boolean value: For example, `1 < 2 != 5 >= 5` holds because every single sub-expression (`1 < 2`, `2 != 5` and `5 >= 5`) holds. Every expression is only evaluated once for its side-effect.

Thus, `A op B op C` is not necessarily semantically equivalent to `A op B && B op C`.

# 4 Handling Whitespace

In order to keep the code simple and easy to look at, Dromedar uses significant whitespace: Blocks of code (such as bodies of **`if`**) statements, are denoted by adding a level of indentation.

Every line is either empty (this includes lines containing only comments), or it contains code.

If a line contains code, the level of its indentation is determined by its relationship to the previous line and its environment:

Two neighboring lines of code within the same block of code must have exactly matching whitespace characters before their respective code starts. If a following line has a deeper level of indentation, it must match the whitespace characters of the previous line and then add a number of additional whitespace characters (space(s) and/or tab(s)).

A line of code can only have a deeper indentation level of one step compared to the previous line. The first line of code is always a global instruction and as such has the lowest level of indentation. If it is indented, this level of indentation corresponds to a baseline indentation that every line of code must share.

Consider this valid example:

```
  global x := 3         # baseline indentation of two spaces
                        # empty line -> indentation doesn't matter
  fn main : args:[string] -> int
          Stdio.println("Hello, World!")  # deeper indentation level
          return 0                        # same indentation level
```

Blocks with the same level of indentation can have different indentation strings, but they must still match their environments, as follows:

```
        if <condition>
              BLOCK A
        else
          BLOCK B
```

The two blocks have a different indentation level, but from context it is still clear that `BLOCK A` is a sub-block of the **`if`**-statement, whereas `BLOCK B` belongs to the **`else`**-statement.

# 5 Buildup of a Program

A program consists of a series of global statements – global variable declarations and function definitions.

## 5.1 Global Declarations

Global variables are all assigned a value at the point of their declaration. This value is evaluated statically – declaration expressions can only contain literals and global variables that were already declared previously.

Functions are also declared globally. They can call each other and themselves recursively within their respective function bodies.

## 5.2 Standard Library

The Dromedar standard library includes the following functions and objects:

### 5.2.1 `Str`: String Operations

| Name | Type | Effect |
|---|---|---|
| `Str.of_int` | **`int`** `->` **`string`** | transforms an integer into a string |
| `Str.of_flt` | **`flt`** `->` **`string`** | transforms a decimal number into a string |

### 5.2.2 `IO`: Standard I/O Operations

Printing operations are always preceded by the `IO` library name.

| Name | Type | Effect |
|---|---|---|
| `IO.print_str` | **`string`** `->` **`void`** | prints a string to the console |
| `IO.print_int` | **`int`** `->` **`void`** | prints an integer to the console |
| `IO.print_flt` | **`flt`** `->` **`void`** | prints a real number to the console |
| `IO.print_char` | **`char`** `->` **`void`** | prints a character to the console |
| `IO.print_bool` | **`bool`** `->` **`void`** | prints *`"true"`* or *`"false"`* to the console |

### 5.2.3 `Util`: Miscellaneous Utility Functions

| Name | Type | Effect |
|---|---|---|
| `Util.randint` | `()` -> **`int`** | random integer from $-2^{6}3$ to $2^{6}3-1$ |
| `Util.randflt` | `()` -> **`flt`** | random floating point number in $[0, 1)$ |

### 5.2.4 `File`: File I/O

| Name | Type | Effect |
|---|---|---|
| `File.readall` | **`string`** -> [**`string`**] | returns all lines from the file with the given input name |

### 5.2.5 `Math` : Mathematical Operations

| Name | Type | Effect |
|---|---|---|
| `Math.sin` | **`flt`** -> **`flt`** | sine function |
| `Math.cos` | **`flt`** -> **`flt`** | cosine function |
| `Math.tan` | **`flt`** -> **`flt`** | tangent function |
| `Math.e` | **`flt`** | Euler's constant |
| `Math.pi` | **`flt`** | $\pi$ constant |

### 5.2.6 `Regex`: Regular Expressions

| Name | Type | Effect |
|---|---|---|
| <**type**> | R | blackbox type that represents a regex automaton |
| compile | **string** -> R? | returns **null** if the compilation to a regex automaton fails |
| matches | (R,**string**) -> **bool** | finds whether a partial match in the given string exists |
| first_match | (R,**string**) -> **string**? | returns the first regex match (if it exists, else **null**) |
| all_matches | (R,**string**) -> [**string**] | finds all matches |

### 5.2.7 `Sys` : System Calls, etc.

| Name | Type | Effect |
|---|---|---|
| cmd | **string** -> **int** | executes a shell command |
| fork | () -> **int** | executes the `fork()` UNIX system call |

### 5.2.8 `Time` : Date and Time Utilities

| Name | Type | Effect |
|---|---|---|
| <**type**> | P | blackbox type representing a point in time |
| <**type**> | D | blackbox type representing a duration |
| clock | () -> **int** | the number of clock ticks since the start of the program |
| time | () -> **int** | returns the UNIX time |
| now | () -> P | the time point representing the point when `now()` was called |
| dt | (P,P) -> D | calculates the time difference between the first and the second time point |
| s | D -> **int** | duration in seconds |
| ms | D -> **int** | duration in milliseconds |
| us | D -> **int** | duration in microseconds |

# 6 Garbage Collection

Dromedar uses an algorithm that combines the mark/sweep and reference counting approaches. It is a *precise* garbage collection algorithm, meaning that it is capable of collecting all non-reachable objects and it will only attempt to GC exactly these (as opposed to conservative GC algorithms like the Boehm-Demers-Weiser garbage collector for C/C++).

Internally, each reference object is stored in a central garbage collection table that counts the numbers of program references that can reach a program, as well as the set of its children.

When a garbage collection run is triggered, the collector will free all objects that are not reachable. An object is deemed **reachable** if either its count of program references is nonzero, or if it is the child of a reachable object.

# 7 Formal Typing Rules

A typing rule takes the following shape:

$$\frac{\text{Hypotheses}}{S, \cdots \vdash_{\mathsf{type}} \texttt{grammar spec}}\ \textsc{name}$$

Here, $S$ represents a list of stacks (resp. a stack) of symbol definitions: $S \in (\mathsf{id} \times \mathsf{type} \times \{c, m\})^n$ for some block depth $n$ at any given point. In global context, $S$ has only one layer. $\mathsf{id}$ corresponds to the set of names that variables can have (related to the Lexer symbol `%Identifier`), $\mathsf{type}$ to the set of types in a given program (related to the Parser symbol `Type`), and $\{c, m\}$ to the mutability of the object: $c$ represents an immutable value (as declared by **`let`**), and $m$ a mutable one (declared by **`mut`**).

Writing $S$ in a proof rule enables access to functions and variables within the same module `M`, whereas $S_{\mathsf{N}}$ corresponds to the context from module `N`.

The symbol $\in$ is defined as follows: $s \in S \Leftrightarrow s$ is contained in *any* layer of $S$, whereas $\in_0$ is true only if the symbol is at the top level of the symbol stack (i.e. defined in the same block). The operator $\cup$ on $S$ adds another binding to the top layer of the stack, whereas $\sqcup$ adds another layer to the stack.

The following are the typing rules for Dromedar programs:

## 7.1 Subtyping Rules

### 7.1.1 Cross-Typing

Cross types are types which aren't related by the subtype relation $\preceq$ but still allow for some typesafe interaction – e.g. by assigning variables of one type to variables of another.

The crosstype relation, denoted by the $\asymp$ operator, commutes.

$$\frac{}{\vdash_{\mathsf{T}} \textbf{int} \asymp \textbf{flt}}\ \textsc{CrossTyIntFlt}, \quad \frac{}{\vdash_{\mathsf{T}} \textbf{flt} \asymp \textbf{int}}\ \textsc{CrossTyFltInt}$$

### 7.1.2 Trivial Rule

$$\frac{}{\vdash_{\mathsf{T}} \texttt{t} \preceq \texttt{t}}\ \textsc{SubTyTrivial}$$

### 7.1.3 References

$$\frac{\vdash_{\mathsf{T}} \texttt{t1} \preceq \texttt{t2}}{\vdash_{\mathsf{T}} \texttt{t1} \preceq \texttt{t2?}}\ \textsc{SubTyRefs}, \quad \frac{\vdash_{\mathsf{T}} \texttt{t1} \preceq \texttt{t2}}{\vdash_{\mathsf{T}} \texttt{t1?} \preceq \texttt{t2?}}\ \textsc{SubTyRefs}$$

### 7.1.4 Arrays

$$\frac{\vdash_{\mathsf{T}} \texttt{t1} \preceq \texttt{t2}}{\vdash_{\mathsf{T}} \texttt{[t1]} \preceq \texttt{[t2]}}\ \textsc{SubTyFuns}$$

### 7.1.5 Functions

$$\frac{\vdash_{\mathsf{T}} \texttt{u1} \preceq \texttt{t1},\ \ldots,\ \vdash_{\mathsf{T}} \texttt{un} \preceq \texttt{tn} \quad \vdash_{\mathsf{T}} \texttt{rt} \preceq \texttt{ru}}{\vdash_{\mathsf{T}} \texttt{(t1,...,tn)->rt} \preceq \texttt{(u1,...,un)->ru}}\ \textsc{SubTyFuncs}$$

### 7.1.6 Subtypes and Supertypes

The functions $\mathsf{subtys}$ and $\mathsf{suptys}$ of $\mathsf{types} \to \mathcal{P}(\mathsf{types})$ compute the sub- and supertype set of the input type, respectively.

They are used – among others – in the ExpLitArr rule.

$$\mathsf{subtys} := \begin{cases} \mathbf{int} & \mapsto \{\mathbf{int}\} \\ \mathbf{flt} & \mapsto \{\mathbf{flt}\} \\ \mathbf{char} & \mapsto \{\mathbf{char}\} \\ \mathbf{bool} & \mapsto \{\mathbf{bool}\} \\ \mathbf{string} & \mapsto \{\mathbf{string}\} \\ \mathtt{t?} & \mapsto \{\mathtt{u}, \mathtt{u?} \mid \mathtt{u} \in \mathsf{subtys}(\mathtt{t})\} \\ \mathtt{[t]} & \mapsto \{\mathtt{[u]} \mid \mathtt{u} \in \mathsf{subtys}(\mathtt{t})\} \\ \mathtt{(t1...tn)\text{->}rt} & \mapsto \{\mathtt{(u1...un)\text{->}st} \mid \mathtt{u1...un} \in \mathsf{suptys}(\mathtt{t1...tn}), \mathtt{st} \in \mathsf{subtys}(\mathtt{rt})\} \end{cases}$$

$$\mathsf{suptys} := \begin{cases} \mathbf{int} & \mapsto \{\mathbf{int}\} \\ \mathbf{flt} & \mapsto \{\mathbf{flt}\} \\ \mathbf{char} & \mapsto \{\mathbf{char}\} \\ \mathbf{bool} & \mapsto \{\mathbf{bool}\} \\ \mathbf{string} & \mapsto \{\mathbf{string}, \mathbf{string}?\} \\ \mathtt{t?} & \mapsto \{\mathtt{u?} \mid \mathtt{u} \in \mathsf{suptys}(\mathtt{t})\} \\ \mathtt{[t]} & \mapsto \{\mathtt{[u]}, \mathtt{[u]?} \mid \mathtt{u} \in \mathsf{suptys}(\mathtt{t})\} \\ \mathtt{(t1...tn)\text{->}rt} & \mapsto \left[ \begin{array}{l} \{\mathtt{(u1...un)\text{->}st}, \mathtt{((u1...un)\text{->}st)?} \\ \quad \mid \mathtt{u1...un} \in \mathsf{subtys}(\mathtt{t1...tn}), \mathtt{st} \in \mathsf{suptys}(\mathtt{rt})\} \end{array} \right. \end{cases}$$

## 7.2 Declared Types

$$\frac{}{T \vdash_{\mathsf{D}} \mathbf{int}}\ \textsc{TDInt}, \quad \frac{}{T \vdash_{\mathsf{D}} \mathbf{flt}}\ \textsc{TDFlt}, \quad \frac{}{T \vdash_{\mathsf{D}} \mathbf{char}}\ \textsc{TDChar}, \quad \frac{}{T \vdash_{\mathsf{D}} \mathbf{bool}}\ \textsc{TDBool}$$

$$\frac{T \vdash_{\mathsf{D}} \mathtt{t}}{T \vdash_{\mathsf{D}} \mathtt{[t]}}\ \textsc{TDArray}, \quad \frac{}{T \vdash_{\mathsf{D}} \mathbf{string}}\ \textsc{TDString}, \quad \frac{\mathtt{id} \in T}{T \vdash_{\mathsf{D}} \mathtt{tid}}\ \textsc{TDNative}$$

$$\frac{T \vdash_{\mathsf{D}} \mathtt{t1},\ \ldots,\ T \vdash_{\mathsf{D}} \mathtt{tn} \quad T \vdash_{\mathsf{D}} \mathtt{rt}}{T \vdash_{\mathsf{D}} \mathtt{(t1,...,tn)\text{->}rt}}\ \textsc{TDFunc}$$

## 7.3 Builtin Operators

Many builtin operators are overloaded, providing functionality for multiple input types.

### 7.3.1 Unary Operators

#### 7.3.1.1 Arithmetic Negation

$$\frac{}{\vdash_{\mathsf{O}} \mathtt{-} :: \mathbf{int}\text{->}\mathbf{int}}\ \textsc{TyUopNegInt}, \quad \frac{}{\vdash_{\mathsf{O}} \mathtt{-} :: \mathbf{flt}\text{->}\mathbf{flt}}\ \textsc{TyUopNegInt}$$

#### 7.3.1.2 Logical Negation

$$\frac{}{\vdash_{\mathsf{O}} \mathtt{!} :: \mathbf{bool}\text{->}\mathbf{bool}}\ \textsc{TyUopNot}$$

### 7.3.2 Binary Operators

#### 7.3.2.1 Power

$$\frac{}{\vdash_{\mathsf{O}} \mathtt{**} :: (\mathbf{int},\mathbf{int})\text{->}\mathbf{int}}\ \textsc{TyBopPowInt}, \quad \frac{}{\vdash_{\mathsf{O}} \mathtt{**} :: (\mathbf{flt},\mathbf{flt})\text{->}\mathbf{flt}}\ \textsc{TyBopPowFlt}$$

$$\frac{}{\vdash_{\mathsf{O}} \mathtt{**} :: (\mathbf{int},\mathbf{flt})\text{->}\mathbf{flt}}\ \textsc{TyBopPowIntFlt}, \quad \frac{}{\vdash_{\mathsf{O}} \mathtt{**} :: (\mathbf{flt},\mathbf{int})\text{->}\mathbf{flt}}\ \textsc{TybopPowFltInt}$$

#### 7.3.2.2 Multiplication, Division, Modulo

$$\frac{}{\vdash_O \texttt{*} :: (\textbf{int},\textbf{int})\texttt{->}\textbf{int}}\ \textsc{TyBopMulInt}, \quad \frac{}{\vdash_O \texttt{*} :: (\textbf{flt},\textbf{flt})\texttt{->}\textbf{flt}}\ \textsc{TyBopMulInt}$$

$$\frac{}{\vdash_O \texttt{*} :: (\textbf{int},\textbf{flt})\texttt{->}\textbf{flt}}\ \textsc{TyBopMulIntFlt}, \quad \frac{}{\vdash_O \texttt{*} :: (\textbf{flt},\textbf{int})\texttt{->}\textbf{flt}}\ \textsc{TybopMulFltInt}$$

$$\frac{}{\vdash_O \texttt{/} :: (\textbf{int},\textbf{int})\texttt{->}\textbf{int}}\ \textsc{TyBopDivInt}, \quad \frac{}{\vdash_O \texttt{/} :: (\textbf{flt},\textbf{flt})\texttt{->}\textbf{flt}}\ \textsc{TyBopDivInt}$$

$$\frac{}{\vdash_O \texttt{/} :: (\textbf{int},\textbf{flt})\texttt{->}\textbf{flt}}\ \textsc{TyBopDivIntFlt}, \quad \frac{}{\vdash_O \texttt{/} :: (\textbf{flt},\textbf{int})\texttt{->}\textbf{flt}}\ \textsc{TybopDivFltInt}$$

$$\frac{}{\vdash_O \texttt{\%} :: (\textbf{int},\textbf{int})\texttt{->}\textbf{int}}\ \textsc{TyBopModInt}$$

#### 7.3.2.3 Addition and Subtraction

-

$$\frac{}{\vdash_O \texttt{+} :: (\textbf{int},\textbf{int})\texttt{->}\textbf{int}}\ \textsc{TyBopAddInt}, \quad \frac{}{\vdash_O \texttt{+} :: (\textbf{flt},\textbf{flt})\texttt{->}\textbf{flt}}\ \textsc{TyBopAddFlt},$$

$$\frac{}{\vdash_O \texttt{+} :: (\textbf{int},\textbf{flt})\texttt{->}\textbf{flt}}\ \textsc{TyBopAddIntFlt}, \quad \frac{}{\vdash_O \texttt{+} :: (\textbf{flt},\textbf{int})\texttt{->}\textbf{flt}}\ \textsc{TyBopAddFltInt}$$

$$\frac{}{\vdash_O \texttt{+} :: (\textbf{int},\textbf{char})\texttt{->}\textbf{char}}\ \textsc{TyBopAddCharR}, \quad \frac{}{\vdash_O \texttt{+} :: (\textbf{char},\textbf{int})\texttt{->}\textbf{char}}\ \textsc{TyBopAddCharL}$$

$$\frac{}{\vdash_O \texttt{+} :: (\textbf{string},\textbf{string})\texttt{->}\textbf{string}}\ \textsc{TyBopAddString}$$

$$\frac{\texttt{t} = \min_{\preceq}(\mathsf{suptys}(\texttt{t1}) \cap \mathsf{suptys}(\texttt{t2}))}{\vdash_O \texttt{+} :: (\texttt{[t1]},\texttt{[t2]})\texttt{->}\texttt{[t]}}\ \textsc{TyBopAddArr}$$

-

$$\frac{}{\vdash_O \texttt{-} :: (\textbf{int},\textbf{int})\texttt{->}\textbf{int}}\ \textsc{TyBopSubInt}, \quad \frac{}{\vdash_O \texttt{-} :: (\textbf{flt},\textbf{flt})\texttt{->}\textbf{flt}}\ \textsc{TyBopSubFlt},$$

$$\frac{}{\vdash_O \texttt{-} :: (\textbf{int},\textbf{flt})\texttt{->}\textbf{flt}}\ \textsc{TyBopSubIntFlt}, \quad \frac{}{\vdash_O \texttt{-} :: (\textbf{flt},\textbf{int})\texttt{->}\textbf{flt}}\ \textsc{TBbopSubFltInt}$$

$$\frac{}{\vdash_O \texttt{-} :: (\textbf{int},\textbf{char})\texttt{->}\textbf{char}}\ \textsc{TyBopSubCharR}, \quad \frac{}{\vdash_O \texttt{-} :: (\textbf{char},\textbf{int})\texttt{->}\textbf{char}}\ \textsc{TyBopSubCharL}$$

#### 7.3.2.4 Shift Operators

-

$$\frac{}{\vdash_O \texttt{<<} :: (\textbf{int},\textbf{int})\texttt{->}\textbf{int}}\ \textsc{TyBopLShift}$$

-

$$\frac{}{\vdash_O \texttt{>>} :: (\textbf{int},\textbf{int})\texttt{->}\textbf{int}}\ \textsc{TyBopBitRShift}$$

-

$$\frac{}{\vdash_O \texttt{>>>} :: (\textbf{int},\textbf{int})\texttt{->}\textbf{int}}\ \textsc{TyBopBitAShift}$$

#### 7.3.2.5 Bitwise Operators

-

$$\frac{}{\vdash_O \texttt{\&} :: (\textbf{int},\textbf{int})\texttt{->}\textbf{int}}\ \textsc{TyBopBitAnd}$$

-

$$\frac{}{\vdash_O \texttt{\^{}} :: (\textbf{int},\textbf{int})\texttt{->}\textbf{int}}\ \textsc{TyBopXor}$$

-

$$\frac{}{\vdash_O \texttt{|} :: (\textbf{int},\textbf{int})\texttt{->}\textbf{int}}\ \textsc{TyBopBitOr}$$

#### 7.3.2.6 Logical Operators

- $$\frac{}{\vdash_{\mathsf{O}} \&\& :: (\mathbf{bool},\mathbf{bool})\text{->}\mathbf{bool}}\ \textsc{TyBopLogAnd}$$

- $$\frac{}{\vdash_{\mathsf{O}} \text{^^} :: (\mathbf{bool},\mathbf{bool})\text{->}\mathbf{bool}}\ \textsc{TyBopLogXor}$$

- $$\frac{}{\vdash_{\mathsf{O}} || :: (\mathbf{bool},\mathbf{bool})\text{->}\mathbf{bool}}\ \textsc{TyBopLogOr}$$

#### 7.3.2.7 Comparison Operators

$$\frac{}{\vdash_{\mathsf{O}} \{=, !=, >, <, >=, <=\} :: (\mathbf{int},\ldots,\mathbf{int})\text{->}\mathbf{bool}}\ \textsc{TyCmpListInt}$$

$$\frac{}{\vdash_{\mathsf{O}} \{=, !=, >, <, >=, <=\} :: (\mathbf{flt},\ldots,\mathbf{flt})\text{->}\mathbf{bool}}\ \textsc{TyCmpListFlt}$$

$$\frac{}{\vdash_{\mathsf{O}} \{=, !=, >, <, >=, <=\} :: (\mathbf{char},\ldots,\mathbf{char})\text{->}\mathbf{bool}}\ \textsc{TyCmpListChar}$$

$$\frac{\vdash_{\mathsf{T}} \mathtt{t1} \preceq \mathbf{string},\ \ldots,\ \vdash_{\mathsf{T}} \mathtt{tn} \preceq \mathbf{string}}{\vdash_{\mathsf{O}} \{=, !=, >, <, >=, <=\} :: (\mathtt{t1},\ldots,\mathtt{tn})\text{->}\mathbf{bool}}\ \textsc{TyCmpListRefStr}$$

$$\frac{\vdash_{\mathsf{T}} \mathtt{t1} \preceq / \succeq \mathtt{t2},\ \ldots,\ \vdash_{\mathsf{T}} \mathtt{t(n\text{-}1)} \preceq / \succeq \mathtt{tn}\quad \mathtt{t1},\ldots,\mathtt{tn}\text{ reference types}}{\vdash_{\mathsf{O}} \{==, !==\} :: (\mathtt{t1},\ldots,\mathtt{tn})\text{->}\mathbf{bool}}\ \textsc{TyCmpListRefVal}$$

## 7.4 Expressions

### 7.4.1 Subtyping Expression Rule

To reduce proof rule size, define the following rules:

$$\frac{S \vdash_{\mathsf{E}} \mathtt{exp} :: \mathtt{t1} \quad \vdash_{\mathsf{T}} \mathtt{t1} \preceq \mathtt{t}}{S \vdash_{\mathsf{ET}} \mathtt{exp} \trianglelefteq \mathtt{t}}\ \textsc{ExpSubLocal},\quad \frac{S \vdash_{\mathsf{G}} \mathtt{exp} :: \mathtt{t1} \quad \vdash_{\mathsf{T}} \mathtt{t1} \preceq \mathtt{t}}{S \vdash_{\mathsf{GT}} \mathtt{exp} \trianglelefteq \mathtt{t}}\ \textsc{ExpSubGlobal}$$

$$\frac{S \vdash_{\mathsf{E}} \mathtt{exp} :: \mathtt{t1},\quad \vdash_{\mathsf{T}} \mathtt{t1} \asymp \mathtt{t}}{S \vdash_{\mathsf{ET}} \mathtt{exp} \bowtie \mathtt{t}}\ \textsc{ExpCrossLocal}\quad \frac{S \vdash_{\mathsf{E}} \mathtt{exp} :: \mathtt{t1},\quad \vdash_{\mathsf{T}} \mathtt{t1} \asymp \mathtt{t}}{S \vdash_{\mathsf{GT}} \mathtt{exp} \bowtie \mathtt{t}}\ \textsc{ExpCrossGlobal}$$

### 7.4.2 Assignability

The rule $\vdash_{\mathsf{A}}$ defines when an expression can be assigned a value; $\vdash_{\mathsf{AC}}$ is equivalent to $\vdash_{\mathsf{A}}$ – except that it also allows immutable objects whose children (e.g. array elements) can still be modified.

$$\frac{(\mathtt{id}, \_, m) \in S}{S \vdash_{\mathsf{A}} \mathtt{id}}\ \textsc{ExpAssnId}\quad \frac{S \vdash_{\mathsf{AC}} \mathtt{e1}}{S \vdash_{\mathsf{A}} \mathtt{e1[e2]}}\ \textsc{ExpAssnSub}$$

$$\frac{(\mathtt{id}, \_, \_) \in S}{S \vdash_{\mathsf{AC}} \mathtt{id}}\ \textsc{ExpAssnId'}\quad \frac{S \vdash_{\mathsf{AC}} \mathtt{e1}}{S \vdash_{\mathsf{AC}} \mathtt{e1[e2]}}\ \textsc{ExpAssnSub'}$$

### 7.4.3 Global Expressions

The rule $\vdash_G$ describes global expressions, which are restricted in a way that they can be computed at compile time. Global variables are also strictly non-**null**.

The following copied rules use $\vdash_G$ instead of $\vdash_E$

#### 7.4.3.1 Literals

- GExpLitInt := ExpLitInt
- GExpLitFlt := ExpLitFlt
- GExpLitChar := ExpLitChar
- GExpLitBool := ExpLitBool

#### 7.4.3.2 Other Rules

- GExpId := ExpId
- GExpUop := ExpUop
- GExpBop := ExpBop
- GExpCmpList := ExpCmpList

Note that global variable declarations cannot feature function calls or **null** declarations.

### 7.4.4 Literals

- $$\frac{}{\vdash_{\mathsf{E}} n :: \textbf{int}} \textsc{ExpLitInt}$$

- $$\frac{}{\vdash_{\mathsf{E}} f :: \textbf{flt}} \textsc{ExpLitFlt}$$

- $$\frac{}{\vdash_{\mathsf{E}} c :: \textbf{char}} \textsc{ExpLitChar}$$

- $$\frac{}{\vdash_{\mathsf{E}} \textbf{true} :: \textbf{bool}} \textsc{ExpLitBoolTrue}, \quad \frac{}{\vdash_{\mathsf{E}} \textbf{false} :: \textbf{bool}} \textsc{ExpLitBoolFalse}$$

- $$\frac{}{\vdash_{\mathsf{E}} s :: \textbf{string}} \textsc{ExpLitString}$$

- In arrays, the typechecker looks for the common subtypes of all array literal elements and looks for the one type `t` that is a supertype of all elements and which is a subtype of all other such subtypes (the minimum of the subtypes given the $\preceq$ relation on types).

  $$\frac{S \vdash_{\mathsf{E}} \mathtt{e1} :: \mathtt{t1},\ \ldots,\ S \vdash_{\mathsf{E}} \mathtt{en} :: \mathtt{tn} \quad \mathtt{t} = \min_{\preceq}(\bigcap_{\mathtt{i}=1}^{\mathtt{n}} \mathsf{suptys}(\mathtt{ti}))}{S \vdash_{\mathsf{E}} \mathtt{[e1,\ldots,en]} :: \mathtt{[t]}} \textsc{ExpLitArr}$$

  Because the graph connecting types and subtypes is a forest of trees, if the intersection of supertypes is nonempty there is a unique solution `t`.

### 7.4.5 Range List

$$\frac{S \vdash_{\mathsf{E}} \mathtt{e1} :: \textbf{int} \quad S \vdash_{\mathsf{E}} \mathtt{e2} :: \textbf{int}}{S \vdash_{\mathsf{E}} \mathtt{[e1\ \%RangeSpecifier\ e2]} :: [\textbf{int}]} \textsc{ExpRangeArr}$$

### 7.4.6 List Comprehension

$$\frac{S \vdash_{\mathsf{E}} \mathtt{l1} :: \mathtt{[t1]}, S_1 := S \cup \{(\mathtt{l1}, \mathtt{t1}, c)\} \vdash_{\mathsf{E}} \mathtt{l2} :: \mathtt{[t2]},\ \ldots,\ S_{n-1} \vdash_{\mathsf{E}} \mathtt{ln} :: \mathtt{[tn]} \quad S_n \vdash_{\mathsf{E}} (\mathsf{e}, \mathsf{c}) :: (\mathsf{t}, \textbf{bool})}{S \vdash_{\mathsf{E}} \mathtt{[e\ :\ x1}\ \textbf{in}\ \mathtt{l1,\ \ldots,\ xn}\ \textbf{in}\ \mathtt{ln\ :\ c]} :: \mathtt{[t]}} \textsc{ExpLC}$$

### 7.4.7 Null

$$\frac{\texttt{t} \text{ is a non-null ref. type}}{S \vdash_{\mathsf{E}} \textbf{null of } \texttt{t} :: \texttt{t?}} \text{ ExpNull}$$

### 7.4.8 Dangerous Dereference

$$\frac{S \vdash_{\mathsf{E}} \mathsf{exp} :: \texttt{t?}}{S \vdash_{\mathsf{E}} \textbf{assert } \texttt{exp} :: \texttt{t}} \text{ ExpDeref}$$

### 7.4.9 Ternary Operator

$$\frac{S \vdash_{\mathsf{E}} \mathsf{c} :: \textbf{bool} \quad S \vdash_{\mathsf{E}} \mathsf{e1} :: \texttt{t1},\ S \vdash_{\mathsf{E}} \mathsf{e2} :: \texttt{t2} \quad \texttt{t} = \min_{\preceq}(\mathsf{suptys}(\texttt{t1}) \cap \mathsf{suptys}(\texttt{t2}))}{S \vdash_{\mathsf{E}} \texttt{?c -> e1 : e2} :: \texttt{t}} \text{ ExpTern}$$

### 7.4.10 Identifiers

$$\frac{(id, \texttt{t}, \_) \in S}{S \vdash_{\mathsf{E}} \texttt{id} :: \texttt{t}} \text{ ExpId}$$

### 7.4.11 Unary Operations

Unary and Binary Operations do not need to do subtype checking, as they operate only on primitives.

$$\frac{S \vdash_{\mathsf{E}} \mathsf{exp} :: \texttt{t1} \quad \vdash_{\mathsf{O}} \mathsf{op} :: \texttt{t1->t}}{S \vdash_{\mathsf{E}} \texttt{op exp} :: \texttt{t}} \text{ ExpUop}$$

### 7.4.12 Binary Operations

$$\frac{S \vdash_{\mathsf{E}} \mathsf{e1} :: \texttt{t1} \quad S \vdash_{\mathsf{ET}} \mathsf{e2} :: \texttt{t2} \quad \vdash_{\mathsf{O}} \mathsf{op} :: \texttt{(t1,t2)->t}}{S \vdash_{\mathsf{E}} \texttt{e1 op e2} :: \texttt{t}} \text{ ExpBop}$$

### 7.4.13 Function Calls

This type rule requires $\texttt{rt} \not\equiv \textbf{void}$ unless otherwise specified in another rule or if the application of the function is only partial.

For an $n$-tuple $L$ over $U$ and a value $v \in U$, let $(L, v)$ correspond to $(L_1, \ldots, L_n, v) \in U^{n+1}$.

$$\frac{S \vdash_{\mathsf{E}} \mathsf{f} :: \texttt{(t1,...,tn)->rt}, \begin{cases} S & \vdash_{\mathsf{ET}} \mathsf{e1} \trianglelefteq \texttt{t1} \vee \mathsf{e1} \bowtie \texttt{t1} \rightsquigarrow () =: L_1 \bigvee \mathsf{e1} \equiv \_ \rightsquigarrow (\texttt{t1}) =: L_1, \\ & \vdots \\ S & \vdash_{\mathsf{ET}} \mathsf{en} \trianglelefteq \texttt{tn} \vee \mathsf{en} \bowtie \texttt{tn} \rightsquigarrow L_{n-1} =: L_n \bigvee \mathsf{e1} \equiv \_ \rightsquigarrow (L_{n-1}, \texttt{tn}) =: L_n \end{cases}}{S \vdash_{\mathsf{E}} \texttt{f(e1,...,en)} :: \textbf{if } L_n = () \textbf{ then } \texttt{rt} \textbf{ else } L_n\texttt{->rt}} \text{ ExpFunc}$$

### 7.4.14 `sprintf` call

Because Dromedar is typesafe, a construct like **`sprintf`** is not possible within the standard framework of the language. Instead, it uses the **`sprintf`** keyword that generates a string from the input string and its arguments.

First, define the rule to decide whether an object type is *printable*:

$$\frac{}{\vdash_{\mathsf{P}} \textbf{int}} \text{ PrtInt}, \quad \frac{}{\vdash_{\mathsf{P}} \textbf{flt}} \text{ PrtFlt}, \quad \frac{}{\vdash_{\mathsf{P}} \textbf{char}} \text{ PrtChar}, \quad \frac{}{\vdash_{\mathsf{P}} \textbf{bool}} \text{ PrtBool}, \quad \frac{}{\vdash_{\mathsf{P}} \textbf{string}} \text{ PrtString}$$

$$\frac{\vdash_{\mathsf{P}} \texttt{t}}{\vdash_{\mathsf{P}} \texttt{[t]}} \text{ PrtArr}$$

$$\frac{0 \leq \texttt{j1} < \texttt{n},\ \ldots,\ 0 \leq \texttt{jm} < \texttt{n} \quad S \vdash_{\mathsf{E}} \mathsf{e1} :: \texttt{t1},\ \ldots,\ S \vdash_{\mathsf{E}} \mathsf{en} :: \texttt{tn} \quad \vdash_{\mathsf{P}} \texttt{t1},\ \ldots,\ \vdash_{\mathsf{P}} \texttt{tn}}{S \vdash_{\mathsf{E}} \textbf{sprintf}\texttt{("...}\{j1\}\texttt{...}\{jm\}\texttt{...", e1,...,en)} :: \textbf{string}} \text{ ExpSprintf}$$

#### 7.4.15 Subscript Access

- $$\frac{S \vdash_{\mathsf{E}} \mathtt{s} :: \textbf{string} \quad S \vdash_{\mathsf{E}} \mathtt{i} :: \textbf{int}}{S \vdash_{\mathsf{E}} \mathtt{s[i]} :: \textbf{char}} \textsc{ExpArrSub}$$

- $$\frac{S \vdash_{\mathsf{E}} \mathtt{e} :: \mathtt{[t]} \quad S \vdash_{\mathsf{E}} \mathtt{i} :: \textbf{int}}{S \vdash_{\mathsf{E}} \mathtt{e[i]} :: \mathtt{t}} \textsc{ExpArrSub}$$

#### 7.4.16 Projection

$$\frac{(\mathtt{f}, \mathtt{t}, \_) \in S_{\mathtt{Id}}}{S \vdash_{\mathsf{E}} \mathtt{Id.f} :: \mathtt{t}} \textsc{ExpProjModule}$$

$$\frac{S \vdash_{\mathsf{E}} \mathtt{e} :: \mathtt{[t]}}{S \vdash_{\mathsf{E}} \mathtt{e.length} :: \textbf{int}} \textsc{ExpProjListLength}$$

#### 7.4.17 Comparison Lists

$$\frac{\vdash_{\mathsf{O}} \mathtt{op1} :: \mathtt{(t0,t1)\text{->}t}, \ \ldots, \ \vdash_{\mathsf{O}} \mathtt{opn} :: \mathtt{(t(n\text{-}1),tn)\text{->}t} \quad S \vdash_{\mathsf{ET}} (\mathtt{e0}, \ldots, \mathtt{en})[\trianglelefteq \mid \bowtie](\mathtt{t0}, \ldots, \mathtt{tn})}{S \vdash_{\mathsf{E}} \mathtt{e0\ op1\ \ldots\ opn\ en} :: \mathtt{t}} \textsc{ExpCmpList}$$

### 7.5 Statements

In the statement typing rules, a statement rule produces a tuple $(S, r)$ where $S$ stands for the newly updated context, and $r \in \{\bot, \top\}$, where $\bot$ means that a statement might not return and $\top$ means that a statement definitely returns.

To prevent potential mistakes, the typechecker prevents statements which are deemed unreachable at compile time. The logical operators $\vee$ and $\wedge$ operate as if $\bot \equiv 0$ and $\top \equiv 1$. A statement creates two such items, one for unreachability due to a **return**; one due to a **break**/**continue** statement. The difference is necessary due to different treatment after the loop body in a **do-while** statement.

The $\top$ or $\bot$ on the LHS of a statement represents whether the current statement is in a block or not (this determines whether **break**/**continue** statements are applicable).

#### 7.5.1 Local Variable Declarations

$$\frac{(\mathtt{id}, \_, \_) \notin_0 S \quad S \vdash_{\mathsf{E}} \mathtt{exp} :: \mathtt{t}}{S, T, \mathtt{rt}, L \vdash_{\mathsf{S}} \textbf{let}\ \mathtt{id\ :=\ exp} \Rightarrow (S \cup (\mathtt{id}, \mathtt{t}, c)), \bot} \textsc{StmtVDeclConst}$$

$$\frac{(\mathtt{id}, \_, \_) \notin_0 S \quad T \vdash_{\mathsf{D}} \mathtt{t} \quad S \vdash_{\mathsf{ET}} \mathtt{exp} \trianglelefteq \mathtt{t} \vee \mathtt{exp} \bowtie \mathtt{t}}{S, T, \mathtt{rt}, L \vdash_{\mathsf{S}} \textbf{let}\ \mathtt{id\text{:}t\ :=\ exp} \Rightarrow (S \cup (\mathtt{id}, \mathtt{t}, c)), \bot} \textsc{StmtVTDeclConst}$$

$$\frac{(\mathtt{id}, \_, \_) \notin_0 S \quad S \vdash_{\mathsf{E}} \mathtt{exp} :: \mathtt{t}}{S, T, \mathtt{rt}, L \vdash_{\mathsf{S}} \textbf{mut}\ \mathtt{id\ :=\ exp} \Rightarrow (S \cup (\mathtt{id}, \mathtt{t}, m)), \bot} \textsc{StmtVDeclMut}$$

$$\frac{(\mathtt{id}, \_, \_) \notin_0 S \quad T \vdash_{\mathsf{D}} \mathtt{t} \quad S \vdash_{\mathsf{ET}} \mathtt{exp} \trianglelefteq \mathtt{t} \vee \mathtt{exp} \bowtie \mathtt{t}}{S, T, \mathtt{rt}, L \vdash_{\mathsf{S}} \textbf{mut}\ \mathtt{id\text{:}t\ :=\ exp} \Rightarrow (S \cup (\mathtt{id}, \mathtt{t}, m)), \bot} \textsc{StmtVTDeclMut}$$

#### 7.5.2 Assignments

$$\frac{S \vdash_{\mathsf{A}} \mathtt{lhs} \quad S \vdash_{\mathsf{E}} \mathtt{lhs} :: \mathtt{t} \quad S \vdash_{\mathsf{ET}} \mathtt{exp} \trianglelefteq \mathtt{t} \vee \mathtt{exp} \bowtie \mathtt{t}}{S, T, \mathtt{rt}, L \vdash_{\mathsf{S}} \mathtt{lhs\ :=\ exp} \Rightarrow S, \bot, \bot} \textsc{StmtAssn}$$

#### 7.5.3 Expression Statements

This rule allows **void** return types for functions for the first rule application in the proof tree above.

$$\frac{S \vdash_{\mathsf{E}} \mathtt{exp} :: \mathtt{t}}{S, \mathtt{rt} \vdash_{\mathsf{S}} \mathtt{exp} \Rightarrow S, \bot} \textsc{StmtExpr}$$

### 7.5.4 **printf** Call

$$\frac{S \vdash_{\mathsf{E}} \texttt{\textbf{sprintf}("...\{\textit{j1}\}...\{\textit{jm}\}...", e1,...,en)} :: \texttt{\textbf{string}}}{S, T, \mathsf{rt}, L \vdash_{\mathsf{S}} \texttt{\textbf{printf}("...\{\textit{j1}\}...\{\textit{jm}\}...", e1,...,en)} \Rightarrow S, \bot, \bot} \text{ StmtPrintf}$$

### 7.5.5 **assert** Call

$$\frac{S \vdash_{\mathsf{E}} \mathsf{exp} :: \texttt{\textbf{bool}}}{S, T, \mathsf{rt}, L \vdash_{\mathsf{S}} \texttt{\textbf{assert} exp} \Rightarrow S, \bot} \text{ StmtAssert}$$

### 7.5.6 If Statements

$$\frac{S \vdash_{\mathsf{E}} \mathsf{c} :: \texttt{\textbf{bool}} \quad S, T, \mathsf{rt}, L \vdash_{\mathsf{S}} \mathsf{b1} \Rightarrow S_1, R_1, B_1 \quad S, T, \mathsf{rt}, L \vdash_{\mathsf{S}} \mathsf{b2} \Rightarrow S_2, R_2, B_2}{S, T, \mathsf{rt}, L \vdash_{\mathsf{S}} \texttt{\textbf{if} c b1 \textbf{else} b2} \Rightarrow S, R_1 \wedge R_2, B_1 \wedge B_2} \text{ StmtIf}$$

### 7.5.7 Checked Null Casts

$$\frac{S \vdash_{\mathsf{E}} \mathsf{exp} :: \texttt{t?} \quad S \cup \{(\mathsf{r}, \mathsf{t}, c)\}, T, \mathsf{rt}, L \vdash_{\mathsf{S}} \mathsf{b1} \Rightarrow S', R_1, B_2 \quad S, T, \mathsf{rt}, L \vdash_{\mathsf{S}} \mathsf{b2} \Rightarrow S'', R_2, B_2}{S, T, \mathsf{rt}, L \vdash_{\mathsf{S}} \texttt{\textbf{denull} r := exp b1 \textbf{else} b2} \Rightarrow S, R_1 \wedge R_2, B_1 \wedge B_2} \text{ StmtNullCast}$$

### 7.5.8 While Statements

$$\frac{S \vdash_{\mathsf{E}} \mathsf{c} :: \texttt{\textbf{bool}} \quad S, T, \mathsf{rt}, \top \vdash_{\mathsf{S}} \mathsf{b} \Rightarrow S', R, B}{S, T, \mathsf{rt}, L \vdash_{\mathsf{S}} \texttt{\textbf{while} c b} \Rightarrow S, \bot, \bot} \text{ StmtWhile}$$

### 7.5.9 Do-While Statements

$$\frac{S \vdash_{\mathsf{E}} \mathsf{c} :: \texttt{\textbf{bool}} \quad S, T, \mathsf{rt}, \top \vdash_{\mathsf{S}} \mathsf{b} \Rightarrow S', R, B}{S, T, \mathsf{rt}, L \vdash_{\mathsf{S}} \texttt{\textbf{do} b \textbf{while} c} \Rightarrow S, R, \bot} \text{ StmtDoWhile}$$

### 7.5.10 For Statements

$$\frac{S \vdash_{\mathsf{E}} \mathsf{estart} :: \texttt{\textbf{int}} \quad S \vdash_{\mathsf{E}} \mathsf{eend} :: \texttt{\textbf{int}} \quad S \cup \{(\mathsf{id}, \texttt{\textbf{int}}, c)\}, T, \mathsf{rt}, \top \vdash_{\mathsf{S}} \mathsf{b} \Rightarrow S', R, B}{S, T, \mathsf{rt}, L \vdash_{\mathsf{S}} \texttt{\textbf{for} id := estart \%RangeSpecifier eend b} \Rightarrow S, \bot, \bot} \text{ StmtFor}$$

$$\frac{S \vdash_{\mathsf{E}} \mathsf{exp} :: \texttt{[t]} \quad S \cup \{(\mathsf{id}, \mathsf{t}, c)\}, T, \mathsf{rt}, \top \vdash_{\mathsf{S}} b \Rightarrow S', R, B}{S, T, \mathsf{rt}, L \vdash_{\mathsf{S}} \texttt{\textbf{for} id \textbf{in} exp b} \Rightarrow S, \bot, \bot} \text{ StmtForIn}$$

### 7.5.11 Break and Continue

$$\frac{}{S, T, \mathsf{rt}, \top \vdash_{\mathsf{S}} \texttt{\textbf{break}} \Rightarrow S, \bot, \top} \text{ StmtBreak}, \quad \frac{}{S, T, \mathsf{rt}, \top \vdash_{\mathsf{S}} \texttt{\textbf{continue}} \Rightarrow S, \bot, \top} \text{ StmtContinue}$$

### 7.5.12 Return Statements

$$\frac{S \vdash_{\mathsf{ET}} \mathsf{exp} \trianglelefteq \mathsf{rt} \vee \mathsf{exp} \bowtie \mathsf{rt}}{S, \mathsf{rt} \vdash_{\mathsf{S}} \texttt{\textbf{return} exp} \Rightarrow S, \top} \text{ StmtReturnExp}, \quad \frac{}{S, T, \texttt{\textbf{void}}, L \vdash_{\mathsf{S}} \texttt{\textbf{return}} \Rightarrow S, \top, \bot} \text{ StmtReturn}$$

### 7.5.13 Blocks

$$\frac{S \sqcup \{\}, \mathsf{rt}, L \vdash_{\mathsf{S}} \mathsf{s1} \Rightarrow S_1, \bot, \bot \quad S_1, T, \mathsf{rt}, L \vdash_{\mathsf{S}} \mathsf{s2} \Rightarrow S_2, \bot, \bot, \ \ldots, \ S_{n-1}, \mathsf{rt}, L \vdash_{\mathsf{S}} \mathsf{sn} \Rightarrow S_n, R, B}{S, \mathsf{rt}, L \vdash_{\mathsf{S}} \texttt{s1 ... sn} \Rightarrow S_n, R, B} \text{ StmtBlock}$$

## 7.6 Global Statements

### 7.6.1 Global Function Declaration

$$\frac{T \vdash_{\mathsf{D}} \mathtt{t1}, \ldots, T \vdash_{\mathsf{D}} \mathtt{tn}, T \vdash_{\mathsf{D}} \mathtt{rt} \quad S \sqcup \{(\mathtt{a1}, \mathtt{t1}, c), \ldots, (\mathtt{an}, \mathtt{tn}, c)\}, T, \mathtt{rt}, \bot \vdash_{\mathsf{S}} \mathtt{b} \Rightarrow S', \top, \bot \text{ (or } \mathtt{rt}{=}\mathbf{void}) \quad \mathtt{a1}, \ldots, \mathtt{an} \text{ distin}}{S, T \vdash_{\mathsf{G}} \mathbf{fn}\ \mathtt{id\ :\ a1:t1,\ \ldots,\ an:tn\ ->\ rt\ b} \Rightarrow S, T}$$

### 7.6.2 Global Variable Declaration

$$\frac{(\mathtt{id}, \_, \_) \notin S \quad S \vdash_{\mathsf{G}} \mathtt{exp} :: \mathtt{t} \quad \mathtt{t} \text{ non-}\mathbf{null}}{S, T \vdash_{\mathsf{G}} \mathbf{global}\ \mathtt{id\ :=\ exp} \Rightarrow S \cup \{(\mathtt{id}, \mathtt{t}, c)\}, T} \text{ GStmtVDeclConst}$$

$$\frac{(\mathtt{id}, \_, \_) \notin S \quad T \vdash_{\mathsf{D}} \mathtt{t} \quad S \vdash_{\mathsf{GT}} \mathtt{exp} \trianglelefteq \mathtt{t} \quad \mathtt{t} \text{ non-}\mathbf{null}}{S, T \vdash_{\mathsf{G}} \mathbf{global}\ \mathtt{id:t\ :=\ exp} \Rightarrow S \cup \{(\mathtt{id}, \mathtt{t}, c)\}, T} \text{ GStmtVDeclConst}$$

$$\frac{(\mathtt{id}, \_, \_) \notin S \quad S \vdash_{\mathsf{G}} \mathtt{exp} :: \mathtt{t} \quad \mathtt{t} \text{ non-}\mathbf{null}}{S, T \vdash_{\mathsf{G}} \mathbf{global\ mut}\ \mathtt{id\ :=\ exp} \Rightarrow S \cup \{(\mathtt{id}, \mathtt{t}, m)\}, T} \text{ GStmtVDeclMut}$$

$$\frac{(\mathtt{id}, \_, \_) \notin S \quad T \vdash_{\mathsf{D}} \mathtt{t} \quad S \vdash_{\mathsf{GT}} \mathtt{exp} \trianglelefteq \mathtt{t} \quad \mathtt{t} \text{ non-}\mathbf{null}}{S, T \vdash_{\mathsf{G}} \mathbf{global\ mut}\ \mathtt{id:t\ :=\ exp} \Rightarrow S \cup \{(\mathtt{id}, \mathtt{t}, m)\}, T} \text{ GStmtVDeclMut}$$

### 7.6.3 Native Declarations

$$\frac{\mathtt{t} \notin T}{S, T \vdash_{\mathsf{G}} \mathbf{native\ type}\ \mathtt{t} \Rightarrow S, T \cup \{\mathtt{t}\}} \text{ GStmtNativeTDecl}$$

$$\frac{}{S, T \vdash_{\mathsf{G}} \mathbf{native\ fn}\ \mathtt{f:\ t1,...,tn\ ->\ rt} \Rightarrow S, T} \text{ GStmtNativeFDecl}$$

$$\frac{(\mathtt{id}, \_, \_) \notin S}{S, T \vdash_{\mathsf{G}} \mathbf{native}\ \mathtt{t\ id} \Rightarrow S \cup \{(\mathtt{id}, \mathtt{t}, c)\}, T} \text{ GStmtNativeVDecl}$$

### 7.6.4 Module Header

$$\frac{}{S, T \vdash_{\mathsf{G}} \mathbf{module}\ \mathtt{M} \Rightarrow S \text{ where } S_{\mathtt{M}} = S \text{ with } \mathtt{M} \text{ as the only active module}, T} \text{ GStmtModule}$$

### 7.6.5 Program

$$\frac{S_0, T_0 \vdash_{\mathsf{G}} \mathtt{gs1} \Rightarrow S_1, T_1 \quad \ldots, \quad S_{n-1}, T_{n-1} \vdash_{\mathsf{G}} \mathtt{gsn} \Rightarrow S_n, T_n}{S_0, T_0 \vdash_{\mathsf{G}} \mathtt{gs1\ \ ...\ \ gsn} \Rightarrow S_n, T_n} \text{ GStmtProgram}$$

## 7.7 Context Buildup

### 7.7.1 Global Function Declarations

$$\frac{(\mathtt{id}, \_) \notin S}{S, T \vdash_{\mathsf{GCF}} \mathbf{fn}\ \mathtt{id\ :\ a1:t1,\ \ldots,\ an:tn\ ->\ rt\ b} \Rightarrow S \cup \{(\mathtt{id}, \mathtt{(t1,...,tn)\text{-}{>}rt}, c)\}, T} \text{ GStmtFCtxtFDecl}$$

$$\frac{(\mathtt{id}, \_) \notin S}{S, T \vdash_{\mathsf{GCF}} \mathbf{native\ fn}\ \mathtt{f\ :\ t1,...,tn\ ->\ rt} \Rightarrow S \cup \{(\mathtt{id}, \mathtt{(t1,...,tn)\text{-}{>}rt}, c)\}, T} \text{ GStmtFCtxtNFDecl}$$

$$\frac{}{S, T \vdash_{\mathsf{GCF}} \mathbf{global}\ \mathtt{X\ :=\ Y} \Rightarrow S, T} \text{ GStmtFCtxtVDecl}$$

$$\frac{}{S, T \vdash_{\mathsf{GCF}} \mathbf{native\ type}\ \mathtt{T} \Rightarrow S, T} \text{ GStmtFCtxtNTDecl}$$

$$\frac{}{S, T \vdash_{\mathsf{GCF}} \mathbf{native}\ \mathtt{t\ id} \Rightarrow S, T} \text{ GStmtFCtxtNVDecl}$$

$$\frac{}{S, T \vdash_{\mathsf{GCF}} \mathbf{module}\ \mathtt{M} \Rightarrow S \text{ where } S_{\mathtt{M}} = S \text{ with } \mathtt{M} \text{ as the only active module}, T} \text{ GStmtFCtxtModule}$$

### 7.7.2 Program: Functions

$$\frac{S_0, T_0 \vdash_{\mathsf{GCF}} \texttt{gs1} \Rightarrow S_1, T_1,\ \ldots,\ S_{n-1}, T_{n-1} \vdash_{\mathsf{GCF}} \texttt{gsn} \Rightarrow S_n, T_n}{S_0, T_0 \vdash_{\mathsf{GC}} \texttt{gs1} \ \ \texttt{...} \ \ \texttt{gsn} \Rightarrow S_n, T_n \text{ with no active module}} \ \textsc{GStmtCtxtFuncs}$$

## 7.8 Rule for Program Typechecking

Let $S^\star$ be the starting context which contains the builtin context. It looks as follows:

$$S^\star := \{\} \sqcup \{(\texttt{id}, \mathtt{t}, c) \ \mid \ \texttt{id} \text{ is builtin with type } \mathtt{t}\}$$

Let maintys be the legal set of types of main functions – one of which must be contained in a program.

$$\mathsf{maintys} := \left\{ \begin{array}{ll} \textbf{void} \ \texttt{->} \ \textbf{void}, & \textbf{void} \ \texttt{->} \ \textbf{int}, \\ \texttt{[}\textbf{string}\texttt{]} \ \texttt{->} \ \textbf{void}, & \texttt{[}\textbf{string}\texttt{]} \ \texttt{->} \ \textbf{int} \end{array} \right\}$$

Then, the program rule Prog shall be:

$$\frac{S^\star, \emptyset \vdash_{\mathsf{GC}} \texttt{prog} \Rightarrow S', T_0 \quad \exists!\ \texttt{ft} \in \mathsf{maintys}:\ (\texttt{main}, \texttt{ft}, c) \in S' \quad S', T_0 \vdash_{\mathsf{G}} \texttt{prog} \Rightarrow S, T_1}{\vdash \texttt{prog}} \ \textsc{Prog}$$

# 8 Formal Grammar

## 8.1 Lexer Grammar

The Lexer Grammar is specified using **regular expressions**:

```
LiteralInt   ::= /[1-9]\d*/
LiteralFlt   ::= /\d+\.\d+/
LiteralChar  ::= /'([^'\\]|(\\[\\nrt']))'/
LiteralBool  ::= /true|false/
LiteralStr   ::= /"([^"\\]|(\\[\\nrt"]))*"/

Identifier   ::= /[a-zA-Z][a-zA-Z0-9_]*/

module       ::= /module/
native       ::= /native/

global       ::= /global/
fn           ::= /fn/
let          ::= /let/
mut          ::= /mut/

type         ::= /type/
int          ::= /int/
flt          ::= /flt/
char         ::= /char/
bool         ::= /bool/
string       ::= /string/
void         ::= /void/

null         ::= /null/
denull       ::= /denull/

of           ::= /of/
in           ::= /in/

if           ::= /if/
elif         ::= /elif/
else         ::= /else/
do           ::= /do/
while        ::= /while/
```

```
for          ::= /for/

break        ::= /break/
continue     ::= /continue/

printf       ::= /printf/
sprintf      ::= /sprintf/
assert       ::= /assert/

return       ::= /return/

Dash         ::= /\-/
Bang         ::= /!/
Star         ::= /\*/
Plus         ::= /\+/
LShift       ::= /<</
RShift       ::= />>/
AShift       ::= />>>/
Bitand       ::= /&/
Xor          ::= /\^/
Bitor        ::= /\|/
Logand       ::= /&&/
Logor        ::= /\|\|/

Equal        ::= /=/
NotEqual     ::= /!=/
Greater      ::= />/
Less         ::= /</
GreaterEq    ::= />=/
LessEq       ::= /<=/

RefEqual     ::= /==/
RefNotEqual  ::= /!==/

Assign       ::= /:=/

Colon        ::= /:/
Arrow        ::= /\->/

Dot          ::= /\./
Comma        ::= /\,/

Dots         ::= /\.\.\./
DotsPipe     ::= /\.\.\|/
PipeDots     ::= /\|\.\./
PipeDotPipe  ::= /\|\.\|/

LParen       ::= /\(/
RParen       ::= /\)/
LBrack       ::= /\[/
RBrack       ::= /\]/

QuestionMark ::= /\?/
```

### 8.2 Parser Grammar

The following grammar specification uses a preceding % for lexer tokens. In order to support human readability of the grammar, it is given in EBNF.

```
Program          ::= { GlobalStatement }

GlobalStatement  ::= GVDeclaration | GFDeclaration | %Module %Identifier | %Native NatDecl

GVDeclaration    ::=  %Global [ %Mut ] %Identifier [ %Colon Type ] %Assign GlobalExpression
GFDeclaration    ::= %Fn %Identifier [ %LParen FArguments %Rparen ] %Arrow ReturnType Block
NatDecl          ::=
```

```
    %Type %Identifier | Type %Identifier
    | %Fn %Identifier [ %LParen FArguments %RParen ] %Arrow ReturnType

FArguments       ::= [ %Identifier %Colon Type { %Comma %Identifier %Colon Type } ]

ReturnType       ::= %Void | Type

Block            ::= { Statement }

Statement        ::=
    VDeclaration | AssignStmt
    | IfStmt | NullCastStmt
    | WhileStmt | DoWhileStmt | ForStmt
    | ExprStmt
    | %Break | %Continue
    | ReturnStmt

VDeclaration     ::= [ %Let | %Mut ] %Identifier [ %Colon Type ] %Assign Expression

AssignStmt       ::= LHS %Assign Expression

IfStmt           ::= %If Expression Block { %Elif Expression Block } [ %Else Block ]

NullCastStmt     ::= %Denull %Id %Assign Expression Block [ %Else Block ]

WhileStmt        ::= %While Expression Block
DoWhileStmt      ::= %Do Block %While Expression
ForStmt          ::=
    %For %Id %Assign Expr RangeSpecifier Expr Block
    | %For %Id %In Expr Block

RangeSpecifier   ::= %Dots | %DotsPipe | %PipeDots | %PipeDotPipe

ExprStmt         ::=
    Expression
    | %Printf %LParen %LiteralStr PrintfArglist %RParen
    | %Assert Expression

ReturnStmt       ::= %Return [ Expression ]

LHS              ::= BaseExpression { Application }

Type             ::= %Int | %Flt | %Char | %Bool | RefType [ %QuestionMark ]

RefType          ::= %String | %LBrack Type %RBrack | %Identifier [ %Dot %Identifier ]

GlobalExpression ::= Expression

Expression       ::=
    %Null %Of RefType
    | %LBrack %RBrack %Of Type
    | %QuestionMark Expression %Arrow Expression %Colon Expression
      | %Assert Expression
    | ExprPrec30

ExprPrec20       ::=
     ExprPrec30 [ (%Equal | %NotEqual | %Greater | %Less | %GreaterEq
         | %LessEq | %RefEqual | %RefNotEqual) ExprPrec20 ]

ExprPrec30       ::= ExprPrec40 [ %Bitor ExprPrec30 ]
ExprPrec40       ::= ExprPrec50 [ %Xor ExprPrec40 ]
ExprPrec50       ::= ExprPrec60 [ %Bitand ExprPrec50 ]
ExprPrec60       ::= ExprPrec70 [ (%LShift | %RShift | %Ashift) ExprPrec60 ]
ExprPrec70       ::= ExprPrec80 [ (%Plus | %Minus) ExprPrec70 ]
ExprPrec80       ::= ExprPrec90 [ %Star ExprPrec80 ]
ExprPrec90       ::= ExprPrec100 [ %StarStar ExprPrec90 ]
```

```
ExprPrec100      ::= SimpleExpression | (%Dash | %Bang) ExprPrec100

SimpleExpression ::= BaseExpression { Application }

BaseExpression   ::=
    %LParen Expression %RParen
    | %LBrack [ Expression { %Comma Expression } ] %RBrack
    | %LBrack Expression RangeSpecifier Expression %RBrack
    | %LBrack Expression %Colon
        [ %Identifier %In Expression { %Comma %Identifier %In Expression } ]
        [ %Colon Expression ] %RBrack
    | %LiteralInt | %LiteralFlt | %LiteralChar | %LiteralBool | %LiteralStr
    | %Identifier
    | %Sprintf %LParen %LiteralStr
        [ Expression { %Comma PrintfArglist } ] %RParen

Application      ::=
    %LParen [ FArg { %Comma FArg } ] %RParen
    | %LBrack Expression %RBrack
    | %Dot Identifier

FArg             ::= Expression | %Underscore
```