Oliver Graf

The

Dromedar

Programming Language

Contents

1 Introduction

Dromedar is a statically typed, garbage collected, *safe* programming language. It is mainly imperative, but combines these constructs with some typically functional language elements like partial application.

2 Language Tutorial

2.1 Hello, World!

Each Dromedar program contains a *main* function that serves as the start point of the execution of the program. The function is denoted by the keyword **fn**, its return type is specified after the arrow: **void** means that the function doesn't return a value.

In order to print *Hello, World!* to the console, the program calls the function *print_str* from the standard library module IO.

```
fn main -> void
    IO.print_str("Hello, World!\n")
```

Using the DROML compiler, the program can be compiled and executed with the following command:

```
droml -o app <FILENAME>.drm
./app
```

2.2 Types

Dromedar knows two families of types: Primitive *value* types and *reference* types. Value types are small types that are stored on the stack, whereas reference types are garbage-collected objects on the heap.

Value types include the following: **bool** (for truth values **true** and **false**), **char** (for 1-byte characters represented in single quotes), **int** (for 64-bit signed integers), and **flt** (64-bit real numbers). Reference types come in two classes: Definitely-not-**null**, and Maybe-**null** types (as denoted by a **?** following the type name).

There are two categories of reference types: Arrays and **string**s. Arrays are denoted by the element type, enclosed in square brackets. For example, [**int**] represents an integer array, whereas [[[**char**]]] stands for a three-dimensional character array. **string** represents a string of characters, **string?** represents values which are either **string** objects, or **null** – a state used for values which do not represent any object.

These types can be nested within arrays: [[[[**string**]?]]?]? is a maybe-**null** array of maybe-**null** arrays containing two-dimensional arrays of maybe-**null** string arrays.

Functions are also handled as reference types.

It is not possible to access objects of maybe-null types in order to ensure null pointer safety. More on maybe-null reference types in later sections in this tutorial.

2.2.1 Subtypes and Crosstypes

Consider the subtype relation $\prec$ and the crosstype commutation relation $\asymp$, as defined in the section on the typing rules.

These relations define which type of values can be assigned to variables of certain types (and passed as function arguments, and so on).

These typing relations are more precisely explained in the typing rule section. Here a quick summary:

Non-**null** types are subtypes of their corresponding maybe-**null** types, e.g. meaning that a non-**null** string expression (such as a string literal) can be assigned to a variable of type **string?**. The same goes for arrays.

Integers and real numbers are cross-types, meaning that it is possible to assign a real number to an integer variable. In that case, the real number is cast to an integer before the assignment takes place.

2.3 Expressions

Dromedar knows the typical mathematical expressions and precedence rules as known by most programming languages in the C family. The precise types of the different operators are explained in detail in the section on the typing rules. Here is a quick summary:

Apart from `+`, `-`, `*`, `/`, `%`, the shifting operators `<<`, `>>`, and `>>>`, the bitwise operators `&`, `|`, `^`, and the logical operators `&&` and `||` using short-circuit evaluation, the language also knows `**` for exponentiation, and `^^` for boolean (instead of bitwise) exclusive or.

Dromedar also allows comparisons using the operators `=`, `!=`, `>`, `>=`, `<`, and `<=`, and allows combining them into comparison chains. For example, `3 < 5 < 7 >= 3` is a legal expression which returns **`true`** since every sub-comparison is true. Each expression in such a chain is only evaluated once and the entire comparison list is short-circuited. The first comparison that is false cuts off the rest of the evaluation, as with the `&&` and `||` operators.

The operators `==` and `!==` do reference comparison: They check whether two references point at the same object, not whether these objects are structurally identical. Reference Equality thus implies structural equality, but not the other way around.

Dromedar also knows the unary operators `-` and `!` for arithmetic and logical negation.

These operators take pairs of the following types: **`flt,flt`** and **`int,int`** for the arithmetic, **`int,int`** for the bitwise, and **`bool,bool`** for the logical operators. The arithmetic operators also allow for mixing of **`flt`** and **`int`**, where such an operation will always cast the integer to a real number and then perform the operation.

Addition and subtraction also allow adding and subtracting from **`char`** values using integers.

The `+` operator is also used for string and array concatenation, the `*` operator can also perform string repetition (e.g. `2 * "hi" ==> "hihi"`).

2.4 Variables, Assignments

2.4.1 Global Variables

Global variables are declared and assigned outside of function bodies. Their assignment expressions are quite restricted: Only value type and string literals are allowed thus far.

These variables are reachable in all code below the point of their initialization.

Global variables are declared using the **`global`** keyword, with the following syntax:

```
global [mut] identifier [: type] := expression
```

Consider the following global variable declarations:

```
global     x      := 3
global     y :int := 7.2
global mut s      := "hi!"
```

The two variables `x` and `y` are immutable constants – their value can never be changed. The string `s` however can be changed to hold any other string reference, as it was declared **`mut`**.

Because the type of `y` was specified **`int`**, the value it was assigned to is cast to an integer before the assignment.

2.4.2 Local Variables

Local Variables are declared using the **`let`** keyword, or just **`mut`** for mutable variables. They can be of any type and the values they are assigned to can be arbitrarily complex expressions. Consider the following local variable declarations:

```
let x            := 9
let y  :flt      := x ** 7
mut s1 :string?  := null
let s2 :string?  := "hi!"
let a  :[[int]?] := [ null, [] ]
```

- `x` is an immutable integer holding the value `7`.
- `y` is a floating point number holding the value `x ** 7 == 4782969`, which is then implicitly cast to a **`flt`** value, `4782969.0`.
- `s1` is a mutable maybe-null **`string`** value which is assigned **`null`**.
- `s2` is an immutable maybe-null string which contains the value `"hi!"`.
- Finally, `a` is an array of maybe-null integer arrays, of which the first value is **`null`** and the second an empty integer list.

2.4.3 Of `null` and empty lists

`null` and empty lists are, per se, typeless. However, Dromedar is strongly typed, doesn't know a base type like `Object` in Java™, and it uses type inference in variable declarations.

Thus, in general, the type of **`null`** and the empty list must be specified: **`null of string`** is **`null`** of the type **`string?`**, whereas **`null of [[[int]]]`** is the **`null`** value corresponding to the type **`[[[int]]]?`**. On the other hand, **`[] of int`** is the empty list of **`int`** values, **`[] of [int]`** the empty list of **`[int]`** values (of type **`[[int]]`**).

In some instances however, the type of an empty list or null reference can be ommitted: In a function call where the argument's type is already clear, or in an expression where the type can be strictly inferred: For example, consider the function **`f: [string]? -> string`** and the following code:

```
let s1             := f(null) # infers null of [string]
let s2             := f([])   # infers [] of string
let a  : [[[int]]] := [[]]    # infers [] of [int]
```

2.5 Conditionals

The simplest way of decision making and control flow in Dromedar is the **`if`**-`elif`-**`else`** statement, with the following grammar:

```
if expression
    <BLOCK>
{
elif expression
    <BLOCK>
}
[
else
    <BLOCK>
]
```

The behaviour of this construct follows most other imperative programming languages.

The condition expression must be of type **`bool`** – a result of a boolean function, a comparison chain, or a boolean literal **`true`** resp. **`false`**. The number of `elif` blocks is arbitrary, and the **`else`** block is optional.

If the first condition expression evaluates to **`true`**, the **`if`** block gets executed, the rest is omitted. If it is **`false`**, the next block is looked at. If evaluation reaches the **`else`** branch (by the **`if`** and all `elif` conditions evaluating to **`false`**), it gets executed without first evaluating any condition.

2.6 Checked Casts, Assertions

In general, it is not possible to read (i.e. dereference) maybe-**`null`** variables since such a dereference might be dangerous (if the value held by the variable is, in fact, **`null`**). Using a checked cast expression, it is possible to get access to the value held by a maybe-**`null`** expression if it is not **`null`**, as follows:

```
# get_maybe_string: () -> string?
let s := get_maybe_string()
denull ns := s
    printf("string: {0}\n", ns)
else
    IO.print_str("null")
```

The expression that `ns` is assigned to can be an arbitrary expression of a maybe-**`null`** type. It doesn't have to be a single variable. Thus, the header of the checked cast also could have been **`denull`** `ns := get_maybe_string()`.

The first block is executed if `s` is not **`null`**. In that case, `ns` holds the value of `s` and it can be treated as a definitely-not-**`null`** variable. Note that even in this block, `s` is still a maybe-**`null`** variable.

The **`else`** block gets executed if `s = null`.

This is the only **safe** way to access a maybe-**`null`**'s internal value. However, for ease of use of the language, it is also possible to "dereference" a value using **`assert`**, as follows:

```
let s := get_maybe_string()
let ns := assert s
printf("String: {0}\n", ns)
```

Here, `ns` is assigned the value held by `s`, if `s != null`. If, however, `s = null` were to hold, the program immediately terminates with an error message to the console. Therefore, these dangerous casts are only to be used if the programmer is entirely certain that a maybe-**null** reference is not **null**. An example is the `Regex.compile` function of the standard library which returns a **null** automaton object if the input string is an illegal expression.

However, if the expression is known at compile time, the programmer can be certain that it isn't faulty.

2.7 `assert` Statements

For example to ensure the program is given the proper inputs, **assert** statements can be used (not to be confused with the **assert** expressions mentioned above). They check a boolean condition and terminate if the condition is false with an error message, as follows:

```
let x := 3
assert x > 4
```

This program writes the following error message before aborting:

```
Assertion failure in {(3 > 4)}
Aborting.
```

2.8 Loops, `break` and `continue`

There are four different kinds of loops: **while**, **do-while**, **for** and **for-in**.

2.8.1 `while` Loops

```
while expression
    <BLOCK>
```

The condition expression (of type **bool**) is evaluated every time before the body. If the condition holds, the body is executed – otherwise the loop is terminated and execution continues to the next instruction.

2.8.2 `do-while` Loops

```
do
    <BLOCK>
while expression
```

This construct is very similar to the **while** loop, except that the condition is evaluated after the loop body is executed. That ensures that the loop body is executed at least once.

2.8.3 `for` Loops

```
for i := a %rangespec b
    <BLOCK>
```

This loop executes a number of times as determined by the distance between `a` and `b`, where `i` can be used as the loop variable within the block. It cannot be edited, and it is automatically incremented/decremented at the end of one execution of the code block.

The `%rangespec` element in the grammar specification is one of the following four syntactic elements: `...`, `..|`, `|..` or `|.|`. `...` means that the values of `i` start at `a` and continue executing up to and including `b`. `..|` stops at one value before `b`, `|..` starts at one step after `a` but includes `b`, and finally `|.|` starts at a value after `a` and stops one step before it reaches `b`.

These loops can count upwards and downwards, and the step direction is automatically inferred. The step length is fixed at `1`.

Consider the following six examples:

```
# Example 1
for x := 0 ... 10
    printf("{0}, ", x)

# Example 2
for x := 0 ..| 10
```

```
    printf("{0}, ", x)

# Example 3
for x := 0 |.. 10
    printf("{0}, ", x)

# Example 4
for x := 0 |.| 10
    printf("{0}, ", x)

# Example 5
for x := 10 ..| 0
    printf("{0}, ", x)

# Example 6
for x := 0 |.| 1
    printf("{0}, ", x)
```

They produce the following outputs:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, # Example 1
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,     # Example 2
1, 2, 3, 4, 5, 6, 7, 8, 9, 10,    # Example 3
1, 2, 3, 4, 5, 6, 7, 8, 9,        # Example 4
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,    # Example 5
                                  # Example 6
```

The loop boundary expressions `a` and `b` must be of type **`int`** (or **`flt`** which is a crosstype of **`int`**, in which case the expressions are first cast to **`int`** before they're evaluated). The bounds are also only evaluated once before the first loop iteration.

2.8.4 `for-in` Loops

```
for x in l
<BLOCK>
```

These loops are similar to `foreach` constructs in other languages like C#.

They take a list `l` and then iterate over all elements of the list, calling them `x`. These variables `x` are, again, immutable. The list expression is also only evaluated before the first iteration.

Consider the following example and its output:

```
for val in [1,2,17,-9]
    IO.print_int(val)
```

```
1
2
17
-9
```

2.8.5 `break` and `continue`

It is possible to steer the flow of the program by either exiting the current loop using **`break`** or skipping the rest of the loop body using **`continue`**. If **`continue`** is used, the loop condition is evaluated again after potentially updating some loop variables (which is the case in a **`for`** or **`for-in`** loop).

Consider the following example:

```
for i := 1 ... 10
    printf("Loop Iteration {0}\n", i)
    if i > 7
        break
    elif i > 5
        continue
```

```
    else
        printf("Value: {0}\n", 2 ** i)
```

This code will produce the following output:

```
Loop Iteration 1
Value: 2
Loop Iteration 2
Value: 4
Loop Iteration 3
Value: 8
Loop Iteration 4
Value: 16
Loop Iteration 5
Value: 32
Loop Iteration 6
Loop Iteration 7
```

2.9 Functions

2.9.1 Function Definitions

Functions are defined in global scope using the **fn** keyword. They are immutable.

Their definition uses a name, a list of typed parameters and a return type, followed by a function body.

If a function has a return type, it must contain at least one **return** statement, and at least one **return** statement must be reached by every possible (terminating) flow of execution – otherwise, the compiler will return with an error.

The following example function computes the scalar product of two real valued vectors:

```
fn scalar_prod (a : [flt], b : [flt]) -> flt
    assert a.length = b.length
    mut res:flt := 0
    for i := 0 ..| a.length
        res := res + a[i] * b[i]
    return res
```

2.9.2 Function Calls

Functions can be called either as expression statements (the function call as a statement), or as part of an expression – provided the function is of type non-**void**.

Consider the following example:

```
let x := scalar_prod ([1,2,3], [1,2,3])
IO.print_int (x)
```

The first function call is used in an expression. It returns `14` according to the function definition in the code example above. The second function call (to `IO.print_int`) results in the value of `x` being printed to the console.

2.9.3 Partial Function Application

Functions can be partially applied (i.e. passing not all arguments to the function to create a new function). This new functor can then be used as a reference object, and it can be called.

Consider the following function definition:

```
fn add (x:int, y:int, z:int) -> int
    return x + y + z
```

Now, this function can be called (for example `add(2,3,4) ==> 9`), but not all arguments can be passed to it: `add(1,2,_)` creates a new function which takes one integer and returns a new integer. The following code shows this in more detail:

```
let f1 := add(1,2,_)
IO.print_int(f1(3))    # f1(3) === add(1,2,3)

let f2 := add(_,6,_)
IO.print_int(f2(1,-9)) # f2(1,-9) === add(1,6,-9)

let f3 := f2(_,-5)
IO.print_int(f3(0))    # f3(0) === f2(0,-5) === add(0,6,-5)
```

2.10 Strings, Arrays, Range Lists, List Comprehensions

2.10.1 Array Literals

Array literals come in three shapes: **Value Lists**, **Range Lists**, and **List Comprehensions**. They are all enclosed by square brackets `[]`.

2.10.1.1 Value Lists

Value lists are simply values separated by commas: `[1,2,3]` creates an array of type `[int]` with three elements: 1, 2 and 3. `['a','b','z']` creates a `[char]` object with three elements.

In such a value list, the type of the array is the least common supertype in the $\prec$ relation of subtypes. For example, `["hi", null of string]` is of type `string?` since `string?` is both a supertype of `string` (corresponding to the string literal element), and of `string?` (itself, corresponding to the `null` element).

2.10.1.2 Range Lists

Range lists can create integer lists in given bounds, using a start and end value expression, linked by a range specifier (`...`, `..|`, `|..`, and `|.|`, as in the `for` statement).

For example, `[1 ..| 10] === [1,2,3,4,5,6,7,8,9]`. Note that range lists can only create integer lists.

2.10.1.3 List Comprehensions

List comprehensions are the most involved way of creating arrays. A list comprehension expression has the following grammar:

```
"[" expr : x in expr, ..., z in expr [ : expr ] "]"
```

Between the two colons, a number of variables can be defined. They iterate over the lists they are linked to by the `in` keyword. The optional expression after the second colon is a condition expression. If it evaluates to `true` (given the current variable values), the expression to the left of the list comprehension expression is evaluated and added to the list, the end result being returned.

Consider the following example:

```
let a := [ 2*x + y  :  x in [1,2,3], y in [4,5]  :  x%y = 0 ]
```

In pseudocode, this list expression gets evaluated as follows:

```
res <- []
for all x in [1,2,3] do
    for all y in [4,5] do
        if x%y = 0 then
            add 2*x+y to res
return res
```

This allows us to create very elegant programs to – for example – sort lists, or find prime numbers:

```
fn primes (limit:int) -> [int]
    assert limit > 0
    return [ x : x in [1...limit] : [ y : y in [1...x] : x%y=0 ].length = 2 ]

fn sort (l:[int]) -> [int]
```

```
    if l.length <= 1
        return l
    else
        let less := sort([ x : x in l : x < l[0] ])
        let more := sort([ x : x in l : x > l[0] ])
        return less + [l[0]] + more
```

Note that the `sort` expression can be written in a single line using the `?->:` ternary operator, but for better readability the code was broken up into multiple lines.

2.11 Standard Library, `sprintf` and `printf`

The standard library is a series of functions and values that are automatically included when a Dromedar program is compiled. It is explained in more detail in later sections.

One specialty of Dromedar is its use of **native types**: They describe *blackbox objects* which are used by standard library functions. They cannot be "opened" by Dromedar programs themselves and must be operated on using native functions. They are used e.g. by the `Regex` module of the standard library, where the type `Regex.R` corresponds to a finite automaton used for regular expression evaluation.

However, the entirety of the `Regex` module is coded in C++ using the `<regex>` module of C++'s standard library, and `Regex.R` corresponds to `std::regex*`. The `Regex` module allows the Dromedar program to hold these automaton pointers in order to pass them to functions and in order to avoid having to recompile a regular expression every time a match is looked for.

The following example of regular expressions shows the use of blackbox native types:

```
let r := Regex.compile("\\d+")
denull r := r
    let matches := Regex.all_matches(r, "123 45.67")
    printf("{0}\n", matches)
```

Since `Regex.compile` returns an object of type `Regex.R?`, a checked cast must be made first. The output generated by that above code snipped is:

```
[123,45,67]
```

2.12 Garbage Collection

Reference objects in Dromedar programs are linked to its garbage collector. It mixes the reference counting and mark-sweep approaches for a very simple design – the actual implementation of the garbage collector requires less than 50 lines of code – with the capability to collect any structures, even circular ones[1].

2.13 Compiling Multiple Files, Modules

One can compile multiple Dromedar source files at the same time by adding them all to the compiler command. Modules separate programs into logically connected chunks. Modules have a depth of 1 – there are no nested modules.

Modules are declared using the **`module`** keyword. Every file is automatically in a module with its own filename as the module name (stopping at the first whitespace character, and excluding the `.drm` file extenstion). This logically separates multiple files compiled together.

The following two files define functions which mutually recursively call one another:

```
# file A.drm
fn f (x:int) -> int
    if x = 0
        return 0
    return 2 * B.f(x-1)

# file B.drm
fn f (x:int) -> int
    if x = 0
        return 1
    return 3 * A.f(x-1)
```

[1] Classical reference counting GC algorithms typically have issues with circular structures

The standard library is defined in a single file and features multiple modules. The following code snippet shows the separation of modules:

```
# Standard Library
# ...

module File

native fn readall (string) -> [string]

module Math

native fn sin (flt) -> flt
native fn cos (flt) -> flt
native fn tan (flt) -> flt
native e  : flt
native pi : flt

# ...
```

Note that these functions are all **native** – they aren't implemented in Dromedar, but rather in C/C++.

3 Typing System

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

3.1 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**.

3.2 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**.

3.3 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.

3.4 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.

4 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:

Operator	Name	Prec.	Assoc.	Types
`-`	Unary Negation	100	non-assoc.	**`int -> int`** **`flt -> flt`**
`!`	Logical Negation			**`bool -> bool`**
`**`	Exponentiation	90	right	**`int,int -> int`** **`flt,flt -> flt`** **`int,flt -> flt`** **`flt,int -> flt`**
`*`	Multiplication	80	left	**`int,int -> int`** **`flt,flt -> flt`** **`int,flt -> flt`** **`flt,int -> flt`**
`+`	String Addition	70	left	**`string,string -> string`**
`+,-`	Addition Subtraction	70	left	**`int,int -> int`** **`flt,flt -> flt`** **`int,flt -> flt`** **`flt,int -> flt`** **`char,int -> char`** **`int,char -> char`**
`<<,>>,>>>`	Left Shift Logical Right Shift Arithmetic Right Shift	60	left	**`int,int -> int`**
`&`	Bitwise And	60	left	**`int,int -> int`**
`^`	Bitwise Xor	50	left	**`int,int -> int`**
`|`	Bitwise Or	40	left	**`int,int -> int`**
`=,!=,>,<,>=,<=`	Comparison	30	non-assoc.	`[`**`int/flt`**`] -> `**`bool`** `[`**`char`**`] -> `**`bool`**
`=,!=,>,<,>=,<=`	Structural Comparison	30	non-assoc.	`[`**`string`**`]->`**`bool`**
`==,!==`	Reference Comparison	30	non-assoc.	`[rt1...rtn]->`**`bool`**
`&&`	Logical And	20	left	**`bool,bool -> bool`**
`||`	Logical Or	10	left	**`bool,bool -> bool`**
`?->:`	Ternary Operator	0	non-assoc.	**`bool`**`,t1,t2 -> t`

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`.

The ternary operator takes three expressions: `? cond -> e1 : e2`. If `cond` evaluates to **`true`**, `e1` is returned, otherwise `e2`. `e1` and `e2` must have a common supertype, which is returned by the expression.

5 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.

5.1 Multiline Expressions

Some instructions are too long to write on a single line – e.g. complicated **`if`** conditions. In that case, it is possible to write such instructions or expressions on multiple lines, provided that the indentation levels of the subsequent lines are deeper than that of the first line of the instruction.

The file `Droml/droml.config` specifies the size of a tab character in terms of spaces in order to allow mixing tabs and spaces.

The following is an example of such multiline instructions:

```
fn main -> void
    let p :=
        [ x
        : x in [1...100]
        : [ y : y in [1...x] : x%y = 0 ].length = 2 ]
    printf("{0}\n", p)
```

This program computes the prime numbers from `0` to `100` using list comprehensions, where the list expression is divided into three separate lines.

5.2 Multiple Statements on a Single Line

While some instructions are long enough to require multiple lines to remain readable, some instructions are so short (such as simple assignments), that it enhances the readability of a chunk of code to put multiple statements on a single line. This is enabled by separating instructions with an optional Semicolon **`;`**, e.g. as follows:

```
fn main -> void
    printf("a\n"); printf("b\n") # the semicolon separates the two commands

    if true
        printf("c\n")  # if a line break follows, a semicolon is optional
    else
        printf("d\n"); # this semicolon can be omitted
```

6 Buildup of a Program

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

6.1 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.

6.2 Standard Library

The Dromedar standard library includes the following functions and objects:

6.2.1 `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

6.2.2 `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

6.2.3 `Util`: Miscellaneous Utility Functions

Name	Type	Effect
`Util.randint`	`() -> int`	random integer from -2^63 to 2^63-1
`Util.randflt`	`() -> flt`	random floating point number in $[0, 1)$

6.2.4 `File`: File I/O

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

6.2.5 `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**	π constant

6.2.6 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

6.2.7 Sys : System Calls, etc.

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

6.2.8 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

7 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.

8 Formal Typing Rules

A typing rule takes the following shape:

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

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. id corresponds to the set of names

that variables can have (related to the Lexer symbol `%Identifier`), 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_{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.

These typing rules deal with templates in an informal way: There are formal rules for template type matching, but the information about detemplated functions is not carried around through the rules in order to reduce the size of the rules and improve their readability.

Instead, they are carried around in two global symbols: $F \in \mathsf{id} \times \mathsf{type}$ refers to a set of generic functions. For every application, all templated types must be resolved at compile-time (this includes partial application!). The symbol $R \in \mathsf{idtype}$ refers to a set of resolved generic functions. These are then appended to the handwritten program. This may perhaps require several steps of template resolution, and template resolution may not have a fixpoint. In that case, the compiler will throw an error.

The following is an example of misuse of generics such that no program can be built:

```
fn f (x : <a>) -> <a>
    return f([x])[0] # resolves f's <a> to [<a>] ->
                     # infinite recursive loop in template resolution

fn main -> void
    IO.print_int(f(0))
```

The following are the typing rules for Dromedar programs:

8.1 Subtyping Rules

8.1.1 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.

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

8.1.2 Trivial Rule

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

8.1.3 References

$$\frac{\vdash_{\mathsf{T}} \mathtt{t1} \preceq \mathtt{t2}}{\vdash_{\mathsf{T}} \mathtt{t1} \preceq \mathtt{t2?}}\ \textsc{SubTyRefs}, \qquad \frac{\vdash_{\mathsf{T}} \mathtt{t1} \preceq \mathtt{t2}}{\vdash_{\mathsf{T}} \mathtt{t1?} \preceq \mathtt{t2?}}\ \textsc{SubTyRefs}$$

8.1.4 Arrays

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

8.1.5 Functions

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

8.1.6 Subtypes and Supertypes

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

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

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

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

8.2 Declared Types

$$\frac{}{T \vdash_{\text{D}} \textbf{int}}\ \text{TDInt}, \quad \frac{}{T \vdash_{\text{D}} \textbf{flt}}\ \text{TDFlt}, \quad \frac{}{T \vdash_{\text{D}} \textbf{char}}\ \text{TDChar}, \quad \frac{}{T \vdash_{\text{D}} \textbf{bool}}\ \text{TDBool}$$

$$\frac{T \vdash_{\text{D}} \texttt{t}}{T \vdash_{\text{D}} \texttt{[t]}}\ \text{TDArray}, \quad \frac{}{T \vdash_{\text{D}} \textbf{string}}\ \text{TDString}, \quad \frac{\texttt{id} \in T}{T \vdash_{\text{D}} \texttt{tid}}\ \text{TDNative}$$

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

8.3 Builtin Operators

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

8.3.1 Unary Operators

8.3.1.1 Arithmetic Negation

$$\frac{}{\vdash_{\text{O}} \texttt{-} :: \textbf{int->int}}\ \text{TyUopNegInt}, \quad \frac{}{\vdash_{\text{O}} \texttt{-} :: \textbf{flt->flt}}\ \text{TyUopNegInt}$$

8.3.1.2 Logical Negation

$$\frac{}{\vdash_{\text{O}} \texttt{!} :: \textbf{bool->bool}}\ \text{TyUopNot}$$

8.3.2 Binary Operators

8.3.2.1 Power

$$\frac{}{\vdash_{\text{O}} \texttt{**} :: \texttt{(}\textbf{int,int}\texttt{)->}\textbf{int}}\ \text{TyBopPowInt}, \quad \frac{}{\vdash_{\text{O}} \texttt{**} :: \texttt{(}\textbf{flt,flt}\texttt{)->}\textbf{flt}}\ \text{TyBopPowFlt}$$

$$\frac{}{\vdash_{\text{O}} \texttt{**} :: \texttt{(}\textbf{int,flt}\texttt{)->}\textbf{flt}}\ \text{TyBopPowIntFlt}, \quad \frac{}{\vdash_{\text{O}} \texttt{**} :: \texttt{(}\textbf{flt,int}\texttt{)->}\textbf{flt}}\ \text{TybopPowFltInt}$$

8.3.2.2 Multiplication, Division, Modulo

$$\frac{}{\vdash_{\mathsf{O}} \texttt{*} :: \texttt{(int,int)->int}}\ \textsc{TyBopMulInt},\quad \frac{}{\vdash_{\mathsf{O}} \texttt{*} :: \texttt{(flt,flt)->flt}}\ \textsc{TyBopMulInt}$$

$$\frac{}{\vdash_{\mathsf{O}} \texttt{*} :: \texttt{(int,flt)->flt}}\ \textsc{TyBopMulIntFlt},\quad \frac{}{\vdash_{\mathsf{O}} \texttt{*} :: \texttt{(flt,int)->flt}}\ \textsc{TyBopMulFltInt}$$

$$\frac{}{\vdash_{\mathsf{O}} \texttt{*} :: \texttt{(int,string)->string}}\ \textsc{TyBopMulIntStr},\quad \frac{}{\vdash_{\mathsf{O}} \texttt{*} :: \texttt{(string,int)->string}}\ \textsc{TyBopMulStrInt}$$

$$\frac{}{\vdash_{\mathsf{O}} \texttt{/} :: \texttt{(int,int)->int}}\ \textsc{TyBopDivInt},\quad \frac{}{\vdash_{\mathsf{O}} \texttt{/} :: \texttt{(flt,flt)->flt}}\ \textsc{TyBopDivInt}$$

$$\frac{}{\vdash_{\mathsf{O}} \texttt{/} :: \texttt{(int,flt)->flt}}\ \textsc{TyBopDivIntFlt},\quad \frac{}{\vdash_{\mathsf{O}} \texttt{/} :: \texttt{(flt,int)->flt}}\ \textsc{TybopDivFltInt}$$

$$\frac{}{\vdash_{\mathsf{O}} \texttt{\%} :: \texttt{(int,int)->int}}\ \textsc{TyBopModInt}$$

8.3.2.3 Addition and Subtraction

- $$\frac{}{\vdash_{\mathsf{O}} \texttt{+} :: \texttt{(int,int)->int}}\ \textsc{TyBopAddInt},\quad \frac{}{\vdash_{\mathsf{O}} \texttt{+} :: \texttt{(flt,flt)->flt}}\ \textsc{TyBopAddFlt},$$
  $$\frac{}{\vdash_{\mathsf{O}} \texttt{+} :: \texttt{(int,flt)->flt}}\ \textsc{TyBopAddIntFlt},\quad \frac{}{\vdash_{\mathsf{O}} \texttt{+} :: \texttt{(flt,int)->flt}}\ \textsc{TyBopAddFltInt}$$
  $$\frac{}{\vdash_{\mathsf{O}} \texttt{+} :: \texttt{(int,char)->char}}\ \textsc{TyBopAddCharR},\quad \frac{}{\vdash_{\mathsf{O}} \texttt{+} :: \texttt{(char,int)->char}}\ \textsc{TyBopAddCharL}$$
  $$\frac{}{\vdash_{\mathsf{O}} \texttt{+} :: \texttt{(string,string)->string}}\ \textsc{TyBopAddString}$$
  $$\frac{\texttt{t} = \min_{\preceq}(\mathsf{suptys}(\texttt{t1}) \cap \mathsf{suptys}(\texttt{t2}))}{\vdash_{\mathsf{O}} \texttt{+} :: \texttt{([t1],[t2])->[t]}}\ \textsc{TyBopAddArr}$$
- $$\frac{}{\vdash_{\mathsf{O}} \texttt{-} :: \texttt{(int,int)->int}}\ \textsc{TyBopSubInt},\quad \frac{}{\vdash_{\mathsf{O}} \texttt{-} :: \texttt{(flt,flt)->flt}}\ \textsc{TyBopSubFlt},$$
  $$\frac{}{\vdash_{\mathsf{O}} \texttt{-} :: \texttt{(int,flt)->flt}}\ \textsc{TyBopSubIntFlt},\quad \frac{}{\vdash_{\mathsf{O}} \texttt{-} :: \texttt{(flt,int)->flt}}\ \textsc{TBbopSubFltInt}$$
  $$\frac{}{\vdash_{\mathsf{O}} \texttt{-} :: \texttt{(int,char)->char}}\ \textsc{TyBopSubCharR},\quad \frac{}{\vdash_{\mathsf{O}} \texttt{-} :: \texttt{(char,int)->char}}\ \textsc{TyBopSubCharL}$$

8.3.2.4 Shift Operators

- $$\frac{}{\vdash_{\mathsf{O}} \texttt{<<} :: \texttt{(int,int)->int}}\ \textsc{TyBopLShift}$$
- $$\frac{}{\vdash_{\mathsf{O}} \texttt{>>} :: \texttt{(int,int)->int}}\ \textsc{TyBopBitRShift}$$
- $$\frac{}{\vdash_{\mathsf{O}} \texttt{>>>} :: \texttt{(int,int)->int}}\ \textsc{TyBopBitAShift}$$

8.3.2.5 Bitwise Operators

- $$\frac{}{\vdash_{\mathsf{O}} \texttt{\&} :: \texttt{(int,int)->int}}\ \textsc{TyBopBitAnd}$$
- $$\frac{}{\vdash_{\mathsf{O}} \texttt{\^{}} :: \texttt{(int,int)->int}}\ \textsc{TyBopXor}$$
- $$\frac{}{\vdash_{\mathsf{O}} \texttt{|} :: \texttt{(int,int)->int}}\ \textsc{TyBopBitOr}$$

8.3.2.6 Logical Operators

- $$\frac{}{\vdash_{\mathsf{O}} \texttt{\&\&} :: \texttt{(bool,bool)->bool}}\ \textsc{TyBopLogAnd}$$
- $$\frac{}{\vdash_{\mathsf{O}} \texttt{\^\^} :: \texttt{(bool,bool)->bool}}\ \textsc{TyBopLogXor}$$
- $$\frac{}{\vdash_{\mathsf{O}} \texttt{||} :: \texttt{(bool,bool)->bool}}\ \textsc{TyBopLogOr}$$

8.3.2.7 Comparison Operators

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

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

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

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

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

8.4 Expressions

8.4.1 Subtyping Expression Rule

To reduce proof rule size, define the following rules:

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

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

8.4.2 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.

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

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

8.4.3 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 $\vdash_E$

8.4.3.1 Literals

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

8.4.3.2 Other Rules

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

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

8.4.4 Literals

- $$\frac{}{S \vdash_{\mathsf{E}} n :: \textbf{int}}\ \textsc{ExpLitInt}$$
- $$\frac{}{S \vdash_{\mathsf{E}} f :: \textbf{flt}}\ \textsc{ExpLitFlt}$$
- $$\frac{}{S \vdash_{\mathsf{E}} c :: \textbf{char}}\ \textsc{ExpLitChar}$$
- $$\frac{}{S \vdash_{\mathsf{E}} \textbf{true} :: \textbf{bool}}\ \textsc{ExpLitBoolTrue},\quad \frac{}{S \vdash_{\mathsf{E}} \textbf{false} :: \textbf{bool}}\ \textsc{ExpLitBoolFalse}$$
- $$\frac{}{S \vdash_{\mathsf{E}} s :: \textbf{string}}\ \textsc{ExpLitString}$$
- 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).

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

  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`.

8.4.5 Range List

$$\frac{S \vdash_{\mathsf{E}} \texttt{e1} :: \textbf{int}\quad S \vdash_{\mathsf{E}} \texttt{e2} :: \textbf{int}}{S \vdash_{\mathsf{E}} \texttt{[e1 \%RangeSpecifier e2]} :: \texttt{[}\textbf{int}\texttt{]}}\ \textsc{ExpRangeArrInt},$$
$$\frac{S \vdash_{\mathsf{E}} \texttt{c1} :: \textbf{char}\quad S \vdash_{\mathsf{E}} \texttt{c2} :: \textbf{char}}{S \vdash_{\mathsf{E}} \texttt{[c1 \%RangeSpecifier c2]} :: \texttt{[}\textbf{char}\texttt{]}}\ \textsc{ExpRangeArrChar}$$

8.4.6 List Comprehension

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

8.4.7 Null

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

8.4.8 Dangerous Dereference

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

8.4.9 Ternary Operator

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

8.4.10 Identifiers

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

8.4.11 Unary Operations

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

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

8.4.12 Binary Operations

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

8.4.13 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 $(L_1, \ldots, L_n, v) \in U^{n+1}$.

$$\frac{S\ \ \vdash_{\mathsf{E}} \texttt{f} :: \texttt{(t1,...,tn)->rt}\ \ \mathsf{resolve}(\texttt{t1,...,tn}; \mathsf{types\ of}\ \texttt{e1,...,en})\ \ ' \left\{ \begin{array}{ll} S & \vdash_{\mathsf{ET}} \texttt{e1} \trianglelefteq \texttt{t1} \vee \texttt{e1} \bowtie \texttt{t1} \rightsquigarrow () =: L_1 \bigvee \texttt{e1} \equiv \_ \rightsquigarrow (\texttt{t1}) =: L_1, \\ & \vdots \\ S & \vdash_{\mathsf{ET}} \texttt{en} \trianglelefteq \texttt{tn} \vee \texttt{en} \bowtie \texttt{tn} \rightsquigarrow L_{n-1} =: L_n \bigvee \texttt{e1} \equiv \_ \rightsquigarrow (L_{n-1}, \texttt{tn}) =: L_n \end{array} \right.}{S \vdash_{\mathsf{E}} \texttt{f(e1,...,en)} :: \textbf{if}\ L_n = ()\ \textbf{then}\ \texttt{rt}\ \textbf{else}\ L_n\texttt{->rt}}\ \textsc{Exp}$$

8.4.14 Template Resolution

The function resolve receives a vector of expected and provided types (as used in the rule above). It, quite informally, changes the expected types by matching templates.

```
resolve(t1,...,tn ; p1,...,pn)
    M := mappings of all template types in t1...tn: [ ty -> unknown ]
    for all i = 1...n do
        if p1 exists (for partial f-app) then
            attempt to match templates such that p1 is a subtype of t1*
            update M accordingly if it works, otherwise fail
        else
            skip
        end
    end

    if function with new types doesn't exist in program
     add it to P to be typechecked after replacing all contained generic types
     if that fails, fail
    else
    return
```

This (quite informal) algorithm attempts to assign all templated types `<id>` real types such that all function arguments match. If this is not possible, a matching cannot be made and the template match fails.

The following example shows an example of template resolution:

```
fn f (e1 : <a>, e2 : <a>) -> <a>
    # do something

...

# [string]?       , [string?]
f(null of [string], [null of string])
```

`f` gets passed values of two types: `[string]?` and `[string?]`. These are not directly related by the subtyping relation $\preceq$. However, they have a common supertype `[string?]?`. Because this is the case, `<a>` can be set to that type `[string?]?`, and the template type matching works.

8.4.15 `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*:

$$\frac{}{\vdash_P \textbf{int}}\ \textsc{PrtInt},\quad \frac{}{\vdash_P \textbf{flt}}\ \textsc{PrtFlt},\quad \frac{}{\vdash_P \textbf{char}}\ \textsc{PrtChar},\quad \frac{}{\vdash_P \textbf{bool}}\ \textsc{PrtBool},\quad \frac{}{\vdash_P \textbf{string}}\ \textsc{PrtString}$$

$$\frac{\vdash_P \mathtt{t}}{\vdash_P \mathtt{[t]}}\ \textsc{PrtArr}$$

$$\frac{0 \le \mathtt{j1} < \mathtt{n},\ \ldots,\ 0 \le \mathtt{jm} < \mathtt{n}\quad S \vdash_E \mathtt{e1} :: \mathtt{t1},\ \ldots,\ S \vdash_E \mathtt{en} :: \mathtt{tn}\quad \vdash_P \mathtt{t1},\ \ldots,\ \vdash_P \mathtt{tn}}{S \vdash_E \textbf{sprintf}(\mathtt{"...\{j1\}...\{jm\}..."},\ \mathtt{e1},\ldots,\mathtt{en}) :: \textbf{string}}\ \textsc{ExpSprintf}$$

8.4.16 Subscript Access

- $$\frac{S \vdash_E \mathtt{s} :: \textbf{string}\quad S \vdash_E \mathtt{i} :: \textbf{int}}{S \vdash_E \mathtt{s[i]} :: \textbf{char}}\ \textsc{ExpArrSub}$$
- $$\frac{S \vdash_E e :: \mathtt{[t]}\quad S \vdash_E \mathtt{i} :: \textbf{int}}{S \vdash_E e\mathtt{[i]} :: \mathtt{t}}\ \textsc{ExpArrSub}$$

8.4.17 Projection

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

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

8.4.18 Comparison Lists

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

8.5 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).

8.5.1 Local Variable Declarations

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

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

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

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

8.5.2 Assignments

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

8.5.3 Expression Statements

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

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

8.5.4 `printf` Call

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

8.5.5 `assert` Call

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

8.5.6 If Statements

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

8.5.7 Checked Null Casts

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

8.5.8 While Statements

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

8.5.9 Do-While Statements

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

8.5.10 For Statements

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

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

8.5.11 Break and Continue

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

8.5.12 Return Statements

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

8.5.13 Blocks

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

8.6 Global Statements

8.6.1 Global Function Declaration

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

8.6.2 Global Variable Declaration

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

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

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

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

8.6.3 Native Declarations

$$\frac{\texttt{t} \notin T}{S, T \vdash_{\mathsf{G}} \textbf{native type}\ \texttt{t} \Rightarrow S, T \cup \{\texttt{t}\}}\ \textsc{GStmtNativeTDecl}$$

$$\frac{}{S, T \vdash_{\mathsf{G}} \textbf{native fn}\ \texttt{f: t1,...,tn -> rt} \Rightarrow S, T}\ \textsc{GStmtNativeFDecl}$$

$$\frac{(\texttt{id}, \_, \_) \notin S}{S, T \vdash_{\mathsf{G}} \textbf{native}\ \texttt{t id} \Rightarrow S \cup \{(\texttt{id}, \texttt{t}, c)\}, T}\ \textsc{GStmtNativeVDecl}$$

8.6.4 Module Header

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

8.6.5 Program

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

8.7 Context Buildup

8.7.1 Global Function Declarations

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

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

$$\frac{}{S, T \vdash_{\mathsf{GCF}} \textbf{global}\ \texttt{X := Y} \Rightarrow S, T}\ \textsc{GStmtFCtxtVDecl}$$

$$\frac{}{S, T \vdash_{\mathsf{GCF}} \textbf{native type}\ \texttt{T} \Rightarrow S, T}\ \textsc{GStmtFCtxtNTDecl}$$

$$\frac{}{S, T \vdash_{\mathsf{GCF}} \textbf{native}\ \texttt{t id} \Rightarrow S, T}\ \textsc{GStmtFCtxtNVDecl}$$

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

8.7.2 Program: Functions

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

8.8 Rule for Program Typechecking

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

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

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

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

Then, the program rule Prog shall be:

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

9 Formal Grammar

9.1 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 ::= /\?/
```

9.2 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             ::= BaseType | FType
FType            ::= %LParen [ Type { %Comma Type } ] %Arrow Type
RefType          ::= %String | %LBrack Type %RBrack | %Identifier [ %Dot %Identifier ]
BaseType         ::= %LParen (FType | RefType) %RParen %QuestionMark
                     | %LParen Type %RParen
                     | %Int | %Flt | %Char | %Bool
                     | %Less %Identifier %Greater [ %QuestionMark ]

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
```