15-745 Advanced Optimizing Compilers: L3 Reference
Spring 2006

January 20, 2006

1 Introduction

This document introduces the language *L3*. *L3* is a simple, C-like language that includes support for such features as pointers, heap-allocated arrays, functions, and structure types. As such, it is possible to write interesting (and even useful) test programs in *L3*.

For this course, we will provide three implementations of a complete working compiler for the *L3* language. Implementations are available in Java, OCaml, and SML/NJ. The compiler takes *L3* source programs and translates them into target programs written in Intel x86 assembly language. When given a program that does not conform to the syntax and static semantics specified in this document, the compiler will stop and issue a reasonably clear and accurate error message, exiting with a non-zero exit code. Finally, the target code, when executed, will abort when any action not defined by the operational semantics is attempted (e.g., when an attempt is made to access an array out of bounds).

This document is organized as follows. Section 2 covers setup and usage of the compiler/test infrastructure. In Section 3, we present an overview of the *L3* language, and follow it in Section 4 with some suggestions regarding its implementation. Finally, in Section 5, we give a mathematical specification for *L3* to disambiguate the English definition in the previous sections.

2 Compiler Setup

This section describes the steps needed to install, compile, and use the *L3* compiler and test suite. Please note that these instructions were tested on an SCS facilitized Linux machine (gs4xxx series). Installation requirements for other systems may vary slightly.

2.1 Setup

Begin by unpacking the provided tarball to a convenient directory. For the remainder of this document, we assume that the files have been unpacked to `~/745/` . If this is not the case, you will need to adjust the paths in the following examples. To install the `gc` conservative garbage collector, download the source tarball from *http://www.hpl.hp.com/personal/Hans_Boehm/gc/* and unpack it. In the resulting `gc6.5` directory, run `./configure; make` to configure and build the `gc` libraries. The build process will have created the shared object files in a hidden directory named `.libs` (note the leading period). Make a note of the full path to the `gc6.5` directory, and set the `GCPATH` variable in `~/745/shared/link` to that value.

2.1.1 Java Setup

Ensure that you are running Sun's Java implementation. Executing `java` should return a message beginning with `Usage: java [-options] class [args...]`. If you encounter a message that refers to `gij`, you are running the GNU Project's java interpreter, which is incompatible with the compiler. Install a copy of J2SE from *http://java.sun.com/* and change the first two variables in `~/745/java/Makefile` and `~/745/java/reference/Makefile` to:

```
JCC = /usr/java/jdk1.5.0_06/bin/javac
JAVA = /usr/java/jdk1.5.0_06/bin/java
```

or wherever the `java` and `javac` binaries are located.

2.1.2 OCaml Setup

Ensure that the OCaml header files and libraries are installed on your machine. In `~/745/ocaml/code/Makefile`, set the variable `LIBPATH` to the location of these libraries.

2.1.3 SML/NJ Setup

Ensure that you are running at least SML/NJ v110.54. On a facilitized machine, take the following steps:

- `su`
- Edit `/usr/local/depot/depot.pref.local` and add the following lines to the file:

  ```
  collection.installmethod copy smlnj
  collection.release beta smlnj
  ```

- `dosupdepot`

2.2 Building the Compiler

(in compiler directory): `make`

2.3 Location of Compiler Binaries

- Java: `~/745/java/compile`
- OCaml: `~/745/ocaml/code/compile`
- SML/NJ:`~/745/sml/compile`

2.4 Invoking the Compiler

```
($COMPILER) code.l3 -o a.s
~/745/shared/link a.s ~/745/shared/runtime/*.c
```

Replacing COMPILER with the location of the compiler. This will create an executable named `a.out` that can be run directly.

2.5 Running the Test Suite

```
cd ~/745/l3_test_suite
```

Edit the Makefile, changing the following variables:

1. COMPILE to point to your compiler
2. REFCOMPILE to point to the reference implementation of the compiler in a particular language (in `~/745/reference/java/compile` etc)
3. NAME your andrew id

Then, execute `make` to run all tests.

3 *L3* Summary

This section is designed to serve as a informal reference for the new features present in *L3*. After reading it, you should have a good idea of what the language is about, but be forewarned that *this is not the complete specification for the language*. The formal definitions found in Section 5 supersede the content of this section; however, you will probably find this summary an easier place to start than the dense mathematics of the formal specification.

We have made our best effort to keep the English and formal specifications in agreement where they overlap, but should you find any contradictions you should let us know ASAP so we can clarify intent. (It is normal for the English to be ambiguous in places or missing some information; this is a "feature" of human languages.)

3.1 Variables

L3 has two base types, **int** and **bool**. Structs and arrays are also available, along with pointers. Integer and pointer values are 4 bytes long.

All variables in an *L3* program must be declared before use, using a **var** declaration that binds the name of the variable to its type. Each variable can be declared only once in any particular scope, and names are case-sensitive. (Attempts to redeclare a variable must result in a compile-time error.) The keywords of the language may not be used as variable names. On the other hand, the names used for struct types are allowed to be used as variable names and vice versa.

Variables which are not initialized before use may have any arbitrary value at runtime. The compiler does not produce an error in this situation; however, you may produce a warning if you desire.

3.2 Structs

The examples given in Figure 1 show some valid and invalid struct declarations and uses. Struct types are declared using the **struct** keyword. No two struct types are allowed to have the same name. The field names, however, only have to be unique within each struct type. Recursive structs are permitted, but with two restrictions:

1. No forward references are allowed within struct definitions. Hence, a struct can be recursive only by referring to itself.

```
struct foo {
  a : int;
  b : foo;    // infinite recursion, not allowed
};

struct bar {
  a : int;
  b : bar*;   // cyclic through pointer: ok
};

struct baz {
  t1 : foo*;
  t2 : bar;   // struct fields are ok
};

{
  var x, y : bar;
  x = y;        // compile-time error -- deep copy of structs is not allowed

  x.a = y.a; //
  x.b = y.b; // the correct way to make a copy of a struct
  return 0;
}
```

Figure 1: `struct` examples

2. To avoid infinite recursion, only pointers to a type may be used recursively. So, for example, cyclic struct declarations like `struct foo` in Figure 1 are not allowed, whereas `struct bar` is legal.

A variable of struct type is not an lvalue and may not be used as the target of an assignment statement. In other words, attempting to copy a struct by assigning it directly to another struct variable will result in a compile-time error. Instead, each field must be individually assigned as shown in Figure 1.

When a variable is declared to have struct type, it is *implicitly* allocated. The compiler will need to reserve the necessary memory automatically.

3.3 Safe Pointers

3.3.1 Overview

L3 uses a notion of "safe" pointers in order to provide a means for indirect references. Like C, *L3* allows pointer arithmetic. However, every attempt to dereference a pointer is checked (either statically or dynamically) to ensure that the pointer is still within the bounds of its originally

allocated heap memory. Out-of-bounds dereferences cause the program to halt immediately with a run-time error message and exit with non-zero status.

A variable `p` can be declared to be a pointer to a value of type `<typ>` using the declaration

```
var p : <typ>*;
```

Note that this notation allows pointers to pointers.

The **alloc()** expression creates a pointer to new array of elements in the heap. Its arguments are the number of elements and the element type. All pointers are initialized by default to a special **NULL** value. **NULL** has the type NS (pronounced "nonsense") which is compatible with any pointer type.

3.3.2 Representation

L3 safe pointers are represented as three-word structures of of the form $\langle base, offset, size \rangle$, where *base* is the memory address of the first element, *offset* is the index into the array of elements, and *size* is the total number of elements. The two keywords **offset()** and **size()** return the current offset in the array and the total number of elements of the array, respectively. They operate only on pointers, and attempting to use them on values of any other type is a compile-time error.

3.3.3 Safety

Safe pointers are thusly named because the compiler inserts checks to insure that pointer dereferences only occur in valid memory. This is the case iff $0 \leq offset < size$. Then, the address may be calculated using the equation

```
address = base + offset * eltsize
```

If a pointer is not valid, dereferencing the pointer will cause a run-time pointer error. The run-time system provides a function `_l3_error()` with the following C-prototype:

```
void _l3_error(const char *srcfile, int line, const char *errmsg);
```

The first and third argument to this function are allowed to be 0 (C's **NULL**). If both are 0, the program silently aborts with a non-zero error code. Otherwise, an error is printed before aborting the program. If `srcfile` is 0, the `line` argument is ignored.

Safe pointers are only checked for safety during dereferencing. Neither pointer arithmetic nor **offset()** or **size()** require that the pointer be valid.

When created, all pointers will be initialized (by the compiler) to **NULL** in order to guarantee the safety of the language. This includes cases such as an array of pointers and pointer fields within structs.

3.3.4 Operations

Two pointers can be compared for equality using the `*==` and `*!=` operators. Two pointers p_1 and p_2 are equal iff base(p_1) == base(p_2) and offset(p_1) == offset(p_2).

The two operators `*+` and `*-` are used to modify the offset of a pointer. Pointer arithmetic is performed without any bounds checking—invalid pointers cause no problem unless they are dereferenced.

```
void main()
{
  var p, q : int*;
  p = alloc(10, int);
  q = &p[4];            // Note that p[x] is equivalent to *(p *+ x).

  if(q[-3] == p[1])
   return 1;

  return 0;
}
```

Figure 2: q points into the same array as p, the result should be `1`

NULL is not influenced by pointer operations—that is, `NULL *+ 5 *== NULL` for example. Note that this does not prevent you from having non-identical representations of **NULL**, so long as `*==` understands that they are the same and dereferencing **NULL** always produces a runtime error.

3.3.5 Referencing

Taking the address of a variable requires that the variable's value be stored in memory. Appel's analysis of escaping variables gives criteria on whether a variable's value may be held in a register or must be stored on the heap.

In the case `&*p`, where the address of a dereferenced pointer is returned, the result is a safe pointer pointing to the same array as `p` did (and not only to the one element to which `p` points). See Figure 2.

3.4 Memory allocation

The run-time system provides the function `_alloc()` with the following C-prototype:

```
void* _alloc(unsigned int nbytes);
```

The compiler will emit code that calls this function to allocate any new memory. As the function returns a raw system pointer, if it was called in response to an **alloc()** expression, the compiler will create a safe pointer for that memory to return to the user. There is no corresponding `free()`—the standard runtime includes a conservative garbage collector that collects dead memory.[1] Additionally, any expression of integer type can be used as a parameter, leading to the problem that the program might request a negative amount of memory from the operating system. The compiler will emit guard code to catch this occurrence, terminating with a run-time error.

3.5 Functions

L3 includes functions whose behavior is very similar to C and Java. There are a few important characteristics worth noting, however:

[1] See *http://www.hpl.hp.com/personal/Hans_Boehm/gc/*

- Functions returning a result are required to have a `return` on every code path. We use a conservative approach and require the compiler to stop with an error message if there exists any code path that reaches the end of the function without executing a `return` statement, regardless of whether it is a possible dynamic execution path.
- For void functions (procedures), a `return;` statement is implicitly included as the last statement and is therefore optional in the body.
- Every `return` statement must return a value of the type specified by the function.
- *L3* allows the programmer to return a pointer to a local variable from a function. In such a case, the variable must be stored on the heap in order to guarantee the availability of the variable after the function returns.
- The entry procedure `main` is not allowed to have any parameters and its return type must be `void`.
- All functions must have distinct names, however, struct-type names, variable names and function names live in different namespaces. It is legal to declare a struct-type `foo`, a function named `foo` and a variable `foo`.
- All variable names in a function must be distinct, but as usual, every function has its own scope and variable names do not interfere across functions.
- Functions are globally visible. A function `foo` may call a function `bar` that is declared after `foo` in the source program.
- The compiler will emit code that follows the standard C calling convention when calling a function/procedure. This is critical, because most test programs will call foreign functions which are written in C. Function declarations with the keyword **foreign** in place of the body are prototypes for external functions, and the compiler will use that information to ensure all calls to that function are correct. Calls to undeclared functions are naturally a compile-time error.

3.5.1 Register Usage Conventions

Code emitted by the compiler follows standard x86 calling conventions. Specifically, registers **%ebp**, **%esp**, **%ebx**, **%esi**, **%edi** must have their original values prior to executing the **ret**instruction. Note that using the **enter** instruction at the start of the target code will automatically save **%ebp** and set the stack pointer **%esp** to a fresh part of the stack. Then, prior to the **ret**, the **leave** instruction will restore the **%ebp**, assuming that any elements that were pushed onto the stack have been removed.

3.6 Loops

In addition to the `if-then-else` construct, *L3* provides `for` and `while` loops. They have the same behavior as in C. `continue` skips the rest of the statements in the loop body and `break` jumps to the first statement after the loop. `continue` and `break` are always bound to the innermost loop.

3.7 Boolean expressions

Some boolean expressions are "short circuited," as in C. The evaluation of the second operand for **&&** and || must not be performed if the first already determines the result.

There are no comparison operators for **bool** values. If needed, they can be formed in *L3* using the boolean expressions `!(a ^^ b)` for `(a == b)` and `(a ^^ b)` for `(a != b)`.

3.8 Safe arithmetic

In order to increase the safety of the language, the compiler will emit code that catches fatal run-time errors resulting from division and modulo by zero. If the program detects an attempt to divide by zero, `_l3_error()` will be called and the program will terminate. Note that it is not equivalent to simply failing with a default floating point exception—the compiler will emit a check for this before every division and modulo that can possibly fail.

The compiler does not produce a compile-time error even if these error conditions can be detected statically. A compile-time warning would be helpful to the programmer, but is optional.

In keeping with most other compilers, shifts by negative values or values greater than 31 have undefined behavior, but will still be accepted by the compiler. Note that the assembler we are using does not accept immediate shift values larger than 8 bits.

Overflow on integer arithmetic is silently permitted without error.

3.9 Run-time Environment

Your target code will be linked against a run-time library which we provide, using the command `gcc foo.s runtime/*.c`. The tests will be run in the standard Linux environment; the produced assembly code must conform to those standards. The run-time library includes I/O support, 2-D graphics output, and a variety of utility functions.

The runtime expects a procedure `main()` that takes no arguments. It is a linker error if `main` is not provided. That means that the programmer who uses your compiler should provide a `main` procedure in the source file. The return type of `main()` must be void. The return value in **%eax** will be interpreted by the operating system as the exit status of the program. This exit status should follow standard conventions; namely, zero exit status on successful completion, non-zero exit status on an error.

The *L3* runtime library provides many functions; here are the ones you will likely use the most:

`_alloc()`	Allocates memory and returns a pointer to it
`_l3_error()`	Prints an error and exits the program with a non-zero exit code
`print_int()`	Prints an integer value
`print_bool()`	Prints a boolean value ("true" or "false")
`print_char()`	Prints the ASCII character corresponding to the passed number
`print_newline()`	Goes to the next line
`print_tab()`	Prints a tab
`print_space()`	Prints the indicated number of spaces
`print_flush()`	Flushes stdout
`read_int()`	Reads an int from the terminal

The C-style prototypes are:

```
void* _alloc(int i)
void _l3_error(const char *srcfile, int line, const char *errmsg)
void print_int(int i)
void print_bool(int b)
void print_char(int c)
void print_newline()
void print_tab()
void print_space(int n)
void print_flush()
int read_int()
```

Please refer to the runtime source files for any other functions you may wish to use.

3.9.1 glut

The `l3graphics` library uses the glut environment. The necessary libraries and files are already installed on facilitized machines.

4 Implementation

This section covers two important internal compiler APIs: the code generator and the intermediate representation. The APIs and sample code are taken from the Java implementation, so names and particulars may vary in the SML/NJ and OCaml implementations.

4.1 Intermediate Representation

The compiler uses a tree-form intermediate representation. The root class for the IR is `tree.IRStatement`. As `IRStatement` implements the Visitable design pattern, it is easy to traverse the representation of an entire source file.

4.2 Code Generation

Code is generated using and Appel-style generator. A list of fragments is passed to the code generator, which uses the visitor pattern to traverse the graph structure of the IR and greedily generate `String` objects containing the assembly code for the IR.

4.2.1 Important Classes/Interfaces

- `edu.cmu.cs.l3.general.Visitor` defines `visitResponse()`, used by the Visitor pattern
- `edu.cmu.cs.l3.general.Visitable` defines `m_className`, used by the Visitor pattern
- `edu.cmu.cs.l3.translate.Fragment` a code fragment that can contain a frame and asm instructions
- `edu.cmu.cs.l3.x86.X86CodeGenerator` the main code generation class

5 *L3* Formalism

This section is designed to serve as the final word on *L3*. While it is as complete as possible, there are some aspects of the project not addressed by these specifications–for instance, the interaction with the runtime system.

We have made our best effort to keep the English and formal specifications in agreement where they overlap, but should you find any contradictions you should let us know ASAP so we can clarify intent.

5.1 Syntax

The syntax of *L3* is given in Figure 3. Note that $\star$ stands for the Kleene closure while $*$ stands for the asterisk terminal.

5.1.1 Precedence of operators

The precedence of unary and binary operators is given in Figure 4.

5.1.2 Derived forms

Some of the expression and assignment operators are derived forms. These are defined in following table. In this table, e_1 is semantically equivalent to e_2.

e_1	e_2
`x[y]`	`*(x *+ y)`
`x->y`	`(*x).y`
`x += y;`	`x = x + y;`
`x -= y;`	`x = x - y;`
`x *= y;`	`x = x * y;`
`x /= y;`	`x = x / y;`
`x %= y;`	`x = x % y;`

5.1.3 Else association

As in most languages, else statements are associated with the most recent if statement.

5.2 Formal Type System

This section formally describes the type system for *L3*.

5.2.1 Contexts for Type Checking

We will use the following context definitions: Σ contains function declarations, Δ contains the user-defined struct types and Γ contains the variables of the current scope. $\cdot$ is the empty context. We have:

⟨program⟩ ::= [⟨struct⟩ | ⟨function⟩]⋆

⟨stuct⟩ ::= **struct** ⟨ident⟩ **{** ⟨ident⟩ **:** ⟨type⟩ [**;** ⟨ident⟩ **:** ⟨type⟩]⋆ [**;**] **}** **;**

⟨function⟩ ::= ⟨type⟩ ⟨ident⟩ **(** ⟨paramlist⟩ **)** ⟨body⟩ | **void** ⟨ident⟩ **(** ⟨paramlist⟩ **)** ⟨body⟩

⟨paramlist⟩ ::= ε | ⟨ident⟩ **:** ⟨type⟩ [**,** ⟨ident⟩ **:** ⟨type⟩]⋆

⟨body⟩ ::= **{** ⟨decl⟩⋆ ⟨stmt⟩⋆ **}** | **foreign**

⟨decl⟩ ::= **var** ⟨ident⟩ [**,** ⟨ident⟩]⋆ **:** ⟨type⟩ **;**

⟨type⟩ ::= **bool** | **int** | ⟨ident⟩ | ⟨type⟩*****

⟨stmt⟩ ::= ⟨simp⟩ **;** | ⟨control⟩ | **;**

⟨simp⟩ ::= ⟨lval⟩ ⟨asop⟩ ⟨exp⟩ | ⟨exp⟩ | **return** ⟨exp⟩ | **return**

⟨control⟩ ::= **if** **(** ⟨exp⟩ **)** ⟨block⟩ [**else** ⟨block⟩] |
for **(** [⟨lval⟩ ⟨asop⟩ ⟨exp⟩] **;** ⟨exp⟩ **;** [⟨simp⟩] **)** ⟨block⟩ |
while **(** ⟨exp⟩ **)** ⟨block⟩ | **continue** | **break**

⟨block⟩ ::= ⟨stmt⟩ | **{** ⟨stmt⟩⋆ **}**

⟨exp⟩ ::= **(** ⟨exp⟩ **)** | ⟨const⟩ | ⟨lval⟩ | ⟨unop⟩ ⟨exp⟩ | ⟨exp⟩ ⟨binop⟩ ⟨exp⟩ |
`&` ⟨lval⟩ | **alloc** **(** ⟨exp⟩ **,** ⟨type⟩ **)** | **offset** **(** ⟨exp⟩ **)** | **size** **(** ⟨exp⟩ **)** |
⟨ident⟩ **(** [⟨exp⟩ [**,** ⟨exp⟩]⋆] **)**

⟨lval⟩ ::= **(** ⟨lval⟩ **)** | ⟨ident⟩ | `*` ⟨exp⟩ | ⟨exp⟩ `[` ⟨exp⟩ `]` |
⟨lval⟩ **.** ⟨ident⟩ | ⟨exp⟩ `->` ⟨ident⟩ | **(** ⟨lval⟩ **)**

⟨const⟩ ::= ⟨intconst⟩ | **true** | **false** | **NULL**

⟨ident⟩ ::= [**A-Z_a-z**][**0-9A-Z_a-z**]⋆

⟨intconst⟩ ::= [**0-9**][**0-9**]⋆

⟨asop⟩ ::= `=` | `+=` | `-=` | `*=` | `/=` | `%=`

⟨binop⟩ ::= `+` | `-` | `*` | `/` | `%` | `<` | `<=` | `==` | `!=` | `>` | `>=` |
`&&` | `||` | `^^` | `&` | `|` | `^` | `<<` | `>>` | `*+` | `*-` | `*==` | `*!=`

⟨unop⟩ ::= `!` | `~` | `-`

Non-terminals are in ⟨brackets⟩.
Terminals are in **bold**.

Figure 3: Grammar of *L3*

Operator	Associates	Class	Meaning
`()`	left	n/a	function call, explicit parentheses
`[] -> .`	left	postfix	subscripting, indirect selection, direct selection
`! ~ - & *`	right	unary	logical not, bitwise not, unary minus, reference, dereference
`* / %`	left	binary	integer times, divide, modulo
`+ -`	left	binary	integer plus, minus
`<< >>`	left	binary	(signed) shift left, shift right
`&`	left	binary	bitwise AND
`^`	left	binary	bitwise XOR
`|`	left	binary	bitwise OR
`*+ *-`	left	binary	pointer arithmetic (plus, minus)
`< <= > >=`	left	binary	integer comparison
`== != *== *!=`	left	binary	comparison (int and pointer)
`&&`	left	binary	logical and
`^^`	left	binary	logical xor
`||`	left	binary	logical or
`= += -= *= /= %=`	right	binary	assignment

Figure 4: Precedence of operators, from highest to lowest

$$
\begin{array}{lll}
\Sigma & ::= & \cdot \mid \Sigma, (\mathit{fn}; \tau_r;\ p_1{:}\tau_1,\ p_2{:}\tau_2,\ \ldots\ p_n{:}\tau_n) \\
\Delta & ::= & \cdot \mid \Delta, (s; f_1{:}\tau_1,\ f_2{:}\tau_2,\ \ldots\ f_n{:}\tau_n) \\
\Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \\
\tau & ::= & \texttt{int} \mid \texttt{bool} \mid \mathrm{s} \mid \texttt{NS} \mid \texttt{void} \mid \tau^*
\end{array}
$$

where *fn* is a metavariable standing for the name of a function, and s for the name of a struct type.
Functions are defined to have a return type τ_r and formal parameters $p_1{:}\tau_1,\ p_2{:}\tau_2,\ \ldots\ p_n{:}\tau_n$. The special type `void` is used for procedures which do not return a result. Structs are defined to have a list of fields in the format *field name* : *type*. `NS` is a special type, used for **NULL**, which is compatible with any pointer type.

We write $\Sigma(\mathit{fn})$ if a function with name *fn* is defined in Σ, $\Sigma(\mathit{fn}) = (\tau_r;\ p_1{:}\tau_1,\ p_2{:}\tau_2,\ \ldots\ p_n{:}\tau_n)$ if a function with name *fn*, return type τ_r and parameters p_i of type τ_i occurs in Σ, and $\overline{\Sigma}(\mathit{fn})$ if *no* function named *fn* occurs in Σ.

We write $\Delta(s)$ if a struct with name s occurs in Δ, $\Delta(s, f){:}\tau$ if a struct with name s and a field with name f of type τ occurs in Δ, and $\overline{\Delta}(s)$ if *no* struct named s occurs in Δ.

For the variable context, we write $\overline{\Gamma}(x)$ if no variable with name x occurs in Γ.

The judgments for contexts are:

$\vdash \Sigma\ \mathit{FCon}$	Σ is a valid (function declaration) context
$\vdash \Delta\ \mathit{SCon}$	Δ is a valid (struct type) context
$\vdash \Gamma\ \mathit{VCon}$	Γ is a valid (variable) context

If any context stands on the left-hand side of a judgment, it is implicitly assumed that it is a valid context:

$$\frac{}{\Sigma \vdash \Sigma\ \mathit{FCon}}\ \mathit{FCon} \qquad \frac{}{\Delta \vdash \Delta\ \mathit{SCon}}\ \mathit{SCon} \qquad \frac{}{\Gamma \vdash \Gamma\ \mathit{VCon}}\ \mathit{VCon}$$

5.2.2 Types

We write $\mathtt{T_{valid}}(\tau)$ if τ is a valid type. $\mathtt{T_{valid}}()$ is defined by the following inference rules. Note that `NS` and `void` are *not* valid types under this definition.

$$\frac{}{\Delta \vdash \mathtt{T_{valid}}(\mathtt{int})}\ \mathit{Tint} \qquad \frac{}{\Delta \vdash \mathtt{T_{valid}}(\mathtt{bool})}\ \mathit{Tbool}$$

$$\frac{\Delta \vdash \mathtt{T_{valid}}(\tau)}{\Delta \vdash \mathtt{T_{valid}}(\tau*)}\ \mathit{Tptr} \qquad \frac{\Delta \vdash \Delta(s)}{\Delta \vdash \mathtt{T_{valid}}(s)}\ \mathit{Tstr}$$

Type checking requires comparison of types. We will write $\mathtt{TypCMP}(\tau_1, \tau_2)$ if the two types τ_1 and τ_2 are compatible. We don't only test for type equality, because we have the special expression **NULL** of type $\mathtt{NS}^*$, which is compatible with but not equal to any pointer type. This can be seen as a special case of subtyping.

Type compatibility is defined by the following inference rules:

$$\frac{\Delta \vdash \mathtt{T_{valid}}(\tau)}{\Delta \vdash \mathtt{TypCMP}(\tau, \tau)}\ tcmp\_valid \qquad \frac{}{\Delta \vdash \mathtt{TypCMP}(\mathtt{NS}, \mathtt{NS})}\ tcmp\_NS$$

$$\frac{\Delta \vdash \mathtt{T_{valid}}(\tau)}{\Delta \vdash \mathtt{TypCMP}(\tau^*, \mathtt{NS}^*)}\ tcmp\_ptr\_NS \qquad \frac{\Delta \vdash \mathtt{T_{valid}}(\tau)}{\Delta \vdash \mathtt{TypCMP}(\mathtt{NS}^*, \tau^*)}\ tcmp\_NS\_ptr$$

$$\frac{\Delta \vdash \mathtt{TypCMP}(\tau_1, \tau_2)}{\Delta \vdash \mathtt{TypCMP}(\tau_1^*, \tau_2^*)}\ tcmp\_ptr$$

We define $\mathtt{Ptr_s}(\tau)$, which is true when τ is the type of pointers to struct type s, or a pointer to a pointer to s, and so on.

$$\frac{\Delta \vdash \mathtt{TypCMP}(\tau, s*)}{\Delta \vdash \mathtt{Ptr_s}(\tau)}\ ptr\_s \qquad \frac{\Delta \vdash \mathtt{TypCMP}(\tau, \tau'*) \quad \Delta \vdash \mathtt{Ptr_s}(\tau')}{\Delta \vdash \mathtt{Ptr_s}(\tau)}\ ptr\_ptr\_s$$

5.2.3 Extension of a context

We now define the valid extension of a context, starting with variable contexts:

$$\frac{}{\Gamma; \Delta \vdash \cdot\ VCon}\ emptyVCon \qquad \frac{\Gamma \vdash \overline{\Gamma}(x) \quad \Delta \vdash \mathtt{T_{valid}}(\tau)}{\Gamma; \Delta \vdash \Gamma, x : \tau\ VCon}\ addVCon$$

For struct contexts:

$$\frac{}{\Delta \vdash \cdot\ SCon}\ emptySCon$$

$$\frac{\Delta \vdash \overline{\Delta}(s) \quad \Delta \vdash \mathtt{T_{valid}}(\tau_1) \quad \cdots \quad \Delta \vdash \mathtt{T_{valid}}(\tau_n)}{\Delta \vdash \Delta, (s; f_1 \colon \tau_1,\ f_2 \colon \tau_2, \ldots\ f_n \colon \tau_n)\ SCon}\ addSCon$$

where no two field names $f_1, \ldots, f_n$ are equivalent, and if s is mentioned in any field type τ_i, then $\Delta \vdash \mathtt{Ptr_s}(\tau_i)$

Finally, for function contexts:

$$\frac{}{\Sigma; \Delta \vdash \cdot\ FCon}\ emptyFCon$$

$$\frac{\Sigma \vdash \overline{\Sigma}(f) \quad \Delta \vdash \mathtt{T_{valid}}(\tau_r) \quad \Delta \vdash \mathtt{T_{valid}}(\tau_1) \quad \cdots \quad \Delta \vdash \mathtt{T_{valid}}(\tau_n)}{\Sigma; \Delta \vdash \Sigma, (f; \tau_r;\ p_1 \colon \tau_1,\ p_2 \colon \tau_2, \ldots\ p_n \colon \tau_n)\ FCon}\ addFCon$$

where τ_r is not a struct type, f is not *main* and no two formal parameter names $p_1, \ldots, p_n$ are equivalent.

$$\frac{\Sigma \vdash \overline{\Sigma}(f) \quad \Delta \vdash \mathtt{T_{valid}}(\tau_1) \quad \cdots \quad \Delta \vdash \mathtt{T_{valid}}(\tau_n)}{\Sigma; \Delta \vdash \Sigma, (f; \mathtt{void};\ p_1 \colon \tau_1,\ p_2 \colon \tau_2, \ldots\ p_n \colon \tau_n)\ FCon}\ addFConVoid$$

where no two formal parameter names $p_1, \ldots, p_n$ are equivalent and f is not *main*.

$$\frac{\Sigma \vdash \overline{\Sigma}(main)}{\Sigma;\Delta \vdash \Sigma, (main; \texttt{void};)\ FCon)}\ addFConMain$$

A context can be assigned a certain content using the := symbol. For example, $\Gamma := \Gamma, x{:}\,\tau_x$ adds the variable x of type τ_x to Γ.

5.2.4 Expressions

In order to type-check an expression e, we must show that all its atomic parts are valid sub-expressions and that they form a valid expression. We will write Lval(l):τ_l if l is a valid L-value of type τ_l. Similar, we write Exp(e):τ_e if e is a valid expression of type τ_e.

Constants

$$\frac{}{\vdash \texttt{Exp}(\textbf{true}) : \texttt{bool}}\ exp\_true \qquad \frac{}{\vdash \texttt{Exp}(\textbf{false}) : \texttt{bool}}\ exp\_false$$

$$\frac{}{\vdash \texttt{Exp}(\langle \text{intconst} \rangle) : \texttt{int}}\ exp\_int$$

$$\frac{}{\vdash \texttt{Exp}(\textbf{NULL}) : \texttt{NS}^*}\ exp\_NS$$

L-values

$$\frac{}{\Sigma;\Delta;\Gamma, x{:}\,\tau \vdash \texttt{Lval}(x) : \tau}\ lval\_var$$

$$\frac{\Sigma;\Delta;\Gamma \vdash \texttt{Exp}(e) : \tau^*}{\Sigma;\Delta;\Gamma \vdash \texttt{Lval}(^*e) : \tau}\ lval\_deref$$

$$\frac{\Sigma;\Delta;\Gamma \vdash \texttt{Lval}(l) : \tau \quad \Delta \vdash \Delta(\tau, \langle \text{ident} \rangle) : \tau_i}{\Sigma;\Delta;\Gamma \vdash \texttt{Lval}(l.\langle \text{ident} \rangle) : \tau_i}\ lval\_struct$$

Binary Operators

$$\frac{\Sigma;\Delta;\Gamma \vdash \texttt{Exp}(x) : \texttt{int} \quad \Sigma;\Delta;\Gamma \vdash \texttt{Exp}(y) : \texttt{int}}{\Sigma;\Delta;\Gamma \vdash \texttt{Exp}(x \diamond y) : \texttt{int}}\ exp\_int\_arith$$

with $\diamond ::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{\&} \mid \texttt{|} \mid \texttt{\^{}} \mid \texttt{<<} \mid \texttt{>>}$

$$\frac{\Sigma;\Delta;\Gamma \vdash \texttt{Exp}(x) : \texttt{int} \quad \Sigma;\Delta;\Gamma \vdash \texttt{Exp}(y) : \texttt{int}}{\Sigma;\Delta;\Gamma \vdash \texttt{Exp}(x \diamond y) : \texttt{bool}}\ exp\_int\_comp$$

with $\diamond ::= \texttt{<} \mid \texttt{<=} \mid \texttt{==} \mid \texttt{!=} \mid \texttt{>=} \mid \texttt{>}$

$$\frac{\Sigma;\Delta;\Gamma \vdash \texttt{Exp}(x) : \texttt{bool} \quad \Sigma;\Delta;\Gamma \vdash \texttt{Exp}(y) : \texttt{bool}}{\Sigma;\Delta;\Gamma \vdash \texttt{Exp}(x \diamond y) : \texttt{bool}}\ exp\_bool\_arith$$

with $\diamond ::= \texttt{\&\&} \mid \texttt{||} \mid \texttt{\^{}\^{}}$

$$\frac{\Sigma;\Delta;\Gamma \vdash \texttt{Exp}(x) : \tau^* \quad \Sigma;\Delta;\Gamma \vdash \texttt{Exp}(y) : \texttt{int}}{\Sigma;\Delta;\Gamma \vdash \texttt{Exp}(x \diamond y) : \tau^*}\ exp\_ptr\_arith$$

with $\diamond ::= \texttt{*+} \mid \texttt{*-}$

$$\frac{\Sigma;\Delta;\Gamma \vdash \texttt{Exp}(x) : \tau_x^* \quad \Sigma;\Delta;\Gamma \vdash \texttt{Exp}(y) : \tau_y^*}{\Sigma;\Delta;\Gamma \vdash \texttt{Exp}(x \diamond y) : \texttt{bool}}\ exp\_ptr\_comp$$

with $\diamond ::= \texttt{*==} \mid \texttt{*!=}$

Unary Operators

$$\frac{\Sigma;\Delta;\Gamma \vdash \texttt{Exp}(x) : \texttt{int}}{\Sigma;\Delta;\Gamma \vdash \texttt{Exp}(\sim x) : \texttt{int}}\ exp\_int\_not \qquad \frac{\Sigma;\Delta;\Gamma \vdash \texttt{Exp}(x) : \texttt{int}}{\Sigma;\Delta;\Gamma \vdash \texttt{Exp}(-x) : \texttt{int}}\ exp\_int\_minus$$

$$\frac{\Sigma;\Delta;\Gamma \vdash \texttt{Exp}(x) : \texttt{bool}}{\Sigma;\Delta;\Gamma \vdash \texttt{Exp}(!x) : \texttt{bool}}\ exp\_bool\_not$$

Miscellaneous Expressions

$$\frac{\Gamma \vdash \texttt{Lval}(x) : \tau}{\Gamma \vdash \texttt{Exp}(x) : \tau}\ exp\_lval$$

$$\frac{\Sigma;\Delta;\Gamma \vdash \texttt{Lval}(l) : \tau}{\Sigma;\Delta;\Gamma \vdash \texttt{Exp}(\ \texttt{\&}l\ ) : \tau^*}\ exp\_ref$$

$$\frac{\Sigma;\Delta;\Gamma \vdash \texttt{Exp}(e) : \texttt{int} \quad \Delta \vdash \texttt{T}_{\texttt{valid}}(\tau)}{\Sigma;\Delta;\Gamma \vdash \texttt{Exp}(\ \textbf{alloc}(e, \tau)\ ) : \tau^*}\ exp\_alloc$$

$$\frac{\Sigma;\Delta;\Gamma \vdash \texttt{Exp}(e) : \tau^*}{\Sigma;\Delta;\Gamma \vdash \texttt{Exp}(\ \textbf{offset}(e)\ ) : \texttt{int}}\ exp\_offset \qquad \frac{\Sigma;\Delta;\Gamma \vdash \texttt{Exp}(e) : \tau^*}{\Sigma;\Delta;\Gamma \vdash \texttt{Exp}(\ \textbf{size}(e)\ ) : \texttt{int}}\ exp\_size$$

$$\frac{\Sigma \vdash \Sigma(fn) = (\tau_r;\ p_1{:}\tau_1,\ p_2{:}\tau_2,\ \ldots\ p_n{:}\tau_n) \quad \forall i.(\Sigma;\Delta;\Gamma \vdash \texttt{Exp}(a_i) : \tau\ \wedge\ \Delta \vdash \texttt{TypCMP}(\tau, \tau_i))}{\Sigma;\Delta;\Gamma \vdash \texttt{Exp}(\ \langle fn \rangle (a_1, a_2, ..., a_n)\ ) : \tau_r}\ exp\_fun$$

5.2.5 Statements

For type checking statements, we need yet another context, Ξ, for control information. For our purpose, it contains only the return type of the current function and an entry `loop`, which is set if the current statement is in a loop. We write $\Xi(\texttt{rettype}){:}\tau_r$ if the return type of the current function is τ_r and $\Xi(\texttt{rettype})$:`void` if the current function does not have a return value. $\Xi(\texttt{loop})$ is used to indicate that the current statement is executed within a loop. The judgment for a valid context Ξ is

$\vdash \Xi\ XCon$ Ξ is a valid context.

The following inference rules define how Ξ may be extended.

$$\frac{}{\Xi;\Delta \vdash \Xi\ XCon}\ XCon \qquad \frac{}{\Xi;\Delta \vdash \cdot\ XCon}\ emptyXCon$$

$$\frac{}{\Xi;\Delta \vdash \Xi, \texttt{loop}\ XCon}\ loopXCon$$

$$\frac{\Delta \vdash \texttt{T}_{\texttt{valid}}(\tau)}{\Xi;\Delta \vdash \Xi, (\texttt{rettype}, \tau)\ XCon}\ rettypeXCon$$

$$\frac{}{\Xi;\Delta \vdash \Xi, (\texttt{rettype}, \texttt{void})\ XCon}\ retvoidXCon$$

The judgment $\texttt{Stmt}(s)$ denotes that s is a valid statement. It is defined by the following inference rules. We write $\cdot$ for an empty statement, i.e. a statement that reduces to nothing.

$$\frac{\Sigma;\Delta;\Gamma \vdash \texttt{Lval}(l) : \tau_l \quad \Sigma;\Delta;\Gamma \vdash \texttt{Exp}(e) : \tau_e \quad \Delta \vdash \texttt{TypCMP}(\tau_l, \tau_e)}{\Sigma;\Xi;\Delta;\Gamma \vdash \texttt{Stmt}(\ l = e\ )}\ stmt\_assign$$

where τ_l is not a struct type.

$$\frac{\Sigma;\Xi;\Delta;\Gamma \vdash \texttt{Stmt}(s_1) \quad \Sigma;\Xi;\Delta;\Gamma \vdash \texttt{Stmt}(s_2)}{\Sigma;\Xi;\Delta;\Gamma \vdash \texttt{Stmt}(s_1; s_2)}\ stmt\_seq$$

$$\frac{\Sigma;\Delta;\Gamma \vdash \texttt{Exp}(e)}{\Sigma;\Xi;\Delta;\Gamma \vdash \texttt{Stmt}(e)}\ stmt\_exp \qquad \frac{}{\Sigma;\Xi;\Delta;\Gamma \vdash \texttt{Stmt}(\ \cdot\ )}\ stmt\_empty$$

$$\frac{\Sigma;\Delta;\Gamma \vdash \texttt{Exp}(e) : \tau \quad \Xi \vdash \Xi(\texttt{rettype}) : \tau_r \quad \Delta \vdash \texttt{TypCMP}(\tau_r, \tau)}{\Sigma;\Xi;\Delta;\Gamma \vdash \texttt{Stmt}(\ \textbf{return}\ e\ )}\ stmt\_retval$$

$$\frac{\Xi \vdash \Xi(\texttt{rettype}) : \texttt{void}}{\Sigma;\Xi;\Delta;\Gamma \vdash \texttt{Stmt}(\ \textbf{return}\ )}\ stmt\_ret$$

$$\frac{\Xi \vdash \Xi(\texttt{loop})}{\Sigma;\Xi;\Delta;\Gamma \vdash \texttt{Stmt}(\ \textbf{continue}\ )}\ stmt\_continue \qquad \frac{\Xi \vdash \Xi(\texttt{loop})}{\Sigma;\Xi;\Delta;\Gamma \vdash \texttt{Stmt}(\ \textbf{break}\ )}\ stmt\_continue$$

Control Statements

Control statements are used to execute several statements under a certain condition.

$$\frac{\Sigma;\Delta;\Gamma \vdash \texttt{Exp}(e) : \texttt{bool} \quad \Sigma;\Xi;\Delta;\Gamma \vdash \texttt{Stmt}(s)}{\Sigma;\Xi;\Delta;\Gamma \vdash \texttt{Stmt}(\ \textbf{if}(\ e\ )\ s\ )}\ stmt\_if$$

$$\frac{\Sigma;\Delta;\Gamma \vdash \texttt{Exp}(e) : \texttt{bool} \quad \Sigma;\Xi;\Delta;\Gamma \vdash \texttt{Stmt}(s_1) \quad \Sigma;\Xi;\Delta;\Gamma \vdash \texttt{Stmt}(s_2)}{\Sigma;\Xi;\Delta;\Gamma \vdash \texttt{Stmt}(\ \textbf{if}(\ e\ )\ s_1\ \textbf{else}\ s_2\ )}\ stmt\_if\_else$$

$$\frac{\begin{array}{l}\Sigma;\Xi;\Delta;\Gamma \vdash \texttt{Stmt}(s_1) \quad \Sigma;\Delta;\Gamma;\Xi, \texttt{loop} \vdash \texttt{Stmt}(s_2) \\ \Sigma;\Delta;\Gamma \vdash \texttt{Exp}(e) : \texttt{bool} \quad \Sigma;\Delta;\Gamma;\Xi, \texttt{loop} \vdash \texttt{Stmt}(s)\end{array}}{\Sigma;\Xi;\Delta;\Gamma \vdash \texttt{Stmt}(\ \textbf{for}(\ s_1;\ e;\ s_2)\ s\ )}\ stmt\_for$$

$$\frac{\Sigma;\Delta;\Gamma \vdash \texttt{Exp}(e) : \texttt{bool} \quad \Sigma;\Delta;\Gamma;\Xi, \texttt{loop} \vdash \texttt{Stmt}(s)}{\Sigma;\Xi;\Delta;\Gamma \vdash \texttt{Stmt}(\ \textbf{while}(\ e\ )\ s\ )}\ stmt\_while$$

5.2.6 Function Declarations

The body of a function consists of a list of variable declarations and a list of statements, where both can be empty. We will write $\langle decl :: stmts \rangle$ to describe those two lists. A declaration has the form, **var** $x : \tau$, and $d, decl$ is the list of declarations consisting of declaration d followed by the list $decl$.

To start, we define the validity of function declarations.

$$\frac{\begin{array}{c}\Sigma, (fn; \tau_r;\ p_1 : \tau_1,\ p_2 : \tau_2,\ \ldots\ p_n : \tau_n); \\ \Delta; \Gamma := \cdot, p_1 : \tau_1,\ p_2 : \tau_2,\ \ldots\ p_n : \tau_n; \Xi := \cdot, (rettype, \tau_r) \vdash \langle decl\ ::\ stmts \rangle\end{array}}{\Sigma; \Delta \vdash fn(\tau_r;\ p_1 : \tau_1,\ p_2 : \tau_2,\ \ldots\ p_n : \tau_n) \langle decl\ ::\ stmts \rangle}\ fun\_valid$$

Now we define the validity of function bodies:

$$\frac{\Delta; \Gamma \vdash \Gamma, x : \tau\ VCon \quad \Sigma; \Delta; \Gamma, x : \tau; \Xi \vdash \langle decl\ ::\ stmts \rangle}{\Sigma; \Xi; \Delta; \Gamma \vdash \langle \textbf{var}\ x : \tau, decl\ ::\ stmts \rangle}\ fun\_var\_decl$$

$$\frac{\Sigma; \Xi; \Delta; \Gamma \vdash \texttt{Stmt}(stmts)}{\Sigma; \Xi; \Delta; \Gamma \vdash \langle\ \cdot\ ::\ stmts \rangle}\ fun\_var\_stmts$$

5.2.7 Typechecking a Program

Type checking a program consists of the following steps:

1. $\Delta := \cdot$ and $\Sigma := \cdot$
2. Iterate through the whole program from top to bottom, adding struct definitions to Δ and function prototypes to Σ.
   - For each struct declaration **struct** s $\{ f_1 : \tau_1;\ f_2 : \tau_2; \ldots;\ f_n : \tau_n\ \}$ do
     $\Delta := \Delta, (s;\ f_1 : \tau_1;\ f_2 : \tau_2; \ldots;\ f_n : \tau_n)$ if
     $\Delta \vdash \Delta, (s;\ f_1 : \tau_1;\ f_2 : \tau_2; \ldots;\ f_n : \tau_n)\ SCon$ (see section 5.2.3)
   - For each function declaration $\tau_r\ fn(p_1 : \tau_1,\ p_2 : \tau_2,\ \ldots\ p_n : \tau_n)$ do
     $\Sigma := \Sigma, (fn; \tau_r;\ p_1 : \tau_1,\ p_2 : \tau_2, \ldots\ p_n : \tau_n)$ if
     $\Sigma; \Delta \vdash \Sigma, (fn; \tau_r;\ p_1 : \tau_1,\ p_2 : \tau_2, \ldots\ p_n : \tau_n)\ FCon$
3. Typecheck all functions following the rules described in section 5.2.6.

Note that functions are visible globally, struct declarations are only visible below their declaration for other struct declarations, but are also globally visible for functions. To achieve this, the program passed to the type checker is expected to have all struct declarations before the function definitions.

5.3 Formal Operational Semantics

This section describes formally how the compiled code must behave when executed. It might even give you some hints on how you can optimize your compiler, however, be warned that the

following description makes unrealistic memory assumptions and might also not be appropriate in other places. You should feel free to implement your compiler however you wish as long as it is behaviorally consistent with these semantics.

The operational semantics is designed to accept type-checked code as defined in Section 5.2, i.e. the program presented to the dynamic semantics must be valid.

We describe the behavior of the language with a set of transition rules for a finite state machine. Execution begins at a well defined entry point, and for successful execution, there should always be exactly one transition that can be taken. For an unambiguous language like *L3*, there always is at most one transition that applies, and if no transitions apply, it is a runtime error.

5.3.1 How to Read Dynamic Semantics

We use the following example to explain how this specification should be read. It consists of four rules that specify the transitions used for the addition of two variables.

$$S, M, R, B, K \vdash a + b \rightsquigarrow S, M, R, B, K \triangleright \square + b \vdash a \tag{1}$$

$$S, M, R, B, K \triangleright \square + b \vdash v_a \rightsquigarrow S, M, R, B, K \triangleright v_a + \square \vdash b \tag{2}$$

$$S, M, R, B, K \triangleright v_a + \square \vdash v_b \rightsquigarrow S, M, R, B, K \vdash eval((v_a + v_b)) \tag{3}$$

$$S : [x \to m], M : [m \to v_x], R, B, K \vdash x \rightsquigarrow S : [x \to m], M : [m \to v_x], R, B, K \vdash v_x \tag{4}$$

The most important symbol in the lines above is $\rightsquigarrow$. It indicates the **transition**. To the left is the machine state required to make the transition. To the right is the state that the machine will be in after the execution. At most one transition will apply, since the language is deterministic.

Next is the **variable context** S. It is essentially a list of variables that are defined at the point of execution. An empty context is represented with a single dot: $\cdot$. In rule 4 above, the context S is enriched by the definition of x. If this appears on the left of $\rightsquigarrow$, it is a requirement for the transition to be applicable. If an enrichment appears on the right of $\rightsquigarrow$, it means that this is the new state of the variable context. Note that the variable is not assigned directly the value, but a memory address m. The memory address is resolved in the **memory context** M, which contains the mappings from memory addresses to values. In rule 4 above, this means that variable x must be v_x, and it will continue to be present in the context with the value v_x.

K is the **current continuation stack**. It contains the expressions/statements that have begun computing, but have not finished. The triangle $\triangleright$ separates the top element (on the right of the triangle) from the rest of the stack - we always only look at the top element. The **box** $\square$, which must be present in each entry, shows where the value must be filled in in order for the computation to continue. Once the computation reaches some kind of a terminal (a value in the case of expressions or a void [written as "()"] in the case of a statement or declaration), there will exist another rule that will take the reached terminal and pop the continuation stack. Rule 2 is an example of that. Note that a non-value may not be plugged into the box.

R is the **return context** which stores the continuation stack K and the variable scope S of the caller of a function. The final context is called B and stores tuples (continuation stack, statements). We use B in loops to specify the behaviour of **break** and **continue**.

The only item that is left is the $\vdash$ sign. It separates the preconditions (the variable context and memory context) and the current continuation stack from either the piece of code that must be executed next or the result of the previous transition. In the case of a result from a previous transition, there must exist a transition that specifies how the value can be used. In the case of a computation,

there must exist a transition that specifies how the expression can be evaluated. In summary, in all cases there must always exist a transition rule that specifies what to do with the contents to the right of $\vdash$ (or if no rule applies, we have a runtime error or have reached the end of the program.)

Let's go through the example rules and examine how they indicate the expression $a + b$ must be executed:

1. In Rule 1, the LHS of $\rightsquigarrow$ is $S, M, K \vdash a + b$. This means that $a + b$ should be executed in the context given by S and M.
   The RHS of Rule 1 shows the first step of how this computation must be done: a is executed[2] first, while the rest of the computation is put on the continuation stack. Remember that the box $\square$ is the place where the result of executing a will be plugged in.

2. Rule 2 describes the transition that can be applied when $\square + b$ is on top of the continuation stack K and the RHS of $\vdash$ is a value v_a. It says that the value v_a is plugged into the box $\square$ and that b should be executed next.

3. Rule 3 shows the transition that can be applied when the top of the continuation stack is $v_a + \square$ and the RHS of $\vdash$ is a value v_b. It indicates that the sum of v_a and v_b can be evaluated and returned. The $eval((...))$ keyword means that the "real" result of the operation (as understood in this metalanguage) must be returned.

4. Rule 4 describes the transition used to look up the value associated with a variable. Its precondition to look up a variable named x is that x is bound in S to a memory address m and m must be bound in M to a value v_x. The postcondition of the transition is that the variable x is still bound to m in S and m is still bound in M. The rule tells us that we can return v_x as the result of executing the variable x, and is used to enable the LHS of rules 2 and 3 in the evaluation.

5.3.2 Values

A **small value** is a direct result of some expression. It is a single unit of data that contains an actual meaning. A small value can be one of the following:

- int
- bool
- memory address

Except for the **return** statement, all statements return "()" (pronounced "void" or "unit") upon successful execution. () is needed in order to step to the next statement. In the case of a **return**, the continuation stack K is emptied as there are no more statements to be executed and the program has successfully terminated.

[2]For this example, a is a variable, but could be an arbitrary (valid) expression. Execution of a variable means to return its current value.

The following 3 **heap values** are not the result of any expression. They are the data types that we use to represent safe pointers, structs, and arrays on the heap. An important note is that any scalar value may also be considered an array of size 1.

- safe pointer–a triple [memory address, int, int], where the first int is the offset and the second int the size
- struct–a list of memory locations where the elements reside
- array of values (of a specific size)

5.3.3 Rules

- Body
  $S, M, R, B, K \vdash \text{decls}; \text{stms} \rightsquigarrow S, M, R, B, K \triangleright \square; \text{stms} \vdash \text{decls}$
  $S, M, R, B, K \triangleright \square; \text{stms} \vdash () \rightsquigarrow S, M, R, B, K \vdash \text{stms}$

- Declarations
  $S, M, R, B, K \vdash id : \tau; \text{decls} \rightsquigarrow S, M, R, B, K \triangleright \square; \text{decls} \vdash id : \tau$
  $S, M, R, B, K \vdash id : \tau \rightsquigarrow S : [id \to m], M : [m \to \text{defaultValue}(\tau)], R, B, K \vdash ()$
  $S, M, R, B, K \triangleright \square; \text{decls} \vdash () \rightsquigarrow S, M, R, B, K \vdash \text{decls}$

- Statements
  $S, M, R, B, K \vdash \text{stm}; \text{stms} \rightsquigarrow S, M, R, B, K \triangleright \square; \text{stms} \vdash \text{stm}$
  $S, M, R, B, K \triangleright \square; \text{stms} \vdash () \rightsquigarrow S, M, R, B, K \vdash \text{stms}$

- Statement
  - Assignment
    $S, M, R, B, K \vdash \text{l} = \text{exp} \rightsquigarrow S, M, R, B, K \triangleright \text{l} = \square \vdash \text{exp}$
    $S, M, R, B, K \triangleright \text{l} = \square \vdash v \rightsquigarrow S, M, R, B, K \triangleright \square = v \vdash Lval(\text{l})$
    $S, M : [m \to Array(v_0, \ldots, v_o, \ldots, v_{s-1}), n \to \text{SafePtr}(m, o, s)],$
    $R, B, K \triangleright \square = v_{new} \vdash n \rightsquigarrow$
    $S, M : [m \to Array(v_0, \ldots, v_{new}, \ldots, v_{s-1}), n \to \text{SafePtr}(m, o, s)], R, B, K \vdash ()$
    if $0 \leq o < s$
    $S, M : [m \to w], R, B, K \triangleright \square = v \vdash m \rightsquigarrow S, M : [m \to v], R, B, K \vdash ()$
    if m does not point to a heap value of type $\text{SafePtr}(\_, \_, \_)$
  - If
    $S, M, R, B, K \vdash \textbf{if}(\text{exp})\{\text{stms}\} \rightsquigarrow S, M, R, B, K \triangleright \textbf{if}(\square)\{\text{stms}\} \vdash \text{exp}$
    $S, M, R, B, K \triangleright \textbf{if}(\square)\{\text{stms}\} \vdash \textbf{true} \rightsquigarrow S, M, R, B, K \vdash \text{stms}$
    $S, M, R, B, K \triangleright \textbf{if}(\square)\{\text{stms}\} \vdash \textbf{false} \rightsquigarrow S, M, R, B, K \vdash ()$

- IfElse
  $S, M, R, B, K \vdash \textbf{if}(\text{exp})\{\text{stms}_1\}\textbf{else}\{\text{stms}_2\} \rightsquigarrow$
  $S, M, R, B, K \triangleright \textbf{if}(\square)\{\text{stms}_1\}\textbf{else}\{\text{stms}_2\} \vdash \text{exp}$
  $S, M, R, B, K \triangleright \textbf{if}(\square)\{\text{stms}_1\}\textbf{else}\{\text{stms}_2\} \vdash \textbf{true} \rightsquigarrow S, M, R, B, K \vdash \text{stms}_1$
  $S, M, R, B, K \triangleright \textbf{if}(\square)\{\text{stms}_1\}\textbf{else}\{\text{stms}_2\} \vdash \textbf{false} \rightsquigarrow S, M, R, B, K \vdash \text{stms}_2$

- Return
  $S, M, R, B, K \vdash \textbf{return}\ \text{exp} \rightsquigarrow S, M, R, B, K \triangleright \textbf{return}\ \square \vdash \text{exp}$
  $S, M, R \triangleright (K', S'), B, K \triangleright \textbf{return}\ \square \vdash v \rightsquigarrow S', M, R, B, K' \vdash v$

- Return (void)
  $S, M, R \triangleright (K', S'), B, K \vdash \textbf{return}\ \rightsquigarrow S', M, R, B, K' \vdash ()$

- While
  $S, M, R, B, K \vdash \textbf{while}(\text{exp})\{\text{stms}\} \rightsquigarrow$
  $S, M, R, B \triangleright (K, \textbf{while}(\text{exp})\{\text{stms}\}), K \triangleright \textbf{while}(\square)\{\text{stms}\} \vdash \text{exp}$
  $S, M, R, B, K \triangleright \textbf{while}(\square)\{\text{stms}\} \vdash \textbf{false} \rightsquigarrow S, M, R, B, K \vdash ()$
  $S, M, R, B, K \triangleright \textbf{while}(\square)\{\text{stms}\} \vdash \textbf{true} \rightsquigarrow$
  $S, M, R, B, K \triangleright \textbf{while}(\textbf{true})\{\square\} \vdash \text{stms}; \textbf{continue}$

- For
  $S, M, R, B, K \vdash \textbf{for}(\text{stm}_1; \text{exp}; \text{stm}_2)\{\text{stms}\} \rightsquigarrow$
  $S, M, R, B \triangleright (K, \text{stm}_2; \textbf{for}((); \text{exp}; \text{stm}_2)\{\text{stms}\}), K \triangleright \textbf{for}(\square; \text{exp}; \text{stm}_2)\{\text{stms}\} \vdash \text{stm}_1$
  $S, M, R, B, K \triangleright \textbf{for}(\square; \text{exp}; \text{stm}_2)\{\text{stms}\} \vdash () \rightsquigarrow$
  $S, M, R, B, K \triangleright \textbf{for}((); \square; \text{stm}_2)\{\text{stms}\} \vdash \text{exp}$
  $S, M, R, B, K \triangleright \textbf{for}((); \square; \text{stm}_2)\{\text{stms}\} \vdash \textbf{false} \rightsquigarrow S, M, R, B, K \vdash ()$
  $S, M, R, B, K \triangleright \textbf{for}((); \square; \text{stm}_2)\{\text{stms}\} \vdash \textbf{true} \rightsquigarrow$
  $S, M, R, B, K \triangleright \textbf{for}((); \textbf{true}; \text{stm}_2)\{\square\} \vdash \text{stms}; \textbf{continue}$

- Continue
  $S, M, R, B \triangleright (K', \text{stms}), K \vdash \textbf{continue} \rightsquigarrow S, M, R, B, K' \vdash \text{stms}$

- Break
  $S, M, R, B \triangleright (K', \text{stms}), K \vdash \textbf{break} \rightsquigarrow S, M, R, B, K' \vdash ()$

- Exp
  $Exp(e)$ indicates that e is a an expression which is executed as a statement.
  $S, M, R, B, K \vdash Exp(\text{exp}) \rightsquigarrow S, M, R, B, K \triangleright Exp(\square) \vdash \text{exp}$
  $S, M, R, B, K \triangleright Exp(\square) \vdash v \rightsquigarrow S, M, R, B, K \vdash ()$

- Expression

  - Constants
    Constants are values, and thus no rules are needed.
    $S, M, R, B, K \vdash \textbf{NULL} \rightsquigarrow S, M, R, B, K \vdash \textbf{null}$

– Binary integer math, integer comparisons

$S, M, R, B, K \vdash \text{exp1}$ **op** $\text{exp2} \rightsquigarrow S, M, R, B, K \triangleright \square$ **op** $\text{exp2} \vdash \text{exp1}$
with **op** = `+` | `-` | `*` | `/` | `%` | `<` | `<=` | `==` | `!=` | `>` | `>=` | `<<` | `>>`

$S, M, R, B, K \triangleright \square$ **op** $\text{exp2} \vdash v1 \rightsquigarrow S, M, R, B, K \triangleright v1$ **op** $\square \vdash \text{exp2}$
with **op** = `+` | `-` | `*` | `/` | `%` | `<` | `<=` | `==` | `!=` | `>` | `>=` | `<<` | `>>`

$S, M, R, B, K \triangleright v1$ **op** $\square \vdash v2 \rightsquigarrow S, M, R, B, K \vdash eval((v1$ **op** $v2))$
with **op** = `+` | `-` | `*` | `<` | `<=` | `==` | `!=` | `>` | `>=` | `<<` | `>>`

$S, M, R, B, K \triangleright v1$ **op** $\square \vdash v2 \rightsquigarrow S, M, R, B, K \vdash eval((v1$ **op** $v2))$
with **op** = `/` | `%` and $v2 \neq 0$

– Boolean Operators

$S, M, R, B, K \vdash \text{exp1}$ **op** $\text{exp2} \rightsquigarrow S, M, R, B, K \triangleright \square$ **op** $\text{exp}\,2 \vdash \text{exp1}$
with **op** = `&&` | `||` | `^^`

∗ Short Circuit And

$S, M, R, B, K \triangleright \square$ `&&` $\text{exp2} \vdash$ **false** $\rightsquigarrow S, M, R, B, K \vdash$ **false**

$S, M, R, B, K \triangleright \square$ `&&` $\text{exp2} \vdash$ **true** $\rightsquigarrow S, M, R, B, K \vdash \text{exp2}$

∗ Short Circuit Or

$S, M, R, B, K \triangleright \square$ `||` $\text{exp2} \vdash$ **true** $\rightsquigarrow S, M, R, B, K \vdash$ **true**

$S, M, R, B, K \triangleright \square$ `||` $\text{exp2} \vdash$ **false** $\rightsquigarrow S, M, R, B, K \vdash \text{exp2}$

∗ No Short Circuit Xor

$S, M, R, B, K \triangleright \square$ `^^` $\text{exp2} \vdash v1 \rightsquigarrow S, M, R, B, K \triangleright v1$ `^^` $\square \vdash \text{exp2}$

$S, M, R, B, K \triangleright v1$ `^^` $\square \vdash v2 \rightsquigarrow S, M, R, B, K \vdash eval((v1$ `^^` $v2))$

– Unary Operators

$S, M, R, B, K \vdash$ **op** $\text{exp} \rightsquigarrow S, M, R, B, K \triangleright$ **op** $\square \vdash \text{exp}$
with **op** = `!` | `-` | `~`

$S, M, R, B, K \triangleright$ **op** $\square \vdash v \rightsquigarrow S, M, R, B, K \vdash eval((\,$ **op** $\,v))$
with **op** = `!` | `-` | `~`

– Pointer Arithmetic

$S, M, R, B, K \vdash \text{exp1}$ **op** $\text{exp2} \rightsquigarrow S, M, R, B, K \triangleright \square$ **op** $\text{exp2} \vdash \text{exp1}$
with **op** = `*+` | `*-`

$S, M : [n \rightarrow \text{SafePtr}(m, o, s)], R, B, K \triangleright \square$ **op** $\text{exp2} \vdash n \rightsquigarrow$
$S, M : [n \rightarrow \text{SafePtr}(m, o, s)], R, B, K \triangleright n$ **op** $\square \vdash \text{exp2}$
with **op** = `*+` | `*-`

$S, M : [n \rightarrow \text{SafePtr}(m, o, s)], R, B, K \triangleright n$ `*+` $\square \vdash v2 \rightsquigarrow$
$S, M : [n \rightarrow \text{SafePtr}(m, o, s), n' \rightarrow \text{SafePtr}(m, eval((o + v2)), s)], R, B, K \vdash n'$
if $m \neq 0$

$S, M : [n \rightarrow \text{SafePtr}(0, o, s)], R, B, K \triangleright n$ `*+` $\square \vdash v2 \rightsquigarrow$
$S, M : [n \rightarrow \text{SafePtr}(0, o, s)], R, B, K \vdash n$

$S, M : [n \rightarrow \text{SafePtr}(m, o, s)], R, B, K \triangleright n$ `*-` $\square \vdash v2 \rightsquigarrow$
$S, M : [n \rightarrow \text{SafePtr}(m, o, s), n' \rightarrow \text{SafePtr}(m, eval((o - v2)), s)], R, B, K \vdash n'$
if $m \neq 0$

$S, M : [n \to \text{SafePtr}(0, o, s)], R, B, K \triangleright n \texttt{ *- } \square \vdash v2 \rightsquigarrow$
$S, M : [n \to \text{SafePtr}(0, o, s)], R, B, K \vdash n$

– Pointer Comparison
$S, M, R, B, K \vdash \text{exp}_1 \textbf{ op } \text{exp}_2 \rightsquigarrow S, M, R, B, K \triangleright \square \textbf{ op } \text{exp}_2 \vdash \text{exp}_1$
with **op** = `*==` | `*!=`

$S, M, R, B, K \triangleright \square \textbf{ op } \text{exp}_2 \vdash p_1 \rightsquigarrow S, M, R, B, K \triangleright p_1 \textbf{ op } \square \vdash \text{exp}_2$
with **op** = `*==` | `*!=`

$S, M : [p_1 \to \text{SafePtr}(m_1, o_1, s_1), p_2 \to \text{SafePtr}(m_2, o_2, s_2)], R, B, K \triangleright p_1 \texttt{ *== } \square \vdash p_2 \rightsquigarrow$
$S, M : [p_1 \to \text{SafePtr}(m_1, o_1, s_1), p_2 \to \text{SafePtr}(m_2, o_2, s_2)], R, B, K \vdash$
$eval((m_1 \texttt{ == } m_2 \texttt{ \&\& } o_1 \texttt{ == } o_2))$

$S, M : [p_1 \to \text{SafePtr}(m_1, o_1, s_1), p_2 \to \text{SafePtr}(m_2, o_2, s_2)], R, B, K \triangleright p_1 \texttt{ *!= } \square \vdash p_2 \rightsquigarrow$
$S, M : [p_1 \to \text{SafePtr}(m_1, o_1, s_1), p_2 \to \text{SafePtr}(m_2, o_2, s_2)], R, B, K \vdash$
$eval((m_1 \texttt{ != } m_2 \texttt{ || } o_1 \texttt{ != } o_2))$

– Alloc
$S, M, R, B, K \vdash \mathit{Alloc}(\text{exp}, \tau) \rightsquigarrow S, M, R, B, K \triangleright \mathit{Alloc}(\square, \tau) \vdash \text{exp}$

$S, M, R, B, K \triangleright \mathit{Alloc}(\square, \tau) \vdash s \rightsquigarrow$
$S, M, R, B, K \triangleright \mathit{Alloc}'(\tau, \mathit{Array}(\square, d_1, \ldots, d_{s-1})) \vdash \text{defaultValue}(\tau)$

$S, M, R, B, K \triangleright \mathit{Alloc}'(\tau, \mathit{Array}(v_0, \ldots, v_{i-1}, \square, d_{i+1}, \ldots, d_{s-1})) \vdash v \rightsquigarrow$
$S, M, R, B, K \triangleright \mathit{Alloc}'(\tau, \mathit{Array}(v_0, \ldots, v_{i-1}, v, \square, \ldots, d_{s-1})) \vdash \text{defaultValue}(\tau)$

$S, M, R, B, K \triangleright \mathit{Alloc}'(\tau, \mathit{Array}(v_0, \ldots, v_{s-2}, \square)) \vdash v \rightsquigarrow$
$S, M : [m \to \mathit{Array}(v_0, \ldots, v_{s-2}, \square), n \to \text{SafePtr}(m, 0, s)], R, B, K \vdash n$

– Address Of
$S, M, R, B, K \vdash \texttt{\&}l \rightsquigarrow S, M, R, B, K \triangleright \texttt{\&}\square \vdash \mathit{Lval}(l)$

$S, M : [n \to \text{SafePtr}(m, o, s)], R, B, K \triangleright \texttt{\&}\square \vdash n \rightsquigarrow$
$S, M : [n \to \text{SafePtr}(m, o, s)], R, B, K \vdash n$

$S, M : [m \to \alpha], R, B, K \triangleright \texttt{\&}\square \vdash m \rightsquigarrow S, M : [m \to \alpha;\ n \to \text{SafePtr}(m, 0, 1)], R, B, K \vdash n$
if m does not point to a heap value of type $\text{SafePtr}(\_, \_, \_)$

– Size
$S, M, R, B, K \vdash \textbf{size}(\text{exp}) \rightsquigarrow S, M, R, B, K \triangleright \textbf{size}(\square) \vdash \text{exp}$

$S, M : [n \to \text{SafePtr}(m, o, s)], R, B, K \triangleright \textbf{size}(\square) \vdash n \rightsquigarrow$
$S, M : [n \to \text{SafePtr}(m, o, s)], R, B, K \vdash s$

– Offset
$S, M, R, B, K \vdash \textbf{offset}(\text{exp}) \rightsquigarrow S, M, R, B, K \triangleright \textbf{offset}(\square) \vdash \text{exp}$

$S, M : [n \to \text{SafePtr}(m, o, s)], R, B, K \triangleright \textbf{offset}(\square) \vdash n \rightsquigarrow$
$S, M : [n \to \text{SafePtr}(m, o, s)], R, B, K \vdash o$

– Function Call
$S, M, R, B, K \vdash \mathit{funname}(n_1 = \text{exp}_1, \ldots, n_n = \text{exp}_n, (\mathit{body})) \rightsquigarrow$
$S, M, R, B, K \triangleright \mathit{funname}(n_1 = \square, \ldots, n_n = \text{exp}_n, (\mathit{body})) \vdash \text{exp}_1$

$S, M, R, B, K \triangleright funname(n_1 = v_1, \ldots, n_i = \square, n_i = \exp_i \ldots, n_n = \exp_n, (body)) \vdash v_i \rightsquigarrow$
$S, M, R, B, K \triangleright funname(n_1 = v_1, \ldots, n_i = v_i, n_i = \square, \ldots, n_n = \exp_n, (body)) \vdash \exp_i$

$S, M, R, B, K \triangleright funname(n_1 = v_1, \ldots, n_n = \square, (body)) \vdash v_n \rightsquigarrow$
$S' = \cdot : [n_1 \to m_1] : \ldots : [n_n \to m_n], M : [m_1 \to v_1] : \ldots : [m_n \to v_n],$
$R \triangleright (K, S), B, K \vdash (body)$

- Lval
  The keyword *Lval* is needed to specify that not the value, but the memory address where the information is stored must be returned.

  – Variable
  $S : [\text{var} \to m], M : [m \to v], R, B, K \vdash \text{var} \rightsquigarrow S : [\text{var} \to m], M : [m \to v], R, B, K \vdash v$
  $S : [\text{var} \to m], M, R, B, K \vdash Lval(\text{var}) \rightsquigarrow S : [\text{var} \to m], M, R, B, K \vdash m$

  – Deref
  $S, M, R, B, K \vdash *\text{exp} \rightsquigarrow S, M, R, B, K \triangleright *\square \vdash \text{exp}$
  $S, M : [n \to \text{SafePtr}(m, o, s), m \to Array(v_0, \ldots, v_o, \ldots, v_{s-1})], R, B, K \triangleright *\square \vdash n \rightsquigarrow$
  $S, M : [n \to \text{SafePtr}(m, o, s), m \to Array(v_0, \ldots, v_o, \ldots, v_{s-1})], R, B, K \vdash v_o$
  if $0 \leq o < s$
  $S, M, R, B, K \vdash Lval(*\text{exp}) \rightsquigarrow S, M, R, B, K \triangleright Lval(*\square) \vdash \text{exp}$
  $S, M : [n \to \text{SafePtr}(m, o, s)], R, B, K \triangleright Lval(*\square) \vdash n \rightsquigarrow$
  $S, M : [n \to \text{SafePtr}(m, o, s)], R, B, K \vdash n$

  – Structure Element
  $S, M, R, B, K \vdash \text{s.elem} \rightsquigarrow S, M, R, B, K \triangleright \square.\text{elem} \vdash \text{s}$
  $S, M : [n \to Struct(\ldots, \text{elem} \to m, \ldots), m \to v], R, B, K \triangleright \square.\text{elem} \vdash n \rightsquigarrow$
  $S, M : [n \to Struct(\ldots, \text{elem} \to m, \ldots), m \to v], R, B, K \vdash v$
  $S, M, R, B, K \vdash Lval(\text{s.elem}) \rightsquigarrow S, M, R, B, K \triangleright Lval(\square.\text{elem}) \vdash \text{s}$
  $S, M : [n \to Struct(\ldots, \text{elem} \to m, \ldots)], R, B, K \triangleright Lval(\square.\text{elem}) \vdash n \rightsquigarrow$
  $S, M : [n \to Struct(\ldots, \text{elem} \to m, \ldots)], R, B, K \vdash m$

- Default Values
  $S, M, R, B, K \vdash \text{defaultValue}(\textbf{int}) \rightsquigarrow S, M, R, B, K \vdash \textbf{undef}_{\text{int}}$
  $S, M, R, B, K \vdash \text{defaultValue}(\textbf{bool}) \rightsquigarrow S, M, R, B, K \vdash \textbf{undef}_{\text{bool}}$
  $S, M, R, B, K \vdash \text{defaultValue}(\tau*) \rightsquigarrow S, M, R, B, K \vdash \textbf{null}$
  null is a memory-address bound in M
  $S, M, R, B, K \vdash \text{defaultValue}(Struct(n_1 : \tau_1, \ldots, n_n : \tau_n)) \rightsquigarrow$
  $S, M, R, B, K \triangleright \text{defaultValue}(Struct(n_1 \to \square, \ldots, n_n : \tau_n)) \vdash \text{defaultValue}(\tau_1)$
  $S, M, R, B, K \triangleright \text{defaultValue}(Struct(n_1 \to v_1, \ldots, n_i \to \square, n_{i+1} : \tau_{i+1}, \ldots, n_n : \tau_n)) \vdash v \rightsquigarrow$
  $S, M, R, B, K \triangleright \text{defaultValue}(Struct(n_1 \to v_1, \ldots, n_i \to v, n_{i+1} \to \square, \ldots, n_n : \tau_n)) \vdash$
  $\text{defaultValue}(\tau_{i+1})$
  $S, M, R, B, K \triangleright \text{defaultValue}(Struct(n_1 \to v_1, \ldots, n_n \to \square)) \vdash v \rightsquigarrow$
  $S, M : [n \to Struct(n_1 \to v_1, \ldots, n_n \to v)], R, B, K \vdash n$

5.3.4 Execution

To begin executing the program, we set the current state of the machine to be

$S = \cdot, M = \cdot : [\mathbf{null} \to \text{SafePtr}(0, 0, 0)], R = \cdot : (\cdot, \cdot), B = \cdot, K = \cdot \vdash (\mathit{main\_body})$.

Upon successful completion, the machine will end in the state $S = \cdot, M, R = \cdot, B, K = \cdot \vdash ()$ for arbitrary B and M.