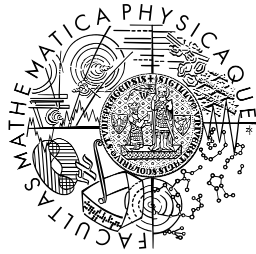


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Lukáš Rozsypal

Kampa: an experimental programming language

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: doc. Ing. Lubomír Bulej, Ph.D.
Study programme: Informatics
Study branch: Programming and Software Systems

Prague 2020

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

I would like to thank my supervisor Lubomír Bulej for his valuable advice and guidance, and Jaroslav Tulach for answering all possible questions about Truffle.

Title: Kampa: an experimental programming language

Author: Lukáš Rozsypal

Department: Department of Distributed and Dependable Systems

Supervisor: doc. Ing. Lubomír Bulej, Ph.D., Department of Distributed and Dependable Systems

Abstract: Kampa is a general-purpose programming language. It is imperative, but influenced by functional programming. Its distinguishing features include value types with a concise tuple syntax, immutability that applies recursively, and custom named operators. Closures and first-class functions are a matter of course. Thanks to dependent types, the size of an array may be bound to any immutable variable or field. Arrays can be embedded in other data structures. This, in conjunction with dependent types, allows almost arbitrary memory layouts. In addition to the specification, this thesis also provides a proof-of-concept implementation built on top of Truffle.

Keywords: Kampa programming language Truffle JIT compiler

Contents

Introduction **3**
Goals . . . 3
Structure of the thesis . . . 3

1 Analysis, design of the language **4**
1.1 Static typing . . . 4
1.2 Memory safety . . . 5
1.3 Syntax basics . . . 5
1.3.1 Initializers . . . 6
1.3.2 Semicolons and code blocks . . . 6
1.3.3 Files . . . 7
1.3.4 Types . . . 7
1.3.5 Parser directives and macros . . . 7
1.4 Value and reference types . . . 8
1.4.1 Value types . . . 9
1.4.2 Reference types . . . 10
1.4.3 Arrays . . . 11
1.5 Functions . . . 12
1.5.1 Requirements . . . 13
1.5.2 Specification . . . 14
1.6 Immutable types . . . 17
1.6.1 Specification . . . 18
1.7 Arrays and dependent types . . . 20
1.7.1 Generic functions . . . 21
1.7.2 Allowed dependencies . . . 21

2 Implementation **22**
2.1 Structure of the compiler . . . 22
2.1.1 Lexical analysis . . . 23
2.1.2 Syntax analysis . . . 23
2.1.3 Semantic analysis . . . 26
2.2 Memory representation . . . 29
2.2.1 Numbers . . . 30
2.2.2 Tuples . . . 30
2.2.3 Names and qualifiers . . . 30
2.2.4 Boxes . . . 30
2.2.5 Functions . . . 30
2.2.6 Arrays . . . 31
2.2.7 Types . . . 31
2.2.8 Instances of type parameters . . . 31
2.3 Temporary representation . . . 31
2.4 Truffle nodes . . . 32
2.4.1 Root nodes . . . 32
2.4.2 Block nodes . . . 32
2.4.3 Statement nodes . . . 32

2.4.4 Value nodes . . . 33
2.4.5 Write nodes . . . 33
2.4.6 Size nodes . . . 34
2.5 Library interface . . . 35
2.6 Functions . . . 36
2.7 Dependent types . . . 37
2.7.1 External dependencies . . . 37
2.7.2 Internal dependencies . . . 38
2.7.3 Dependencies in function types . . . 38
2.7.4 Consuming dependencies . . . 38
2.7.5 Other types . . . 38
2.7.6 Additional rules . . . 39
2.7.7 Operations . . . 39
2.8 Limitations . . . 40

Conclusion **41**
Future work . . . 41

Bibliography **42**

List of Figures **43**

List of Abbreviations **44**

A Attachments **45**
A.1 Kampa compiler and library . . . 45
A.2 Examples . . . 46

Introduction

People are used to think differently than how processors in computers work and programming languages were invented to serve as the point that is still understandable to the human, but can already be converted to the instructions for the computer. In this sense, the programming language is a common language for the human and the computer.

The language should not be yet another obstacle. If the programmers often find themselves evading the restrictions of the language, using patterns to satisfy artificial requirements, or inventing abstractions to get something more primitive than the language offers, then the language does not serve its purpose well, as it is conceptually not anywhere between the programmer and the computer, but rather alien to both.

As an extreme example, multiplying two numbers in Brainfuck is relatively difficult and inefficient, although it is innate to both humans and most processors. Brainfuck is not a good programming language. More practical examples include using objects to get functions, enforcing primitive values to be reference types, and in a way even using dynamic types where the programmer means to use one concrete type.

Goals

In this thesis, we plan to:

1. Define a set of requirements that will make the language convenient for the developer, while still mapping to machine instructions reasonably well.
2. Design a relatively ordinary general-purpose programming language that meets these requirements. In addition, the language should have simple and orthogonal semantics, while being at least as expressive as current programming languages.
3. Provide a proof-of-concept implementation in order to verify the feasibility and test the properties of the language. Since this is just a prototype, performance and a comprehensive library are not the main objectives.

Structure of the thesis

In the first chapter, we will present the analysis, define the requirements, and specify the solution in our language, topic by topic.

The implementation is described in the second chapter. The text complements the Javadoc comments. It documents the overall architecture, while the Javadocs concern each class mostly in isolation.

The attachments contain the implementation and a few example programs.

1. Analysis, design of the language

In this chapter, we will analyze the requirements the language should satisfy. We will present examples of common programming languages that do not. This does not mean the languages are bad or unusable, but that there is room for improvement. Kampa will definitely have its own problems.

We have chosen languages that are popular and the author has at least some experience with them. They are: Java, C, C++, C#, Python and JavaScript. All of them have been in the top 8 positions in the TIOBE index for several years [1] and are among the 8 most popular general-purpose programming languages according to the Stack Overflow Developer Survey 2019.[2] The other positions are indeterminate.

In each section, we will focus on one topic, define the requirements and the rationale, and specify the solution in our language. The ordering by topic (instead of separating analysis and specification) is used in order to minimize the need for cross-references.

1.1 Static typing

The language should be statically typed, i.e. the type safety should be verified at compile-time. Static typing has many advantages over dynamic typing, including:

Reliability. Type errors in dynamically typed programs are not discovered until the faulting statement is executed.

Documentation. In statically typed languages — unless using a Hindley-Milner type system — the type of a variable or parameter is explicitly stated in the program and it does not change. When reading the program, it is not necessary to look up all assignments to a variable and all call sites of a function to understand what type of data it will contain and what interface it will expose.

And vice versa: the user of a (library) function can see in the function signature, what should be passed to it. Passing an incorrect type results in an error at the function call, instead of an internal error many levels deeper.

Performance. Dynamically-typed programs must perform type checks at runtime. Additionally, without having the type information, many expressions are ambiguous, such as method calls or the "+" operator in many languages. In addition to often unnecessary branching, it means less optimizations can be carried out, or they must be speculative.

IDE support. If the compiler knows the types of variables at compile time, so can the IDE or language server. It can offer much more accurate completions during typing, and highlight more errors than just invalid syntax.

Many dynamically-typed languages later introduced some form of type annotations or type hints, such as Python[3] or PHP[4].

1.2 Memory safety

Bugs that cause memory corruptions are hard to trace, as they are unpredictable, often manifesting later and in another (even unrelated) component of the program. This is even more problematic in large projects with multiple developers, working on different components, where such bugs may get reported to the wrong person or team. Memory safety violations are common source of bugs and security vulnerabilities in languages like C and C++.[5] Some languages, such as Java and most scripting languages are memory-safe, meaning they detect errors and report them immediately. There are also languages that only allow unsafe operations like pointer arithmetic in so-called "unsafe blocks", such as C# and Rust. Unsafe blocks do not help at all, because any function can contain an unsafe block.

Error detection often causes overhead, and unsafe operations allow some programs to be written more efficiently. Whether it is worth the risk depends on the developers and the users. Kampa is designed with the premise that it almost never is. Therefore:

1. Unsafe operations like conversions between integer and reference types, or between incompatible reference types should not be allowed.
2. Array accesses have the potential of corrupting memory, so they should be either proven correct or checked.

Given that the proof-of-concept implementation runs in a managed environment, it is easier to implement a safe language than an unsafe one, but we would like to emphasize that this is by design, not by force of circumstances.

Also note that both requirements still make difference in our compiler, since many types use the same representation, and some writes that make sense for one type could be illegal for the other. This means that with unsafe conversions and unchecked array access, we could still manage to get a kind of memory corruption, although with limited range.

1.3 Syntax basics

This section will be a bit descriptive, but it is essential, as we will need to use this syntax in the examples. There is no other document that could serve as the specification this thesis could refer to. As a usual imperative language, Kampa has expressions and statements. Expressions have a value, which can be used by embedding them in other expressions. Statements generally do not have a value, they are put to the program to perform an action. Kampa has five kinds:

1. An **expression statement** consists of a single expression, whose value (result) is not used. These are used mostly for calls and assignments.
2. A **declaration** also consists of a single expression, but this one is parsed as a type, which is then used for a new local variable.
3. **Macro invocations** are replaced with their definition by the parser.
4. Restricted **goto** statements are intended for use in macro definitions.
5. Statement **labels** serve as targets for goto statements.

Whether a statement is an **expression statement** or a **declaration** depends on the expression. If it is a named expression (`identifier: another expr`) it is considered a declaration. Likewise with tuples/boxes thereof (tuples and boxes will be defined in 1.4, not important now). Other expressions, like the mentioned calls and assignments, are plain expression statements.

Most statements are terminated with a semicolon. The semicolon is *not* optional as in many hot new languages.

1.3.1 Initializers

An expression can be embedded in a statement trivially, using an expression statement. Initializer-expressions (initializers for short) provide the other direction: embedding statements in expressions. Because statements do not have values, the value must be provided by the initializer itself. The syntax of an initializer is `TYPE { STATEMENTS }` . When evaluated, it creates a new value of `TYPE` and then exectues all `STATEMENTS` . Besides other actions, they can initialize the value. The value is then "returned" from the initializer.

Notice the difference from initializers in other programming languages: that we can use arbitrary *statements* in an initializer, while other languages usually only allow a constant number of *expressions*. The code in the initializer may call any function, for example to add elements, set fields and properties, and so on. It may also declare local variables that are not seen outside the block.

Last, but not least, it can define new members, using **declaration** statements, but prefixing them with a dot (`.`). These are called OUTPUT declarations. They update the type of the value to be returned. It is useful for defining OOP objects, which are not otherwise supported in Kampa. That and more can be seen in the examples in A.2.

1.3.2 Semicolons and code blocks

We would like to avoid a *perceived* inconsistency present in many languages, but especially noticeable in JavaScript. Some braces should be followed by semicolon, while others should not (e.g. `function e() { }` vs. `e = function() { };`). While perfectly logical, it does not look good and is often omitted on purpose or by mistake. In JavaScript, the problem is that the function gets called if the next line starts with a parenthesis. In other languages, it at least does not compile.

In Kampa, it is not hard to determine when to add a semicolon: semicolon immediately following a brace is simply always superfluous. Note that no token- or whitespace-sensitive heuristic is used; a top level initializer expression *always* closes the expression and the statement. To suppres this behavior, it can be parenthesized.

On that note, we do not need code blocks as another type of statements. When a code block is used as a statement, it is in fact parsed as an expression statement containing an initializer with type void. Although semantically different from C code blocks, it has precisely the same effects, so it can still be used for grouping of statements.

1.3.3 Files

The file itself is an initializer, just without the braces. It starts with type void, but can extend it using OUTPUT declarations. The resulting value can be imported by other files, by importing this file. A file is imported using the expression `\"path/to/file.kampa"`, where the path is relative to the importing file.

As with usual blocks, the file may also use non-OUTPUT declarations to define "private" static variables. And as with usual blocks, the file may perform arbitrary calculations before returning, providing a form of static initialization.

The main file — the one that is given to the compiler on the command-line — should return a single function. This is the main function, used by the compiler as the entry point of the application.

Theoretically, it would be possible to return an arbitrary data structure to the compiler, like metadata, a set of functions for linking, etc. Currently, the primitive launcher in A.1 would consider this an error.

1.3.4 Types

Most type expressions will be introduced later, but these two are used in almost all examples, so it is appropriate to define them in advance.

A type is a value, and as such, it has a type ("metaclass" in OOP). The type of type `T` is called `? T`. The only current use is a type alias definition, such as `MyInt: ? Int`.

In some cases, the type expression may contain a default value. This is written as `TYPE = DEFAULT`. For example, to define a local variable `i` of type `Int` defaulting to 42, one would write `i: Int = 42`. The type may be omitted, in which case it is inferred, e.g. `i: = 42`.

1.3.5 Parser directives and macros

Parser directives are used to alter the way the parser processes the program. They can be used in the same places as statements, being distinguished from statements by starting with a question mark (`?`). There are three types of directives:

- Macro definitions.
- Precedence diretives.
- Directive imports.

Precedence directives will be defined later, in 1.5.2. Directive imports are written as `? \"path/to/file.kampa";` and they import all directives the file exports. A directive is exported by adding a dot (`.`) immediately after the `?`, which is intended to be similar to OUTPUT declarations.

Macro definitions

Macros are a way to extend the compiler. Unlike the textual C preprocessor, macros in Kampa operate on the syntax level. This means that they avoid many of their problems, such as those in figure 1.1, but not all of them, such as name collisions and accidental repeated evaluation of arguments. However, without

Figure 1.1: Problems with C macros

```
#define SUM(a,b) a+b
int forty_two = SUM(3, 3)*7; // evaluates to 3+3*7 = 24

#define FAIL(msg) puts(msg); exit(1);
if(1 < 0) FAIL("Bad math"); // exits always
```

being able to extend the compiler, people often resort to generating source code, which is a much worse idea.

Macros allow libraries to define structures that behave in the same way as standard control structures. In fact, all the standard control structures such as `if` and `while` are implemented using macros. The parser directive for defining a macro has the form `? keywords and arguments = statement template`.

Keywords are written in double quotes (`"`). The definition must start with a keyword, after that they can be interleaved arbitrarily. An argument can be one of following:

- `{IDENTIFIER}` — a statement
- `IDENTIFIER` — an expression
- `(` or `)` — a literal parenthesis
- `';'` (including the single-quotes) — a literal semicolon

The identifiers used to define statement and expression arguments are then substituted in the statement template. Macro definitions have the goto statement and the label statement at their disposal. There are two types of goto: one jumps at the start of a labeled statement, the other jumps at its end. Note that both are restricted to the statements inside of which they are. This is enough to implement loops and multi-level continue and break, yet it cannot skip declarations or make the control flow graph irreducible. Their syntax is:

- `-> label: statement` — a labeled statement
- `-> label;` — jump to the beginning of a statement
- `-> label !;` — jump to the end of a statement

1.4 Value and reference types

In this section, we will discuss the advantages and disadvantages of value and reference types. The difference between value and reference types is that a variable or field of a value type contains directly the value. A reference type instead stores the addres of the memory where the value is located.

The term composite type describes a type that is composed from other types. In this section, it will always be qualified whether we mean a value or a reference composite type.

1.4.1 Value types

Many languages omit the support for composite value types, often for simplicity. These include Java, JavaScript, and Python (which does not have even primitive types).

Composite value types are not strictly necessary, but there are many concepts that are more naturally expressed as value types than reference types, such as pairs, coordinates and dimensions, colors, date and time, and many more. One property they have in common is that they do not have identity; they are determined exclusively by their contents, exactly like primitive value types.

Additionally, if the language does not support composite value types, then any composite value must be stored in a separately allocated piece of memory. These allocations are done on the heap, which is more expensive. The only allocation that is needed for a local variable or parameter of value type is on stack, which usually does not have any overhead. For a field of value type, only the object that contains the field has to be allocated (and it would have to be anyway). This means that we always spare one allocation per instance when using a value type instead of a reference type.

It cannot be said that a language *needs* value types, but they have advantages that we do not want to ignore. Kampa supports composite value types in the form of tuples. It is using a lightweight syntax with commas (fig. 1.2), usually wrapped in parentheses, because commas have the lowest priority after semicolons. Tuple items can be accessed using names, if they are defined, destructuring is needed otherwise (fig. 1.3). They cannot be accessed by index, because the items can have different types — the result of an item access would have ambiguous type.

Figure 1.2: Tuple syntax basics

```
// tuple type definition
Color: ? (r: Byte, g: Byte, b: Byte);

// tuple value
cyan: Color = (r: 0, g: 255, b: 255);

// reading tuple item
zero: Byte = cyan.r;

// reading tuple as whole, copying it
blue: Color = cyan;

// updating tuple item (does not affect "cyan")
blue.g = 0;
```

Figure 1.3: Destructuring of tuples without names

```
readFD: Int, writeFD: Int;
// UNIX syscall pipe returns 2 file descriptors.
// not actual function in Kampa, just an example
(readFD, writeFD) = pipe();
```

It has not yet been established, when two tuple types are considered compatible (for assignment or passing to functions). Currently, it depends on the (only) implementation. Obviously they must be compatible item-wise, but it is not clear how names are treated. Insisting on exact match requires specifying names of function arguments. Ignoring names completely would cause unrelated types with the same layout to match. What our proof-of-concept implementation does is described in 2.1.3.

1.4.2 Reference types

All or almost all general-purpose languages provide some form of references.

C and C++ have pointers. Taking a pointer to an object, or dereferencing a pointer is done explicitly using `&` and `*` respectively. They support pointer arithmetic — using a pointer to calculate a pointer to another value.

Besides pointers, C++ has "references". They do not support pointer arithmetic and must always point to valid objects. Syntactically, they are used in the same way as the values they point to. The target of a reference never changes.

In Java, JavaScript, and Python all non-primitive types are reference types. Unlike pointers, they do not support pointer arithmetic. Unlike C++ "references", the target of a reference can be changed by assignment to the variable. In these languages, `.` works like `->` in C and C++: dereferencing and accessing a member.

Pointers and C++ "references" are unsafe, because their memory management usually does not take into account existing references, which allows use-after-free bugs, resulting in memory corruption. Additionally, a bug in code using pointer arithmetic can have similar consequences.

As said above, Kampa does not provide unsafe operations. Pointer arithmetic is not supported. Moreover, it is not possible to take a reference to a local variable or its part, because it could cease to exist before the reference. Taking references to parts of heap objects is also not allowed, as this could lead to memory leaks. For example, an integer reference could keep alive a tree node containing the integer, along with the entire subtree. From these two requirements follows that references can only point to whole objects on heap, like in Java or Python.

We do not need composite reference types (like Java classes), because these can be formed as a combination of a tuple type (defined above) and a simple reference type. This is the approach taken by C: pointers point to just one type (which can be a struct). C# instead provides both composite value types (`struct`) and composite reference types (`class`). This is probably to make inheritance simpler. Kampa does not have inheritance.

We will call Kampa's reference type "box", because is similar to Box in Rust and boxed types (wrappers) in Java. It is just a reference-type wrapper for a value type. There are two important differences from Java:

- Boxes in Kampa may be mutable.
- Java has only eight value types and eight unrelated wrapper classes in library. In Kampa, programmers can define their own value types, and the language itself provides the reference wrappers.

The only way to obtain a box is — unsurprisingly — boxing. It is always done explicitly, by putting the value in square brackets. Unboxing is also explicit,

Figure 1.4: Boxes syntax

```
x: Int, xBox: [Int];
x = 42;

xBox = [x]; // OK, changes reference
xBox = x; // error

xBox[] = x; // OK, changes target
xBox[] = [x]; // error

x = xBox[]; // OK, unboxes
x = xBox; // error
```

Figure 1.5: Implicit unboxing

```
x: Int, xBox: [contents: Int];
x = 42;

xBox = [contents: x]; // OK, changes reference

xBox[] = (contents: x); // OK, changes target
xBox[].contents = x; // OK, changes target
xBox.contents = x; // OK, changes target, implicit []

x = xBox[].contents; // OK
x = xBox.contents; // OK, implicit []
```

placing empty square brackets after it. Both can be seen in figure 1.4. The only exception when unboxing is implicit is member access, as in figure 1.5. When boxing a local variable, its value is copied to the box. Later modifications to the variable do not affect the box and vice versa. A new box is created each time the value is boxed, creating multiple independent copies. It is possible to create boxes of boxes, use boxes in tuples and tuples in boxes.

1.4.3 Arrays

Unlike many languages, arrays in Kampa are value types. As said in the previous subsection, there is no reason to have many different reference types. It is possible to make the usual referenced arrays by putting the array into a box. On the other hand, there are many use cases for value arrays. Section 1.7, dedicated to arrays, contains examples like rectangular arrays, including code. A practical example from our compiler is an abstract syntax tree with variable number of children. In Java, we had to solve this by having an object whose only field was a reference to an array. They could once be used for variable arguments, but currently, the compiler does not support arrays on stack (as said in 2.8), which means boxing is needed to pass an array to a function.

1.5 Functions

When evaluating the support for functions in a particular language, we are interested in the following features:

- Treating functions as first-class citizens, i.e. as any other value. Most importantly, allowing them to be passed to and returned from other functions (so-called higher-order functions).
- Anonymous functions (functions created by an expression), or at least named nested functions.
- Closures — functions that capture the variables from the enclosing scopes (lexical environment).

Most current languages support passing functions as parameters and returning them. In C and C++, this can be done using function pointers. Object-oriented languages often do not allow functions to exist without a class, so an object having only one useful method is required.

On the other hand, many languages underestimated the need for nested functions, lambda expressions, and closures. Even languages that originally did not support these features introduced support later:

- Java supports anonymous classes since version 1.1 (1997).[6] Anonymous classes have access to the enclosing method's variables, as well as the fields. To define an anonymous class implementing interface `Runnable`, one would write `new Runnable() { public void run() { /* ... */ } }`.

  Java 8 (2014) finally introduced lambda expressions, allowing the same to be written as `() -> /* ... */`.[7]

- C++ has supported local classes (with their own methods) since the beginning. But these cannot use the enclosing function's local variables (unless static or constant).

  Lambda expressions with closure were introduced in C++11.[8]

- C# was publicly announced in 2000, version 1.0 was released in 2002.[9]

  Version 2.0 (2005) introduced the support for "anonymous methods", which were superseded by lambda expressions in 3.0 (2007).[9]

- In JavaScript, treating functions like values is a core concept. It is possible to simply use the name of the function as a variable. In spite of this, strings were abused for this (such as in `setTimeout` [10]).

  Anonymous functions are supported since NS6 and IE5.5 (both 2000) and arrow functions since ECMAScript 2015.[11]

Python supports lambda expressions since version 1.0 (1994).
By contrast, C does not support anonymous functions and closures.

1.5.1 Requirements

The language support for functions should satisfy the following requirements:

- Functions should be treated as any other value. It should be possible to use a function identifier in the same way as any other identifier. The language should allow to pass them as parameters, return them, or store them in data structures.

  Conversely, the values that represent functions should be also treated as functions. No `.apply()` should be required in order to call a function received via parameter.

- All functions that have the same signature should have the same type and properties (obviously except for what they do when called).

  C++ has the following trichotomy: function pointers (dynamic, cannot carry data), functors (static, can carry data, but require template specialization), and `std::function` (dynamic, can carry data).

  Java does not have any function type, but numerous interfaces with one method. A lambda expression can mean any of them.

- Function definitions should be allowed to use variables defined in enclosing scopes.

- There should be as few different syntaxes to define functions as possible (and practical).

  In Java, `arguments -> value` seems unrelated to the declaration of methods, and `object::method` seems unrelated to both foregoing.

  In C++, lambda expressions start with square brackets and sometimes contain an arrow. Function declarations originally contained neither. Now there is one more syntax for function declarations, containing an arrow.

  JavaScript has two syntaxes for anonymous function: `(`*args*`) -> {`*body*`}` and **`function(`***args***`) {`***body***`}`**. C# too, using the keyword **`delegate`**.

- Operators should be functions, so that they can be defined in the same way, and they can be passed as functions.

  Java `Integer` has method `sum(int, int)`, so that `(a, b) -> a + b` can be replaced with `Integer::sum`. Interestingly enough, there is no `Integer::product` or similar.

  C++ goes much further, defining `std::plus` for `+`, `std::divides` for `/`, `std::bit_and` for `&`, etc.

- Conversely, functions should be usable as operators, so that arithmetic operators are not abused for their infix syntax. An example of such abuse is C++ with its `<<`, whose meaning has shifted from bit shifts to printing.

  This allows us to use operator names instead of symbols (where no conventional symbol exists, or it is not on keyboards): *vec1* `dot_product` *vec2*, *iter* `filter` *predicate*, *iter1* `zip` *iter2*, *N* `choose` *K*, and also English-like syntax: *elem* `in` *collection*, *string* `matches` *regex*, *x* `between` *(y, z)*, or *map* `with` *newElem* (for persistent maps).

1.5.2 Specification

In this subsection, we shall specify the syntax and semantics of functions.

Definition of functions

Recall that the syntax for tuple types and box types imitates the syntax of tuple values and box values respectively. In the same way, both function types and function values use the arrow symbol. However, the rules for where a function type is expected, and where it should be a concrete function, are different.

When using an arrow expression in the context of a value, it is considered a **function value**. Likewise in a local variable declaration, but not parameter. In the context of a type other than local variable declaration, it is considered a **function type**. Figure 1.6 shows examples.

- The syntax for a **function type** is `ArgType -> RetType`. A function accepts single argument and returns single result. `ArgType` and/or `RetType` may be named tuples, allowing the function to accept or return multiple values at once.

- The syntax for a **function value** is `ArgType -> RetValue`. The expression `RetValue` may be an initializer-expression, which gives us following: `ArgType -> RetType { /* ... */ }`. If additionally the code block contains no OUTPUT declarations, then the function is special to the compiler in two ways. First, the function *is allowed* use the return macro. Second, the function *is allowed* to use names defined later in the source code. Otherwise, the initializer-expression behaves as usual. Trying to use these features in functions or initializer-expressions that are not eligible will result in an error, and not different results.

Figure 1.6: Function values and function types

```
// used as value => always function value
button.onClick = [(e: ClickEvent) -> print("Clicked!")];
button.setOnClick((e: ClickEvent) -> print("Clicked!"));

// used as type => depends...

// used as type of local variable => function value
square: (x: Int) -> x * x;

// used as any other type => function type
twice: (f: () -> ()) -> { // parameter
    f();
    f();
}
IntFunction: ? (Int -> Int); // type definition
```

Figure 1.7: Capturing by value

```
str: = "original";
str = "updated BEFORE creating func";
func: () -> print(str);
str = "updated AFTER creating func";
func(); // prints "updated BEFORE creating func"
```

Figure 1.8: Capturing by reference

```
[str: = "original"];
str = "updated BEFORE creating func";
func: () -> print(str);
str = "updated AFTER creating func";
func(); // prints "updated AFTER creating func"
```

Closures

The function can use variables declared from the enclosing scope. All variables that are used within the inner function are captured at the time of creation. The values are copied: modifications in the inner function do not propagate to the parent and neither vice versa (as in figure 1.7). Consecutive invocations of the inner function also do not see each other's modifications. To enable the propagation, the variable can be defined as a box, because copying a box means copying the reference to the same value (as in figure 1.8).

Assignments

Although functions can be passed in arguments, put into structures with other values, and so on, they cannot be reassigned. This restriction is imposed to avoid unpleasant surprises. In cases where assignment is desirable, a box of function can be used, reassigning the whole box instead, like in the `onClick` example. A call does not implicitly unbox the function. To call a function that is in a box, one writes `fn[](arg)`, which is more explicit.

Calls

Calling function `f` with argument `x` is expressed as `f x`. This is common in functional languages. Parentheses can be used, as in `f(x)`, but these are arithmetic parentheses, not a part of the call syntax.

Function calls are left-associative and have the same precedence as member access, so that `f(x).m` parses as `(f(x)).m`, and `o.m(x)` parses as `(o.m)(x)`. Note that `f x.m` parses as `(f x).m`. If it were parsed as `f(x.m)`, it would mean that — substituting `(y)` for `x` — `f(y).m` would also be parsed as `f((y).m)`, which would be a disaster.

The common function call syntax `f(x, y, z)` is a combination of a call expression and a tuple expression.

Partial application

Partially applying function `f` to argument `x` is expressed as `x f`, and it has he same precedence and associativity as function call. This gives us infix notation: the expression `x f y` parses as `(x f) y`, which means "partially apply `f` to `x` and apply the result to `y`". The reason for this particular solution is following:

Consider two possible APIs for a list. One is based on methods. Elements are added like this: `myList.add(elem)`. The other API is based on functions. Elements are added using `add(myList, elem)` or (infix) `myList add(elem)`.

If one wants to obtain a function that adds elements to a concrete list, the first API makes it straightforward: `myList.add`. Using partial application, it is possible to do the same in the second API: `myList add`.

Operator precedence

Now that we can define custom named operators, we should also be able to configure their priorities. And given that non-named (symbolic) operators are no longer different from named ones, there is no reason to have their priorities hardcoded. Thanks to this, the parser can treat even operator symbols as identifiers.

Operator precedence is defined by the user or a library, using an intuitive syntax. To define `+` and `*` to have their usual relative priority, one would use `(*) + (*)` in a parser directive. It says: "when `*` appears on either side of `+`, it has higher priority".

The relation of "having higher priority" is a partial order. The parser must ensure **transitivity** [`(A) B (A)` and `(B) C (B)` imply `(A) C (A)`] and check **antisymmetry** [combination of `(A) B (A)` and `(B) A (B)` is an error].

Each operator corresponds to two elements in the partially ordered set. One element represents the operator when on the left, one on the right. Why one element is not enough can be seen from the following example (assuming the usual rules): elements `+` and `-` could not be compared. In `x+y-z`, `+` has priority, but in `x-y+z`, it is `-`. But if we do distinguish the sides, `+, L` has priority over `-, R`, and `-, L` has priority over `+, R`.

In Kampa, one writes this as `(-) + ()` and `(+) - ()`. It is also needed to relate `-, L` and `-, R`, so that `x-y-z` is unambiguous. With `+, L` and `+, R` it is irrelevant for integers, but not so for floats and strings. This yields two additional rules: `(-) - ()` and `(+) + ()`. All four rules can be expressed as one: `(+ -) + - ()`. The opposite associativity would be written `() A (A)`.

Figure 1.9 contains the parser directives for `+`, `-`, `*`, `/`, and `**` (power).

Figure 1.9: Operator precedence directives

```
// precedence
? (**) * / (**);
? (* /) + - (* /);
// associativity
? () ** (**); // right
? (* /) * / (); // left
? (+ -) + - (); // left
```

Non-operators

Throughout this thesis and the compiler, the word "operator" is never used for other syntactic constructs. What follows is an incomplete list of constructs that are *not* considered operators in Kampa. These cannot be redefined, overloaded, or passed to functions. They are built into the language, while all operators must have a declaration somewhere.

- Comma (`,`). Unlike operators, it also appears in types.
- Arrows (`->`) and assignments (`=`). They do not take evaluated operands, but rather use the subexpressions.
- Boxing (`[x]`) and unboxing (`x[]`). Square brackets are brackets, not an operator.
- Call. The use of an operator is a call itself.
- Member access (`.`). The right-hand side is an identifier, not an expression.
- Logical AND (`&&`), OR (`||`), NOT (`!`).

  Short-circuiting, expected from the first two, cannot be achieved with functions in an eagerly-evaluated language. The expressions `c ? a : b` and `!c ? b : a` should be equivalent. C++ allows overloading them, losing the short-circuiting property and breaking the condition inversion. Overloading any of the three is confusing and useless. What may be useful is conversion to boolean, as in Python, and overloading bitwise operators (`&`, `|`). Kampa currently does not support conversion to boolean, but it may be added in the future.

1.6 Immutable types

It is widely accepted that immutable objects have many advantages, including shareability (among objects and among threads) and simplification of the program. Yet, most imperative languages do not offer a way to make a variable/field immutable. Modifiers `final` in Java, `const` in C and C++ and `readonly` in C# do only part of the job: The variable itself cannot be assigned to, but in case it points to another part of memory, the modifier does not apply to the data referred to.

Because individual instances of reference types cannot be marked immutable, the decision must be done at the time the class is defined. When both options (mutable and immutable) are useful, it is necessary to define two classes, or one that only allows modification conditionally. Due to these difficulties, it is common to only define a mutable object class (examples include java.util.Calendar, java.awt.Dimension). Such a class then cannot be used as the type of a field of an immutable class. It can still *act* as being immutable by providing getters only, but this is often suboptimal (due to defensive copies) and error-prone.

When the type of a variable is not known, it cannot be marked immutable (e.g. "type `K` in `Map<K, V>` must be immutable" cannot be expressed). For a human reader, not only generic-typed variables pose a problem. One cannot tell whether a `final` variable is actually immutable without knowing the particular type.

1.6.1 Specification

Kampa distinguishes two kinds of values: MUTABLE and IMMUTABLE. Both are subtypes of READABLE. The qualifier of a field or a variable determines the restrictions and the guarantees of the value. A MUTABLE variable can be modified. An IMMUTABLE variable does not allow modifications and it guarantees the value will never change. A READABLE variable does not allow modifications either, but the value may change due to some other variable viewing the data as MUTABLE.

The qualifier is part of the type specification. Wherever a type is expected, a qualifier is also allowed. This means it may (or may not) appear in a variable definition as well as in a type definition. Multiple qualifiers can be used, e.g. one in a variable definition and one in the definition of its type, or one qualifier for the whole and one for its member. In that case, IMMUTABLE has precedence, then READABLE, and last MUTABLE. Qualifiers on boxes apply recursively to the boxed data (notice the difference from `final`, `const`, and `readonly`).

When no qualifier is specified, MUTABLE is assumed for non-boxed values ("on stack") and READABLE for boxed values ("on heap"). This mean that non-boxed values behave exactly as in C, Java, or Python. And this is safe, because modifications to these do not affect any other code. The default for boxed values is READABLE in order to (1) allow local variables and parameters to hold both MUTABLE and IMMUTABLE objects, and (2) disallow functions from modifying data through their parameters. When this default is not desirable, a single qualifier is sufficient.

In the source code, the flag `@` stands for MUTABLE, `\` stands for IMMUTABLE, and a combination (`@\` or `\@`) stands for READABLE.

Functions

Besides their parameters, functions also have access to variables in their environment (lexical scope). The variables are copied at the time of creation of the function. Reference variables are also copied — thus sharing the state of the referred data. There are three kinds of functions, depending on how they interact with the environment. For brevity, we will call these three kinds "M-function", "R-function" and "I-function".

M-function For "MUTABLE environment" — it can read or write the captured variables. Equivalent to functions in C, C++, Java, or Python.

R-function For "READABLE environment" — it can only read the captured variables. Similar to methods with `const` suffix in C++. It cannot have side-effects (except for modifiable parameters).

I-function For "IMMUTABLE environment" — it can only capture IMMUTABLE variables. The function is pure (except for modifiable parameters).

An I-function is a R-function — it satisfies all requirements, and adds more guarantees. For the same reason, a R-function is an M-function.

An IMMUTABLE variable can only contain I-functions — otherwise, it would be able to indirectly refer to MUTABLE data. A READABLE variable can only

contain I- and R-functions, because M-functions have the potential to modify it. A MUTABLE variable can contain any function.

Some functions act as methods of some larger object. Sometimes, we want to put a MUTABLE object into a READABLE variable. But its M-methods would prevent the conversion (notice that it generally does not make sense to convert M-functions to R-functions). For this reason, we define a fourth kind of function:

D-function For "disabled" — attempt to call it results in a compile-time error. It is the equivalent of void.

Any M-function can be implicitly converted to a D-function. This happens when the M-function is used in READABLE or IMMUTABLE context. The type specifications reflect this: when applying READABLE or IMMUTABLE qualifier to an M-function, it becomes D-function. Additionally, an R-function used in IMMUTABLE context becomes an I-function.

Functions are R-functions by default. To define an I- or M-function, the flag can be added to the arrow.

The following example defines the type of a simplistic *fixed*-size String list.

```
MyList: ?(
    getSize: () \-> Int,
             // the internal size field is immutable,
             // this can be an I-function
    getItem: Int -> String,
             // does not modify the list,
             // can be an R-function
    setItem: (Int, String) @-> ()
             // modifies the list,
             // must be an M-function
);
```

When using this type within READABLE context, `getSize` stays I-function and `getItem` stays R-function, but `setItem` gets disabled. The same within IMMUTABLE context, except that `getItem` will also become I-function.

Imports

It is not possible to convert a MUTABLE object to IMMUTABLE, or vice versa; a copy must be created. However, there is one exception to this rule: when importing a file, its result is always made IMMUTABLE. This also disables global M-functions and converts global R-functions to I-functions. Consequently, the only way to modify global data is during the compilation of that file.

Note on examples

The examples in the previous sections ignored the need for mutable qualifiers. This was intentional, in order to not interleave the topics too much.

Also note that the library provides I-functions that have side effects (IO). This is incorrect, but necessary due to the limitations of the implementation.

1.7 Arrays and dependent types

Kampa's arrays are unusual in two ways: first, they are value types, which is not very common in today's new languages. Second, the length of an array is a part of its type. However, this does not mean it is constant. A type may depend on another variable, in the case of arrays an immutable integer.

Syntax for arrays is `TYPE...LENGTH`. `TYPE` can be any other type, including tuples, boxes, and other arrays. All elements must have the same size: if the element type contains an unboxed array, its length must not be a part of the element. It must be outside of the array. It is possible to include a variable-length array by boxing it, since all boxes are the same size (the size of the pointer).

Elements are accessed using the same syntax as function calls. For array `a` and index `i`, `a i` or `a(i)`. If the array is boxed, the unboxing can be combined with the array access. For boxed array `b` and index `i`, `b[i]` instead of `b[](i)` or `b[]i`. The rest of this page contains examples that show this the syntax in use, but also demonstrate the versatility of Kampa arrays.

```
// example element type and one example variable of it
ElemType: ? /* ... */;
myElem: ElemType;

// sized array
array1: [length: \Int,  data: ElemType...length];

myElem = array1.data(42);

// array with length as external dependency
length: \Int;
array2: [ElemType...length];

myElem = array2[42];

// rectangle array
width: \Int, height: \Int;
array3: [ElemType...width...height];

myElem = array3[42](54);

// "jagged" array
numRows: \Int;
array4: [[rowLen: \Int, ElemType...rowLen]...numRows];

myElem = array4[42][54];

// square array
order: \Int;
array5: [ElemType...order...order];

myElem = array5[42](54);
```

1.7.1 Generic functions

One more use of dependent types is in generic functions. The function takes a parameter of type `?`, which is the type parameter. In the rest of the function type can then depend on the value of the type parameter, by using its value as a type.

The support for generic functions is still in very early stage.

The function in the following example swaps the values of two boxes:

```
swap: (T: ?, a: @[T], b: @[T]) -> {
    (a[], b[]) = (b[], a[]);
}
```

1.7.2 Allowed dependencies

As seen in the examples, the type of a variable may depend on another variable, and an item of a tuple may depend on some previous item. Additionally, the return type of a function may depend on an argument, although this second kind does not work perfectly in the compiler. This is not a problem of the type system, but a result of some limitations of the compiler. See section 2.8.

2. Implementation

One of the goals of this thesis is to provide a proof-of-concept implementation of the language. There were several alternatives:

1. Writing a compiler from scratch, together with a custom garbage collector. While giving us all the flexibility, it would consume much of the time that could be spent on language features. The produced machine code would not be more eficient, either.

2. Targeting LLVM bitcode, JVM bytecode or WebAssembly. A garbage collector exists on each of these platforms. This still involves generating the code by hand or using only limited abstraction provided by a library. A major advantage is the low runtime overhead; Kampa is not too dynamic.

3. Interpreting the AST of the program. This is the easiest option, allowing the language to be extended rapidly. It also allows reusing the features of the underlying platform, like garbage collector or operating system interface. On the other hand, it has the worst performance due to frequent branching and the absence of optimization.

4. Interpreting the AST with Truffle. While retaining the advantages of the previous item, it should perform better. Truffle partially evaluates the interpreter for the program being interpreted, thus allowing at least some optimizations, reducing code size, and sparing the processor excessive branching.

Performance is currently not a major concern, and 1 or 2 would make the development process unnecessarily slow. From the other two alternatives, 4 did not have any significant disadvantages, so we have opted for the Truffle framework for the implementation of the prototype, putting the work on an optimal compiler (i.e. 2) off until the language itself is complete, which is beyond the scope of this thesis.

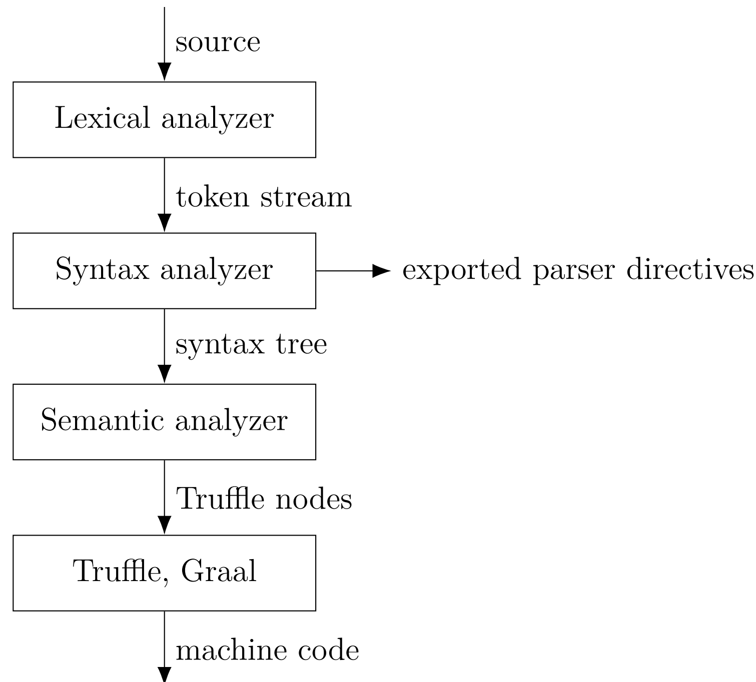
In this chapter, we describe the mentioned Truffle-based interpreter/compiler. Its source code can be found in the first attachment (see A.1), together with a launcher and a library. The launcher and the library are simple single-file projects. Not being components of the compiler, they are not described here, but in the appendix.

2.1 Structure of the compiler

The frontend of our compiler follows the traditional[12] division into lexical analyzer (lexer), syntax analyzer (parser) and semantic analyzer. The intermediate representation takes the form of Truffle nodes (instances of class `com.oracle.` `.truffle.api.nodes.Node`). The backend is replaced by the Graal JIT compiler.

When importing a file, it is first compiled to Truffle nodes (and possibly machine code) and executed in order to obtain its return value. But the return

Figure 2.1: The structure of our compiler. Source: Compilers[12], modified



value may not be enough; the importing file can also use some of the parser directives (like macros) defined in the imported file. This means that the compiler must have two outputs. Just the machine code is not enough to import the file.

2.1.1 Lexical analysis

Kampa does not have a textual preprocessor, nested comments, off-side rule, HereDoc syntax or contextual keywords, and line numbers are counted by Truffle. This makes the lexical analysis an extremely simple task. The lexer is a single hand-written Java class. A new instance is used for each source and it acts as the stream of tokens, being consumed by reading all the tokens.

2.1.2 Syntax analysis

The compiler uses a recursive-descent parser. The parser cannot be generated from a grammar due to macros and configurable operator precedence rules. Again, it is a class that is instantiated for each source file. It takes an instance of the lexer, reads all its tokens and returns the syntax tree. The syntax tree is immutable and consists of nodes (different from Truffle nodes) of three types: code block, expression and statement. The root of the tree returned by the parser is always a code block. It does not contain any parser directives or macro calls — they are all processed by the parser.

Macros

Macros are implemented as persistent immutable trees of `MacroNode`s. When parsing a macro instance (argument list), the parser walks this tree from root and

(depending on the tree) decides how to parse the arguments. Each node contains a (possibly empty) map from keywords to other nodes, which represent the named edges. To take a named edge, the parser must read the exact same keyword from the input. If no named edge can be taken, the parser attempts to read an argument. Some argument types are literal, like delimiters and parentheses. Others instruct the parser to read a unit of code (e.g. expression or statement) to be substituted in the template. If the node does not define any argument, or it is literal and does not match, error is reported.

The macros are built using a minimal set of instructions:

`merge(MacroNode, MacroNode)` Constructs the union of two existing macro trees. Used when adding the definition of a new macro and when importing macro definitions.

`empty()` Empty macro tree. The neutral element with respect to `merge`. Used when initializing the parser.

`makeEnd(Statement template)` End of the macro. When reached, nothing more is parsed for the macro, and the template is instantiated. Used when reaching the `=` in the macro definition.

`makeKeyword(String keyword, MacroNode next)` Reads a keyword from the input, then continues as `next`. Used when parsing a macro definition: `keyword` is the current token and `next` is parsed from the following tokens recursively.

`makeArgument(MacroArgumentType type, MacroNode next)` Reads an argument from the input, then continues as `next`. Used when parsing a macro definition: `type` depends on the current token and `next` is parsed from the following tokens recursively.

The original implementation did not use a tree, but only a path (represented as an array). This was problem even for the `if` statement, which has an optional `else`. So there was a way to add multiple endings to a macro. This only worked at one level; these endings could not themselves have multiple endings. The old and the new syntax are compared in figure 2.2.

Operator precedence

The requirements are described in 1.5.2.

An operator is treated as ordinary identifier until it is used in a precedence definition for the first time. At that point, it is internally assigned two numbers. These numbers do not mean anything, they just identify the operator in the partially ordered set.

The partially ordered set is represented using a matrix — a list of arrays. Instead of having a square matrix of booleans, we use a triangle matrix of signs (-1, 0, $+1$). As this is a typical algorithmic problem, where we usually care about the asymptotic complexity, let us analyze the data structure:

- Comparing elements is easy and fast: just one read from the matrix. This makes the decisions when parsing expressions approximately as fast as in compilers with precedence hardcoded.

- Adding rules is easy and slow: $\Theta(N^2)$ for a set of size N. Using the matrix representation, this cannot be better in worst case, but maybe it could be optimized to $\Theta(N)$ amortized.
- Adding new operators means adding one row to the matrix and not touching existing rows. Allocating the row is $\Theta(N)$. Appending to the list is $\Theta(N)$, or $\Theta(1)$ amortized. This gives us $\Theta(N)$ with or without amortization.

Fortunately, the latter two are not performed often, and N tends to be small. Additionally, profiling has shown that our compiler spends the most time in semantic analysis.

To parse call expressions, a modified variant of the shunting-yard algorithm[13] is used. It differs from the original algorithm in several points:

- Its input does not read individual parentheses and brackets, just expressions. An opening parenthesis starts an expression that is parsed recursively up to the corresponding closing parenthesis.
- It does not write to the output, but it has instead a single variable named OPERAND. In the beginning, the variable is null. When reading a new operand OPERAND2 from the input, it is written to OPERAND. If this write would overwrite an existing value, we instead create a call expression of (OPERAND, OPERAND2) and put that to OPERAND.
- Where the original algorithm pops an OPERATOR from the translator stack and writes it to output, we instead create a call expression of (OPERATOR, OPERAND) and put that to OPERAND.
- Where the original algorithm pushes an OPERATOR to the translator stack, we instead push a call expression of (OPERAND, OPERATOR), clearing the value of OPERAND. (Note that a call where the argument is before the function is a partial application.)
- The priorities of operators are not hardcoded. The parser consults the partial ordering described above. When comparing operators that are incomparable, error is reported.

Figure 2.2: Examples of macro definition syntaxes

```
// old
"if" (COND) {BODY}...OPTIONAL_ELSE = COND ? { BODY; } :
                                     { OPTIONAL_ELSE; }
                  ..."else" {BODY} = BODY;
                  ...              = ;
// new
"if" (COND) {THEN_BODY}
                = COND ? { THEN_BODY; } : { }
"if" (COND) {THEN_BODY} "else" {ELSE_BODY}
                = COND ? { THEN_BODY; } : { ELSE_BODY; }
```

(both omitting the parser directive prefix)

Figure 2.3: Examples of expressions that can be type or value

```
// Let us have expression AMBIGUOUS_EXPR,
// which can be a valid value or a valid type.

// Definitely a value (types do not have members).
(AMBIGUOUS_EXPR).method();

// Definitely a type (in initializer).
(AMBIGUOUS_EXPR){}.method();
```

2.1.3 Semantic analysis

Expressions have two different usages. A value expression is used to point to a memory location, or to compute a new value. A type expression is used to reference a type or construct a new type. Value expressions are indistinguishable from type expressions. For example, in the definition `x: [];`, `[]` means *the type of* an empty box, while in the assignment `x = [];`, `[]` creates *an actual* new empty box.

The syntax is never ambiguous. How the expression is used depends on the position in the syntax tree. But there are examples where it cannot be decided until having read the whole expression (as in fig. 2.3). For this reason, the expressions in the syntax tree do not store this information. How it is interpreted depends on the class that is tasked with interpreting it.

There are two such classes: `ExprToType` and `ExprToValue`. They get the syntax subtree of an expression, usually call themselves recursively, and return a `Type` or a `Value` respectively. In the cases where a value expression contains a type expression, `ExprToValue` calls `ExprToType` and vice versa. Examples include value expression `A{B}` (initializer), where `A` is a type expression. An example of the opposite direction is type with default value: `T = D`, where `T` is another type and `D` is the default value.

Code blocks are interpreted by `BlockToNode`, which iterates over the statements and converts each statement to a `StatementNode`. These are then returned in a single `AbstractBlockNode`.

Types

Types are represented by a type graph, in which each node corresponds to a `Type` instance. These are mostly immutable, but in some cases, this is not possible. A box type can directly or indirectly reference itself, as in this example:

```
LinkedList: [head: Int, tail: LinkedList];
```

The example defines four new `Type` nodes (the bold labels are arbitrary):

LinkedList: BoxType(**X**)

X: TupleType(**XItem1**, **XItem2**)

XItem1: NamedType(“head”, **Int**)

XItem2: NamedType(“tail”, **LinkedList**)

Here, **LinkedList** references itself (through **X** and **XItem1**). Should all the nodes be immutable, **LinkedList** would have to be constructed before itself. This problem is avoided by making the field `contentType` of a `BoxType` lazily evaluated. Its value is not known until queried for the first time. Thanks to this, the `BoxType` representing our **LinkedList** node may be constructed and added to the scope without investigating its contents, **X**. Once the content type is queried, its AST is interpreted, but by the time, **LinkedList** will already be constructed and in scope.

A similar approach is used for function signatures. A function parameter or return type may contain the function type itself. It is not as uncommon as it may seem: consider how often Java methods `return this`.

Values

Class `Value` represents a value during the semantic analysis. An instance of this class knows its type and the values its type depends on. It has method `makeValueNode()`, which returns the Truffle node that calculates its value at runtime. It also provides methods for accessing its members or elements, dereferencing, and similar actions.

It is usually only valid in a single function. Using instances of this class in other (even inner) functions due to a bug in our compiler would result in using inappropriate frame slots at runtime.

Class `Value` is abstract and it has subclasses `LValue` and `RValue`. The terms LValue and RValue originate in CPL[14] and are widely known thanks to C and C++.

LValue is a value that has a location. It can be written to (if mutable) or read. The following are LValues:

- variables (local, captured, parameters)
- results of dereferencing (of RValues or LValues)
- members of other LValues
- elements of array LValues
- (named) tuples of other LValues

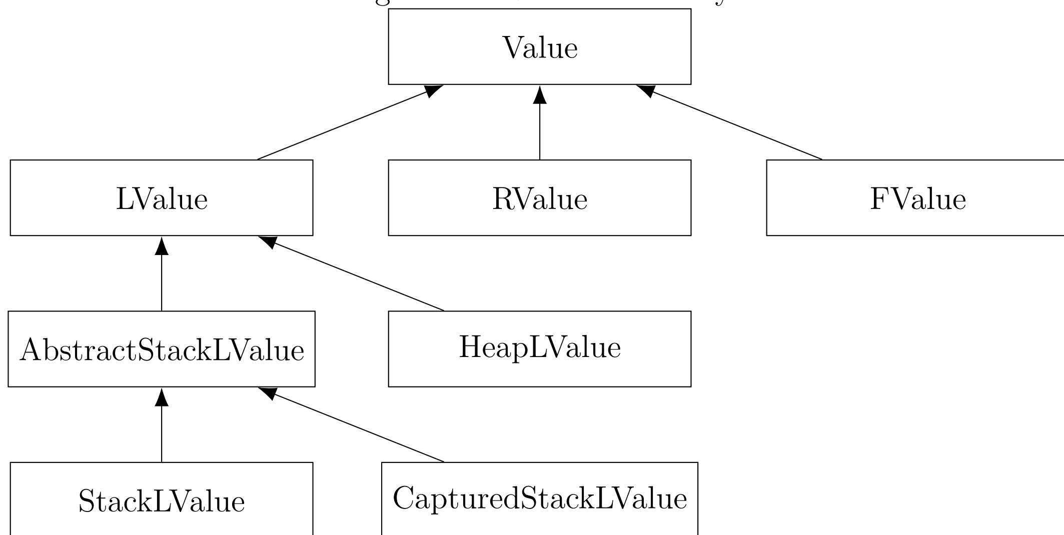
RValue is a temporary value. It can only be read. Values that are not LValues are RValues.

RValues

Class `RValue` is a `Value` wrapper for Truffle nodes. Having a Truffle node, knowing its type and the dependencies of the type, one can construct a `RValue` object. Method `makeValueNode()` of a `RValue` instance may only be called once, to avoid evaluating it multiple times by accident. Subsequent calls cause an internal exception.

Class `FValue` is used to represent a RValue that is the result of partially applying a function (see 1.5.2). This class was added to support infix notation when partial applications were not supported yet. It does not partially apply the

Figure 2.4: Values hierarchy



function, but rather waits for being called, eliminating the partial application. In Kampa syntax: converting `a fn b` to `fn(a, b)`. Now that partial applications are supported, its existence is no more strictly required, but it still makes infix calls efficient by avoiding the need to create a new function object. Because infix notation is most often used for arithmetic operations, which are fast, the difference is an order of magnitude.

LValues

Unlike `RValue`, class `LValue` does not store its `ValueNode`, but creates it when requested. It also has abstract method `makeWriteNode()`, which creates a node that writes an object it receives to the memory location of this LValue. This class has four subclasses:

AbstractStackLValue A value whose data are stored on the stack. Backed by frame slots provided by Truffle. Base class for the following two.

StackLValue An instance of this class remembers a range of frame slots, which belong to the frame of the current function.

CapturedStackLValue An instance of this class points to some other AbstractStackLValue, which is valid in the enclosing function. It has also a reference to a map of captured frame slots. When the `CapturedStackLValue` is read or written, it creates slots in the current function, and adds the slots to the map. This ensures that the captured slots are copied to the inner function at runtime.

HeapLValue An instance of this class points to some other `Value` of the same function. That value is used as the reference to the storage of this value. That is, when the heap LValue is read or written at runtime, the ValueNode of the reference is evaluated first and dereferenced (unboxed). The representation of a reference (box) is described in section 2.2.

Figure 2.5: Why ignoring the logical structure may be a good thing

```
drawCirc: (pos: (x: Int, y: Int), radius: Int) -> ...;

// usually calling like this:
myPos: (x: Int, y: Int) = (3, 4);
myRadius: Int = 5;
drawCirc(myPos, myRadius); // becomes (x: 3, y: 4, 5)
```

Type checks

As said in 1.4.1, the exact rules for type checking are implementation-defined. Our compiler uses the following approach:

When performing a type check, the types are "flattened" — tuples containing other tuples are converted to flat tuples. This is needed for two reasons: (1) to compare qualifiers correctly (`\(T1, T2)` and `(\T1, \T2)` should be equivalent), and (2) to allow the types to have different logical structures and only check the physical structure, like in figure 2.5. Flattening is done by walking the type recursively and collecting its items to a list. Boxes, functions, and arrays are considered single items, while tuples, named types and qualified types are processed recursively. The output is a list of types, a list of qualifiers, and a list of type dependencies. Each of them contains one element for every item. These lists are then compared pairwise, recursively checking items that are not primitive.

When encountering a pair of types during recursive checking of the exact same (identical) pair, the types are assumed to be compatible. This is necessary to support recursive types, and it does not harm the correctness of the type checker, because it still checks each pair of types at least once.

Additionally, the set of names is remembered during the flattening, each name being represented by a string and a range of items. How this set is used depends on the context. In assignment, the names on the right-hand side must be a subset of the names on the left-hand side. Similarly with passing parameters. In conditional expressions, both sides must be equivalent. When checking the types of two functions to be equivalent, the names of the parameters are not compared at all (a function is free to name its arguments). They are just not allowed to cross, as in $((\bullet, \bullet), \bullet)$ vs. $(\bullet, (\bullet, \bullet))$. On the other hand, the names inside the result must match exactly.

2.2 Memory representation

This section describes how values are stored in memory and passed between nodes. We use the term SIZE to refer to the number of slots/fields occupied in the parent object, whether the parent object is a frame or its part, or an `Object[]`. It is *not* the size in bytes (which is not defined in Java). Copying a value with SIZE N means copying $\Theta(N)$ memory.

Generally, the memory representation never includes any type information. All necessary checks, name resolutions (of variables, members, and labels), etc. are done at compile-time.

The type information stored by Java is not used by any of Kampa's Truffle nodes. They either expect a value to be of particular type and cast it, or they copy one location to another without trying to interpret in any way. Our compiler is similar to native compilers in this sense, except that the compiled program throws a ClassCastException instead of behaving unpredictably in case of a compiler bug.

2.2.1 Numbers

Although the compiler accepts value ranges in types, it does not interpret them in any way. All numbers have the fixed range of $[-2^{63}, 2^{63})$, that is Java long. They are always stored boxed, using `java.lang.Long`, so that they can be treated as objects. This is extremely inefficient, and would be unacceptable in a production compiler, but for our compiler it is good enough, and it allows us to handle all data in the same way in most nodes. As indicated in the previous paragraph, we could get away with a union of long and pointer, but this would be unsafe and Java does not support this.

The SIZE is always 1.

2.2.2 Tuples

Tuples are similar to structs in C, except that Kampa does not support bit fields and we do not have to deal with alignment. The SIZE occupied by a tuple in the parent object is the sum of the SIZEs of all items. The memory representation of the tuple is the concatenation of the items representations. Specifically, the empty tuple (void) has an empty memory representation - its SIZE is 0.

2.2.3 Names and qualifiers

Names and qualifiers do not influence the SIZE and memory representation in any way. They are not included in it; the compiler only uses them during the analysis.

2.2.4 Boxes

A box is represented by an `Object[]`. The length of the array is equal to the SIZE of the contents. The SIZE of the box itself (i.e. the reference) is always 1, which is needed to store the reference.

2.2.5 Functions

A function is represented by a `FunctionData` instance. Class `FunctionData` has two fields: a Truffle call target, which represents the code of the function, and an Object which contains some data required by the function. The call target can be likened to a function pointer in C or a virtual method table in Java. It is shared among all instances of the same function. The mentioned Object field may be different for each instance of the function. Its type depends on the call target and we will describe it later in 2.6. The SIZE of a function is always 1, which is needed to store the reference to the `FunctionData`. This also holds for disabled functions (D-functions, defined in 1.6.1).

2.2.6 Arrays

The memory representation of an array is the concatenation of the representations of its elements. The type of all elements in the array must be the same and the SIZE of an element is only allowed to depend on external values, so all elements have the same SIZE. From this follows that the SIZE of the array can be computed as a product of the element SIZE and the number of elements. The number of elements is a single numeric value on which the type of the array depends. While it is not a compile time constant, it can be calculated without reading the array.

2.2.7 Types

A type definition does not use any runtime information, so its memory representation is empty and its SIZE is 0, as with void. The SIZE of type parameters, on the other hand, is 1, as they have the same representation as numbers. The number is the SIZE of the type passed in that parameter. This allows generic functions to take parameters of various sizes without having to be specialized for each type.

2.2.8 Instances of type parameters

As said in the previous subsection, the SIZE of a generic type is equal to the value of the type parameter. Its representation is not known to the generic function.

2.3 Temporary representation

A Truffle node is evaluated by calling one of its methods, passing parameters to it, and receiving its return value. Java requires us to hardcode the number of parameters (naturally) and the number of return values (limiting it to at most one). However, we would like some nodes to take and return values of arbitrary SIZE, sometimes not even a compile-time constant. There are two ways of bypassing this limitation (excluding obviously bad options like static storage):

1. Pass the parameters and the results in a temporary `Object[]`.
2. Allocate more frame slots at compile time and use them to store the parameters.

The second option has the advantage that it does not allocate anything at runtime and it can be later improved to not box primitive values. However, it still requires the SIZE to be a compile time constant, which is not the case with arrays. Moreover, the code would be much more complex — even without having to fix the arrays problem. The frame descriptor would have to be passed everywhere, ready to allocate new temporary slots.

For these two reasons, the temporary `Object[]` option was preferred. Because we are only interested in the elements of the array (not its type, length, or identity), two special (but very common) cases can be optimized: An empty array may be replaced by a singleton object or null, and one-element array may

be replaced by the element itself. But deciding the temporary representation according to the SIZE introduces one more problem, namely requiring the additional code for additional cases. This cannot be entirely eliminated, but we can at least make sure it does not make it to the resulting code. It is necessary to pick the appropriate branch at compile-time. Values that are always empty (SIZE = 0) are represented by null. Values that are always singletons (SIZE = 1) represent themselves. Values with SIZE greater than 1 *or* unknown at compile-time are packed into an array.

2.4 Truffle nodes

The tree of Truffle nodes is generated completely during the analysis. The nodes do not ever change at runtime or use any speculation features offered by Truffle. They also do not contain any type information (with the exception of FileRootNode, see below).

2.4.1 Root nodes

These nodes extend Truffle's abstract class `RootNode` so that they can serve as the roots of their function trees. Although technically possible, Kampa does not use these nodes at any other position in the tree.

Each FileRootNode is executed exactly once during the compilation of the file. It executes the file's root code block (more precisely, BlockNode) and then reads from stack the result that should be returned. This is the only Node class that knows its type. It returns it to the caller together with the resulting value in a KampaObject. This is necessary to communicate the type of the file result to the file that imported it, or to the launcher.

All other root nodes are described in section 2.6 about functions.

2.4.2 Block nodes

These nodes represent sequences of statements. Class NonemptyBlockNode uses Truffle's BlockNode, as recommended. But BlockNode requires that a block is not empty. This does not accord with Kampa rules for blocks. To provide a replacement in places where a block node is expected, empty blocks are compiled to EmptyBlockNode. AbstractBlockNode is the common base class of both classes.

2.4.3 Statement nodes

Statements are the elements of blocks. Abstract class Statement has four subclasses. An ExprStatementNode represents an expression that is used as a statement. Most commonly, the expression will be a function call or an assignment, although this is not required. A DeclNode represents a declaration. It may contain AssignmentNode children that initialize the value that is being declared. Note that a DeclNode does *not* declare the variable or allocate it on the stack, this is done at compile time. A GotoNode throws a GotoException, which is then caught by an enclosing LabelNode. Depending on the information in the

exception object, the LabelNode may restart itself (used as "continue"), terminate itself (used as "break"), or rethrow the exception if it is not intended for this LabelNode.

2.4.4 Value nodes

We will not describe all value nodes, because there are relatively many and they are not very important in the overall structure. Value nodes related to functions are described in section 2.6. The select few are:

AssignmentNode It has two children: a write node and a value node. First gets evaluated the value node. Its result is passed to the write node and also returned.

CondNode Evaluates a condition and depending on the result, evaluates either the "then" branch or the "else" branch (both being expressions). Returns the result of the picked branch. This is the main method of branching, but there are others (e.g. the short-circuiting AndNodes and OrNodes).

HeapReadNode Evaluates its child ValueNode, dereferences the result, and returns a slice of it.

StackReadNode Reads the contents of some slots in the virtual stack frame, provided by Truffle. One StackReadNode instance always reads some fixed sequence of frame slots.

InitExprNode Executes a sequence of AssignmentNodes instances, like DeclNode. Then executes a BlockNode. Then executes a ValueNode and returns its result. These three actions may not seem related, but the frontend generates InitExprNode instances in such a way, that they are. The assignment nodes come from the type expression that forms the header of the initializer expression — they assign the default values. The block node is the body of the initializer. The ValueNode is then the Node that reads the newly initialized value from stack, and returns it as any other ValueNode. It is common that there are no AssignmentNodes and the ValueNode returns void — which is the case of plain blocks that are not even intended as initializers.

TupleNode Evaluates all its children and concatenates their results into one value.

2.4.5 Write nodes

These nodes write (copy) a value to a memory location. A write node does not have a ValueNode that generates the value for it; the value is received from the parent node (usually an AssignmentNode). This allows us to join write nodes to tuples in the same way as value nodes (using a DestructuringWriteNode instead of a TupleNode).

HeapWriteNode Evaluates its child ValueNode, dereferences the result, and rewrites a slice of it with new data.

StackWriteNode The same as StackReadNode, except that it writes the slots.

DestructuringWriteNode Breaks a value into some smaller values and forwards these smaller values to its children.

2.4.6 Size nodes

Size nodes are used to calculate the SIZE of a value. They never take into account the actual length of the object array. Actually, they are often used before the construction of the array precisely to know its length. As the offset within a value is the SIZE of a specific prefix, size nodes can also be used for that.

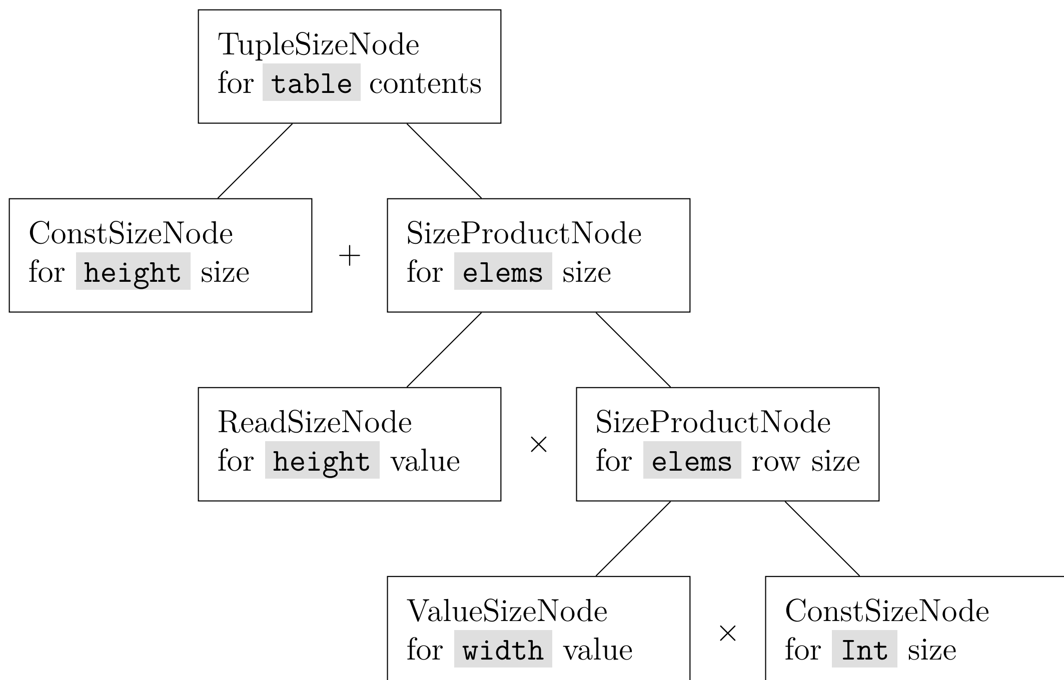
The SIZE of a value may be a constant, or it may depend on some value — either external or a member of the value in question. The result reported by a size node always depends on the structure of the size node, but only sometimes on the value itself. A size node is always created for a specific type. When used with a value of another type, it may throw an internal runtime exception or just return a wrong result. This could only happen due to a bug in this interpreter, the guest program does not have control over this.

Figure 2.6 contains an example of a size node tree. The difference between ReadSizeNode and ValueSizeNode is that *Read*SizeNode *reads* the size from the current object, while *Value*SizeNode evaluates another *Value*Node to obtain the size.

Figure 2.6: Example of a size node tree

```
// the separate width is just for the sake of example
width: \Int;
table: [height: \Int, elems: Int...width...height];
```

The size node for the contents of `image`:



Size nodes are created from other size nodes (at compile time), like when building a mathematic expression. The class `SizeNode` provides static factory methods for doing this — the classes that make up the structure are not public. The reason for using factory methods is that public constructors like `new SizeSumNode(szn1, szn2)` would make algebraic simplification impossible. Currently, this potential is unfulfilled, letting Graal take care of it.

2.5 Library interface

Source files written in Kampa can import other source files to get access to functions exported by them. Our compiler further extends this to functions written in any JVM language. It is required that these functions are compiled before running our compiler, and made accessible in a class on the classpath. The mentioned class must implement the following interface:

```
import java.util.function.Supplier;
import java.util.List;
import java.util.Map.Entry;
import java.util.function.Function;

Supplier<List<Entry<String,Function<Object[],Object>>>>
```

The process of importing such a library is as follows:

1. A Kampa source file imports a special path consisting of `java://` and the fully qualified name of the class, e.g. `java://org.example.kfx.KampaFX`.

2. The compiler uses the current class loader to load the requested class.

3. The compiler reflectively instantiates the class using the parameterless constructor.

4. The compiler calls the `Supplier.get()` to obtain the List.

5. The compiler iterates the list, processing each `Entry` as follows: The String key is interpreted as a type expression. It must be a named function, e.g. `myFunc: (T1, T2) -> T3`. The Function value is used as is.

6. All the functions are put into one tuple and returned to the importing file.

Note that all this is done at compile-time, so there is no runtime overhead due to the reflection, parsing, or searching the function.

The interface is designed in this way so that the libraries do not depend on the compiler. A better solution would be to create an interface on which the compiler *and* the libraries would depend. However, the build system makes dependencies between subprojects unnecessarily complicated, so we use Java library classes as this common dependency instead.

2.6 Functions

Functions are values that can be called. Their runtime representation (class `FunctionData`) has already been described in 2.2.5. At compile time, their type is an instance of class `FunctionType`. This class has two fields. Field `signature` is lazily evaluated (see 2.1.3) and it contains the function's argument type, return type, and environment qualifier (see 1.6.1). Field `impl` contains an object representing the body of the function. There are currently three options:

Indirect The body of such function is not known. It could even be a different function every time. This is the case of functions that are received in parameters, or located in objects. Calls to these functions are compiled to `IndirectFunctionCallNode`s.

A node of this class relies on Truffle's `IndirectCallNode`. When evaluated, it uses it to call the call target it finds in the `FunctionData`.

Direct Known at compile time. Functions defined locally, or imported from other Kampa files are direct. Calls to these functions are compiled to `DirectFunctionCallNode`s.

A node of this class relies on Truffle's `DirectCallNode`, which holds the call target.

In recursive functions, this is a chicken-and-egg problem. The call node cannot be constructed, because the root node does not yet exist. To solve it, an `IncompleteValueNode` is returned. This node is completed by filling in the call node as soon as the function is finished.

Java functions A "Java function" is known to originate from a particular Java import. Calls to these nodes are compiled to `JavaFunctionCallNode`s.

Such a node is not a real call node. It includes the imported Java function object and calls it when evaluated. As a consequence, it is inlined by Truffle, so for example when the function adds two numbers, the node effectively becomes an addition node.

Note that the `FunctionData` must always include a call target, although it is unused many times. This is necessary due to the possibility that a **direct** or **Java** function is passed to a function, implicitly converting it to **indirect**. The compiler does not support any conversions changing the representation, such as adding a call target to the function data.

Class `FunctionData` has one more field, called `arg0`. This field has type Object, as its type (as well as the type of the root node) depends on how the function was obtained. There are three ways of doing so.

Definition

Usual function definition (using `->`) compiles to a `FunctionCreationNode`. This node copies the slots used by the function to an `Object[]` and uses this as `arg0`. The root node (of class `FunctionRootNode`) then uses the `arg0` to initialize corresponding slots inside the function. The `Object[]` is allocated earlier: at the beginning of the nearest enclosing `LabelNode`. This allows the function to be captured by a function defined higher in the source code.

Java import

Java functions do not have any `FunctionCreationNode`. Only one instance of `FunctionData` exists for each. Its `arg0` is null. The root node has type `JavaFunctionRootNode`, but it is only needed when it is called indirectly.

Partial application

Partial function application compiles to a `PartialApplicationNode`. This node puts the original `FunctionData` and the partial argument value together into a new `FunctionData.arg0`. The types of the root node and `FunctionData.arg0` are following:

```
PartiallyAppliedFunctionRootNode
PartiallyAppliedFunctionRootNodeArg0
```

Exactly one such root node is created for each partial application node, because it is not possible to create them at runtime. The root node joins the partial argument from `arg0` and its own argument. Then it uses the original `FunctionData` to indirectly call the original function with the updated argument.

2.7 Dependent types

The type of a value may depend on another value. The array type depends on a value through its length and generic types depend on the type parameter, which is itself a value. Internally, the compiler distinguishes two types of dependencies: external and internal. Recall the example used with size nodes:

```
// the separate width is just for the sake of example
width: \Int;
table: [height: \Int, elems: Int...width...height];
```

Here, `width` is an external dependency of `table`, while `height` is its internal dependency.

2.7.1 External dependencies

Types do not have external dependencies. Values do. Every `Value` instance contains a list of `AbstractStackLValue`s it depends on. The type of the value may then refer to an element in the list using a `ParameterDependency` object, which holds the index to the list.

Class `ParameterDependency` implements interface `Dependency`. The difference and another implementation will be described soon. Also note that by saying “refer to list”, we do not mean holding a refernce to the list, but only an index.

If the type of the value in question is a tuple, the types of its items do *not* refer to the same list. Instead, the tuple contains a list of `Dependency` objects for each item, and the parameter dependencies inside the item type refer to this per-item list.

2.7.2 Internal dependencies

The other implementation of `Dependency` is `SiblingDependency`. A sibling dependency does not require any item to be present in any list. It instead holds an offset of a value inside of the same tuple. The offset is relative to the start of the tuple. Note that the dependency does not have to be an item of the tuple, it may be an item of an item, or deeper.

2.7.3 Dependencies in function types

A function type behaves similarly to tuple type — a pair of the argument and result. Class `FunctionSignature` has the following fields:

```
Type argType;
ImmutableList<ParameterDependency> argDeps;
Type retType;
ImmutableList<Dependency> retDeps;
```

The the parameter dependency indices in `argType` point into `argDeps` and similarly with `retType` and `retDeps`. The return type may have both parameter and sibling dependencies. The reason for this is that the return type may depend on the argument, but not vice versa.

2.7.4 Consuming dependencies

All this was about forwarding referencies to children. Now about consuming them. All that is needed for a type to use a dependency is to have a `Dependency` object. In a way, forwarding the dependencies is just one case of consuming them. There are two types that consume dependencies without forwarding them to children: `UnknownType` and `ArrayType`.

Class `UnknownType` is trivial: it has no fields at all. It is always assumed to have a parameter dependency with index 0.

Class `ArrayType` is interesting in that it forwards some parameters *as well as* consuming one. It has the following fields:

```
Type elemType;
ImmutableList<ParameterDependency> elemTypeDeps;
ParameterDependency countParam;
```

The list `elemTypeDeps` is used by parameter dependencies inside `elemType`, and `countParam` points (indirectly) to the value that gives the element count.

2.7.5 Other types

To wrap up: tuple types forward their dependencies to their children explicitly, on a per-item basis. So do function types and array types. Sibling dependencies are only allowed in the dependency lists of (1) tuple items, and (2) function returns.

The other types like `BoxType` or `NamedType` just forward all dependencies to their children.

2.7.6 Additional rules

1. The parameter dependencies must be consumed in order, e.g. parameter dependency with index 3 *before* one with index 4. In other words, they must be put into the lists in order of consumption.

2. Each parameter dependency must be used *exactly* once. If two items in a tuple depend on the same external value, that external value must be present twice in the list.

3. Recursive types have as few dependencies as possible. The type does not pass a dependency to itself just to be able to pass it to itself on the next level.

These three rules ensure that there is only one allowed representation, which makes type checking easier. Additionally, this form is very easy to build and maintain: no set operations are needed, just concatenation.

It contains a lot of redundancy, which has the disadvantage of memory consumption and allocation overhead, but it may also lead to earlier discovery of bugs.

2.7.7 Operations

Reinterpreting a value — adding or removing name or qualifier — does not change the list of external dependencies. Neither does boxing or unboxing.

Slicing — tuple member or array element access — may remove some external dependencies, due to some dependent items not being part of the result. But it may also create new external dependencies from internal dependencies. When this happeds, the value of the dependency must be copied to a temporary local variable, so that it is reachable even after the slicing. This poses a problem, because `Value`s cannot directly add code to the current function. We solved it by storing additional information in the resulting `Value` object. When `makeValueNode` or `makeWriteNode` is evenually called, the information is used to build a `DepCopyNode`, which becomes a part of the resulting value node or write node.

Joining tuples may convert some external dependencies to internal dependencies: for example when joining an integer variable and an array whose size is exactly that variable. This is implemented in `ValueBuilder` and its base class `ValueBuilderBase`.

A similar task emerges in compiling functions: to find the dependencies on arguments in the return value. Unresolved dependencies must be handled differently: `CapturedStackLValue`s must be replaced by the original objects (so that the function type dependencies make sense in the parent function), and `StackLValue`s must be reported as errors (the return type depends on a local variable, which is necessarily unknown to the caller). This is implemented in `FunctionValueBuilder`, also inheriting `ValueBuilderBase`.

2.8 Limitations

While the compiler is able to run all the examples in attachment A.2, it has some serious limitations:

Runtime performance. As said in 2.2, every number — even boolean — is represented as a 64bit number, and a boxed one at that.

Arrays on stack are not supported. They do not fit into Truffle's concept of stack frames — a constant number of slots. Implementing them was not worth the complexity and overhead incurred even on code not using them.

Dependent type checks are too restrictive. They do not consider two types equal if they depend on different memory locations, even if the values are provably the same (e.g. because one had been copied to the other). Additionally, numeric constants are not accepted for array sizes, for the same reason (they do not even have a memory location). Value numbering, a technique used for the same purpose in optimizers, can help. I am going to explore this opportunity in the future.

The architecture of the semantic analyzer is based on mere tree transformations (syntax tree → Values&Types → Truffle nodes). This makes the static analysis harder and it is a hindrance to resolving the previous point. As another consequence, variables are not checked to be initialized. The necessity of copying dependencies when slicing (see 2.7.7) could be avoided entirely, if temporary variables were used for all subexpressions.

Interoperability with other Truffle languages is limited to executing Kampa functions and converting Kampa objects to numbers. No tuple member accesses, no array element accesses. Class `ConvertedKampaObject` is close to providing this functionality, but it does not export the necessary messages. This is fixable with enough time.

Modules. Packages are used for the internal structuring of the program. Modules were planned to hide all the public classes, but caused problems in GraalVM. They are currently not used. The visibility is not much of a problem, as the compiler is never used as a library and it will never be.

Conclusion

To conclude, let us get back to the goals of the thesis:

> 1. Define a set of requirements that will make the language convenient for the developer, while still mapping to machine instructions reasonably well.

We have defined the requirements in the first chapter to be: static typing, memory safety, support for value types, simple reference types (one for each value type), unique function type, closures, operators as functions, immutable types, dependent types (for array sizes), and generic functions.

Some of these requirements undoubtedly contribute to the programmer's convenience (memory safety, closures). Some on the other hand depend heavily on personal preferences and the task itself (especially static typing).

Concerning the mapping to machine instructions: none of the requirements do harm, except for the memory safety requirement. Static typing saves instructions, dependent array types obviate the need for storing array sizes inside arrays. Closures do not help, but still they are implemented more straightforwardly than interfaces.

> 2. Design a relatively ordinary general-purpose programming language that meets these requirements. In addition, the language should have simple and orthogonal semantics, while being at least as expressive as current programming languages.

We have specified Kampa along with the requirements.

> 3. Provide a proof-of-concept implementation in order to verify the feasibility and test the properties of the language. Since this is just a prototype, performance and a comprehensive library are not the main objectives.

The implementation is described in the second chapter. It is in the attachment A.1. Albeit far from perfect, it manages to execute the example programs.

Future work

The compiler could be improved on extensively. Its limitations are named in 2.8. It should also be retargeted to LLVM bitcode, JVM bytecode or WebAssembly, because for a static language even the best interpreter will never be as efficient as a true JIT or AOT compiler.

Counting only 123 lines of code, the library is extremely rudimentary. For comparison: the GNU C library has approximately 15000× as much.

But first, the language must be completed. As said in 1.4.1, the exact rules for tuple type checks have not yet been drawn up. What the proof-of-concept implementation does is by far not perfect. And there are many features to be added, most of them already being present in other languages. These include the inference of type parameters, overloading, and various minor improvements.

Bibliography

[1] TIOBE index. https://www.tiobe.com/tiobe-index/, 2020. [Online; accessed 26-May-2020].

[2] Stack Overflow developer survey 2019. https://insights.stackoverflow.com/survey/2019, 2019. [Online; accessed 26-May-2020].

[3] Python 3.8.3 documentation: typing – support for type hints. https://docs.python.org/3/library/typing.html, 2020. [Online; accessed 01-June-2020].

[4] PHP manual: Function arguments. https://www.php.net/manual/en/functions.arguments.php#functions.arguments.type-declaration, 2020. [Online; accessed 01-June-2020].

[5] Common Vulnerabilities and Exposures. https://cve.mitre.org/, 2020. [Online; search "buffer overflow" or "use after free"].

[6] The Java programming language: Changes for java 1.1. http://java.sun.com/docs/books/javaprog/firstedition/1.1Update.html, 1997. [Online; accessed 14-Feb-1998].

[7] The Java language specification. https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf, 2015. [Online; accessed 30-May-2020].

[8] cppreference.com. Lambda expressions. https://en.cppreference.com/w/cpp/language/lambda, 2012. [Online; accessed 30-May-2020].

[9] The history of C#. https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-version-history, 2020. [Online; accessed 30-May-2020].

[10] MDN: Web APIs: setTimeout. https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/setTimeout, 2020. [Online; accessed 30-May-2020].

[11] S. Holzner. *Inside JavaScript*. New Riders, 2003.

[12] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, & Tools (Second Edition)*. Addison Wesley, 2006.

[13] E. Dijkstra. Algol 60 translation. https://ir.cwi.nl/pub/9251, 1961. [Online; accessed 30-May-2020].

[14] cppreference.com. Value categories. https://en.cppreference.com/w/cpp/language/value_category#History, 2016. [Online; accessed 28-May-2020].

[15] GraalVM reference manual: Embedding reference. https://www.graalvm.org/docs/reference-manual/embed/, 2020. [Online; accessed 27-May-2020].

List of Figures

1.1 Problems with C macros 8
1.2 Tuple syntax basics 9
1.3 Destructuring of tuples without names 9
1.4 Boxes syntax 11
1.5 Implicit unboxing 11
1.6 Function values and function types 14
1.7 Capturing by value 15
1.8 Capturing by reference 15
1.9 Operator precedence directives 16

2.1 The structure of our compiler 23
2.2 Examples of macro definition syntaxes 25
2.3 Examples of expressions that can be type or value 26
2.4 Values hierarchy 28
2.5 Ignoring the logical structure in type checks 29
2.6 Example of a size node tree 34

A.1 Commands to compile and execute the project 45

List of Abbreviations

AOT: ahead of time

API: application programming interface

AST: abstract syntax tree

IDE: integrated development environment

JIT: just in time

JVM: Java virtual machine

OOP: object-oriented programming

A. Attachments

A.1 Kampa compiler and library

The first attachment consists of a POM project (artifact ID `kampa`) with three subprojects:

- Subproject `kampa-compiler` provides the definition of Truffle language Kampa and its interpreter. It does not contain an entry point, but when on classpath, it can be discovered by Truffle and made accessible through Polyglot. We call it "compiler", because that is what we get in conjunction with Truffle.
- Subproject `kampa-library` can be likened to native libraries of some languages. It defines functions that cannot be implemented in Kampa itself. These include basic arithmetic operations like addition, and two IO operations: `putchar` and `time`.
- The main class of subproject `kampa-launcher` uses Polyglot[1] to execute Kampa source files. Formally, it does not depend on the other subprojects, but it requires that *some* implementation of Kampa is present. For convenience, it contains an `exec:java` goal that starts `kampa-launcher` with `kampa-compiler` and `kampa-library` on classpath.

Use the commands shown in figure A.1 to compile the project and launch the compiler with a specified source file. You will need GraalVM on `$JAVA_HOME`

[1] Polyglot API provides a way to run programs written in guest languages (like Kampa) from the host language (Java) or other guest languages.[15]

Figure A.1: Commands to compile and execute the project

```
mvn compile
mvn -pl launcher exec:java -Dexec.args=FILENAME.kampa
```

A.2 Examples

The second attachment is a folder containing example programs.

- File `base.kampa` is the base of the standard library (and the only currently existing part). It exports definitions of the usual types, control structures, operators and functions. It cannot be executed; it is meant to be imported from other files.

- Files `base.test{0..3}.kampa` test most of the functionality exported by `base.kampa`. They do not use the `assert` function to verify the results, because a malfunction of the compiler or library could break the `assert` function itself. Human operator or diff is needed.

  `base.test3.kampa` in particular tests timing functions, and thus can be used as a benchmark, but it does not do any representative sequences of operations, just a binary counter in an array. Equivalent C version (using array of longs, but not boxed Longs) performs approximately ten times better.

- File `prng.kampa` defines a random number generator interface, and an implementation (using linear-feedback shift register). They are meant to demonstrate OOP, but it had to be modified in order to work with the non-finished compiler. It cannot be directly executed, but it can be imported to get the interface and the implementation.

- File `qsort.kampa` contains an implementation of quicksort. It demonstrates dependent types, generics, inner functions, and passing functions/ operators. It cannot be directly executed, but it can be imported to get the sorting function.

- File `prng+qsort.test.kampa` tests the previous two, by generating a random array and sorting it in multiple ways. It is not comprehensive and does not check the results, just prints them.