FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Lukáš Rozsypal

Kampa: from a prototype to practical usability

Department of Distributed and Dependable Systems

Supervisor of the master thesis:	doc. Ing. Lubomír Bulej, Ph.D.
Study programme:	Computer Science
Study branch:	Software Systems

Prague 2023

I would like to thank my supervisor Lubomír Bulej, my friends, and my family.

Title: Kampa: from a prototype to practical usability

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 an experimental general-purpose imperative programming language influenced by functional programming. While Kampa offers several potentially useful features, its usability is limited by some properties of its current implementation as well as of the language itself. We add support for coroutines and parameter inference, making the language more expressive. We also make changes to the syntax that improve code readability, in particular reducing the tendency to "line noise". We refine the implementation to remove several arbitrary restrictions regarding generic and dependent types, enabling generic type definitions (among others). Lastly, we demonstrate the practicality of the result by writing a library providing collections, optional, asynchronous programming, generators, and string utilities.

Keywords: programming language design Kampa programming language strongly typed languages

Contents

Introduction

Kampa is a programming language developed by the author in his bachelor thesis [1]. Rather than being a test of some extraordinary computational model, the aim was just to design a general-purpose imperative programming language with emphasis on clean language design.

One of the main goals of the original thesis was a language that is simple and orthogonal, meaning that it consists of a small set of primitives that can be used independently of each other. For example, Kampa (similarly to e.g. C and Rust) offers one tool to create a homogeneous collection type (array), one tool to create a product or record type (tuple in Kampa, `struct` in C), and one tool to create a reference type for a given type (box in Kampa, pointer in C). The programmer can but does not have to combine these tools (e.g. create a pointer to a `struct` or a boxed tuple). Compare this to e.g. Java or most scripting languages, which do not allow using these tools separately. The only way to create a product type in Java is a class, which at the same time introduces a reference. Similarly, Java's array type is implicitly a reference to a structure with two fields (`length` and the array data).

Arrays in Kampa avoid both the implicit reference and the implicit length field. Instead of being stored in a field, the length is specified in the type of the array. The length can be a compile-time constant, but it does not need to be. This may introduce a dependency of some types on run-time values, making the language dependently typed. The same mechanism is then further used to implement generic functions. A function can be made generic by adding a parameter of type Type Variable, which is then available for use in the types of the remaining parameters.

Similarly, it should be possible to define a generic type as a function returning a Type Variable. However, the current implementation of Kampa (as attached to the bachelor thesis) does not support this and even if it did, it would not be able to type-check the usages of such a generic type. These and related limitations are addressed in Section 3.5. Workable generic type definitions are crucial for library generic types such as `Optional` (Section 4.1.1) and collections (Section 4.1.2).

Additionally, there are several limitations caused by the language itself. This thesis improves the syntax (Section 2.1) and adds two new language features. Type parameter inference (Section 2.3.2) allows calling generic functions without having to specify the types of the generic arguments. Coroutines (Section 2.3.5) offer a general way to describe generators and asynchronous functions. Additionally, this thesis splits the interpreter into two separate parts, one analyzing the source code and producing bytecode (in a Kampa-specific format, see Section 2.4.1), and the other interpreting the bytecode.

1. Context

To make this thesis more self-contained, we present a summary of the syntax and semantics of the language as specified and implemented by the original thesis. This will be our starting point for the next two chapters.

Some details about the syntax are intentionally skipped for now. We will return to these in Section 2.1.

1.1 Kampa program structure

As in most imperative languages, the source consists of value expressions, type expressions, and statements (we abandon this distinction later in this thesis). Any expression can be used as a statement (in the form of an *expression statement*). Statements usually appear in *code blocks*. Similarly to e.g. Rust and GNU C, a code block can be used as a value expression.

Some expression statements are considered as declarations – their members enter the current scope. A declaration can additionally be marked as *output*. Output declarations contribute to the value of the block they appear in. The last statement is *not* used for this (unlike Rust or GNU C). The value of a block is a tuple – if the block contains no declarations it will be an empty tuple.

The top level of a file (the root of the syntax tree) is a regular code block; it consists of statements and it is not restricted to declarations only. A file may import another file using the *import expression* (`\"path/to/file.kampa"`). This expression executes the other file and evaluates to the value of its code block, i.e., a tuple of variables exported by that file using top-level output declarations. This happens immediately when the expression is parsed, never at runtime.

The interpreter accepts one file on its command line – that file is expected to export a single unnamed function (the main function). Once all files are parsed, type-checked and converted to an internal representation, the interpreter initiates program execution by calling the main function.

As such, although the implementation is incapable of producing code that can be run some time later, it strictly distinguishes between compile-time and runtime.

1.2 Type system

Kampa is a structurally-typed language with a limited set of built-in families of types. The following page lists the nine built-in families, along with the source code notation. For most of the types, there is a notation for the type itself and a notation for its values. These two notations are similar or the same. Whether an expression is to be interpreted as a type or a value (that is, whether it is a type- or value-expression) depends on the context. For example, the parameters in a function definition are type-expressions.

Number is the type of conditions, and array indices/sizes. A value is created using numeric or character literals. The type is written as a range (`min..max`) syntactically, but this information is dropped during parsing (some future version of the language will hopefully be able to take it into account). The library defines several aliases for this type, including `Int` and `Bool` . Its representation is currently always a 64-bit signed integer.

Tuple type is the product of zero or more types. Both the type and its values are denoted with a comma-separated list of item types or values respectively. Tuples are usually parenthesised due to their low precedence. For example: `(Int, Int)` , `(42, 43)` . Tuple is a value type, similarly to C structs.

Function type is determined by its argument type and return type. The syntax for function types is `ArgType -> ReturnType` . Values are created using lambda expressions with the same syntax: `ArgType -> returnValue` . In addition to the code itself, values of the function type may also carry additional data (most often the function's closure).

Named type is a thin "wrapper" for another type. It serves as a tag which can be used to retrieve the value from a tuple or from the scope. Both the type and the value is written as `identifier: expression` . Most often used in a tuple (`(x: 42, y: 43).x` evaluates to `42`) or a function argument (`(x: Int) -> x * x`).

Box is the pointer/reference type. Unlike usual pointers, it always points to a separate object on the heap. It is not possible to take a pointer to an existing value without copying that value. Both the type and the value is constructed by surrounding a type/value in square brackets. It can be used to define cyclic data structures or create mutable shared state. Boxes do not support pointer arithmetic and cannot be null.

Array is a sequence of values of the same type. The number of elements is part of the type. However, it does not have to be a compile-time constant. The syntax for array types is `ElementType...sizeValue` . Array values can be created using the same syntax (`elementValue...sizeValue`), which evaluates its left-hand side repeatedly. Array elements are stored directly in the value of the array, no pointer is involved.

Type Variable is the type of all types (including itself). The syntax for the Type Variable type is `?` and the syntax for a Type Variable value is `?TypeExpr` (e.g. `?Int` or `?(x: Int, y: Int)`). Type variables can be used like any other type expression. In memory, each type variable is represented by a single 64-bit value, which is the size of the type.

Type is similar to the previous. Unlike the previous (which is a single type), this is a family of unit types. Each type has its own Type type. For example, the Type for `Int` and the Type for `Int -> Int` are distinct types. This family of types can be used to define aliases for existing types. It is a mere convenience and is not important for the type system. The syntax for the type of a Type is `?TypeExpr` (i.e. the same as *value* of Type Variable).

Unknown is used for variables and values whose type is not known. It is a dependent type that depends on the value of some Type Variable.

1.2.1 Dependent types

Two items of the above list are dependent types: arrays and unknowns. An array type depends on its size and an unknown type depends on its type variable. Additionally, a whole tuple/function/named/box type becomes dependent when some of its components depend on other values. The value a type depends on can be one of these:

- Another tuple item, e.g. `N: Int, array: Int...N` or `T: ?, t: T`.
- Function parameter, e.g. `(N: Int) -> Int...N` or `(T: ?) -> T`.
- An immutable variable.

There are conversions possible between these three variants, but they are not allowed in all places in which they should be. This limitation will be discussed in more detail and remedied later.

1.2.2 Immutability

For an imperative language, Kampa offers a relatively large degree of control over side effects. It does not allow global mutable state and requires each function to declare in its type what parameters it modifies and whether it modifies variables in its closure.

Any type can optionally be qualified IMMUTABLE, MUTABLE, or READABLE. If a value is qualified IMMUTABLE it cannot be modified to and it is guaranteed by the type system to never change its value (it is *referentially transparent*). A MUTABLE value can be assigned to and it may potentially evaluate to a different value every time. The READABLE qualifier does not allow modification, but it also does not guarantee referential transparency.

Any value, whether IMMUTABLE or MUTABLE can be used as READABLE. That is, both a MUTABLE type and an IMMUTABLE type are subtypes of the corresponding READABLE type. Additionally, it is allowed to change the qualifier of parts of the value that are not behind references, and thus are not shared.

The qualifier may be used in any type expression. As such, it can be used in the definition of a type (an example from the library is a string, whose characters are IMMUTABLE), or in any place the type is used (such as the mentioned function parameters).

In many other languages, if a reference type is defined as mutable, it is generally not possible to create immutable instances (or to accept a parameter that does not allow modification). For example, the C `const` qualifier, when applied to a type defined as `typedef struct ... * my_type_t;`, only applies to the pointer, but not the structure field itself. A function cannot be declared to accept an unmodifiable object of such a type.[1] In Java, the situation is even worse, since any user-defined type will be a reference type. There is no way in the type system to create a class of objects that can be either mutable or immutable. Kampa takes a different approach: the IMMUTABLE qualifier applies deeply to all components

[1]This problem can be partially solved by not using pointer typedefs, but it arises again when a pointer appears as a field of a structure.

of a type, including the contents of boxes and environments of functions (the latter being explained in more detail below). The READABLE qualifier also applies deeply, except it does not affect already-IMMUTABLE parts. Finally, MUTABLE does not affect either IMMUTABLE or READABLE parts. Its only purpose is to "protect" parts of the type from becoming READABLE by default.

Functions

Functions may also be qualified. However, the meaning of a qualifier on a function is slightly different. Functions can never be reassigned. The qualifier applies to the function's environment instead. It has implications on both the function definition (what it can and cannot do) and its use.

Let us start with the restrictions on the function definition. When a function is qualified IMMUTABLE, it means that its environment must be IMMUTABLE. It is fine if its definition has READABLE or MUTABLE variables in its scope, but it must not use them (since they cannot be typed as IMMUTABLE). An IMMUTABLE-qualified function is pure (except for MUTABLE parameters, if any). A READABLE-qualified function (the default) can refer to any and all variables in its scope. However, it must not modify them, since it only sees them as READABLE or IMMUTABLE. A MUTABLE-qualified function can use its environment without any additional restrictions. Variables that were MUTABLE are captured as MUTABLE, variables that were IMMUTABLE are captured as IMMUTABLE.

We now move on to the rules of using the functions. The subtyping rules must clearly be different from those applied to values: a conversion from MUTABLE to READABLE is fine in values (it just disallows modifications), but it is unsound for functions, because a READABLE-qualified function promises not to modify its environment, while a MUTABLE-qualified does not. Forbidding such a conversion would however mean that a value containing MUTABLE-qualified functions cannot ever be used as READABLE. This is unnecessarily restrictive. The only thing we need here is disallowing MUTABLE function calls in non-MUTABLE contexts. Technically, this is achieved by adding a fourth qualifier, DISABLED. Whenever the READABLE qualifier is applied to a MUTABLE function, the result will be qualified as DISABLED. Such functions cannot be called or converted to any function type with a different qualifier. On the other hand, there is one conversion possible that is not allowed with values: Any IMMUTABLE- or READABLE-qualified function can be converted to MUTABLE-qualified, since this only weakens the requirements on the function itself, not the caller. This is not very useful for objects containing functions, but it enables passing IMMUTABLE- or READABLE-qualified functions for example as callbacks where a MUTABLE-qualified callback is expected.

It remains to specify the rules for using other functions from inside of IMMUTABLE- or READABLE-qualified functions, although they are not very surprising. Using a MUTABLE-qualified captured function amounts to modification, and is thus only allowed in other MUTABLE-qualified functions. An IMMUTABLE- or READABLE-qualified function sees MUTABLE-qualified functions in its scope as DISABLED. Similarly, using READABLE-qualified functions from IMMUTABLE-qualified functions is not allowed, since it is equivalent to reading a value that is not IMMUTABLE.

1.3 Syntax details

The syntax is free-form, that is, mostly whitespace-insensitive. It is based on the C programming language family and its curly-braced code blocks, semicolon-terminated statements, and some expression syntax, such as `=` assignment, `.` member access, `&&` and `||` short-circuiting logical expressions, `!` negation, and `?:` conditionals. In some other aspects it deviates from most C-family languages, such as in using juxtaposition for function application (`f x`), treating blocks primarily as expressions, and treating operators as plain identifiers. In this section, we only deal with notations that are, to the best of the author's knowledge, specific to Kampa.

1.3.1 Partial application

Like in many functional programming languages, a function can only take a signle parameter/argument. The preferred way, in Kampa, to define functions with multiple arguments is using tuples, e.g. `hypot: (a: Int, b: Int) -> isqrt(a*a + b*b)` (although nothing stops the programmer from using the curried form, `hypot: (a: Int) -> (b: Int) -> isqrt(a*a + b*b)`).

It is still possible to partially apply a function (that is, to fix some items of the tuple expected by the function without calling it) using a special notation, `x f` where `f` is the function and `x` is a tuple of some of the arguments. The function used as the example above can be partially applied using `3 hypot` or `(a: 3) hypot`. The resulting partially applied function now only expects the remaining argument. To complete the call, one could write for example `(3 hypot) 4` (or `3 hypot 4` thanks to the left-associativity of calls).

In addition to providing a way to create functions without the need for a lambda expression, this also allows functions to be called like methods (for example `list add(elem)`) or like operators (`elem in list`).

1.3.2 Operator precedence[2]

Arithmetic and comparison operators such as `+`, `&`, `!=`, `~`, and so on, are plain identifiers. The standard library contains their definitions as regular functions. As shown in the previous section, they can be called using an infix notation as a consequence of the partial application syntax. This stops working as intended in more complex expressions. For example, `a == b + c*d` would parse as `((a == b) + c) * d`.

To avoid this, the language has a mechanism to define operator precedence. 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 verify **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

[2]Part of this subsection is taken over from the original thesis

Figure 1.1: Examples of macros

```
// definition:
//   - parentheses mean literal parentheses
//   - curly braces mean statement
//   - identifier without curly braces means expression

?. "if" (COND) {THEN}  =  COND ? { THEN; } : { }
?. "if" (COND) {THEN} "else" {ELSE}
                       =  COND ? { THEN; } : { ELSE; }
// use:

if(x != 0) {
    if(x < 0)
        println("Negative");
    else
        println("Probably positive");
}
```

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)`.

1.3.3 Macros

Macros provide a way to extend the syntax with custom constructs. They are based on the substitution of syntax trees (as opposed to arbitrary computation or joining token streams). A macro definition consists of a pattern and its replacement. The pattern is just a sequence of keywords, parameters, and some other tokens. The replacement is a statement containing placeholders corresponding to the parameters. Figure 1.1 shows two example definitions and their uses.

A macro cannot be recursive, but it can use other macros in its body, in which case they are expanded once during the definition. In case of nested macro invocations, the innermost is expanded first. The result of an expansion is not further processed.

2. Analysis and design

The goal of this thesis is to push the Kampa language closer towards practical usability. Practical usability depends on many aspects. Besides properties of the language itself, these include tooling, availability of libraries, and even external factors (adoption, community). Implementing tools such as IDE support with autocomplete makes for a thesis of its own. Similarly, implementing a comprehensive library is beyond the scope. The main focus of this thesis will be the core language.

When Kampa was first designed and implemented, the only code written in it was a set of eight source files, only three of which implemented some functionality (the rest were tests). The functions in the source files worked mostly in isolation, taking only numbers and simple arrays (mostly strings) as their arguments. However, for practical use, just implementing loops and arrays correctly is not enough. There are several limitations and inconveniences, some already mentioned in the original thesis, others only discovered later. They do not have much in common, except that they all limit the language's expressivity or readability, with some even forcing the programmer to use unreasonable workarounds.

This chapter presents a detailed list of these limitations and proposes potential solutions, also exploring further ways to extend the language. However, the only way to verify the suitability of the modifications is by using the language to write practical software. While it is still not possible to write application or system software due to the lack of advanced IO and standard libraries, the libraries themselves can already be considered practical if they are at least as capable as their counterparts in more widespread modern programming languages. As such, we will use the ability to implement practical libraries as a measure of practical usability.

2.1 Syntax deficiencies

This section describes some improvements that need to be done on the syntactic level. While they are not of any theoretical interest, they are important in making the language at least writeable (and ideally also readable).

2.1.1 Type instantiation

One intended property of the syntax is that types are constructed using the same syntax as their values. For example, the type of `[x: 42, y: 43]` can be written as `[x: Int, y: Int]` and the type of `succ: (x: Int) -> x+1` can be written as `succ: (x: Int) -> Int`. However, one (much more important) feature of the language is that types are values as any other (they are so-called first-class). This means that `Int` in the above examples is a regular variable that just happens to have a Type type. The above example, `[x: Int, y: Int]`, could equally well mean `[x: `*`type object`*`, y: `*`type object`*`]`. There needs to be a rule or a set of rules that dictate when `Int` refers to the type object and when it refers to an instance.

(1) Perhaps the cleanest approach would be to specify that `Int` always refers to the type object. To describe an unspecified value of the type, one would have to explicitly instantiate it (e.g. `Int()`). On the other hand, this is also the least practical approach, as it causes unnecessary clutter even in contexts where using the type object would not be sensible, such as in function parameter specification. E.g. in the `(x: Int) -> x+1` example, `x` is obviously an integer, not the integer type. There should be no reason to write `(x: Int()) -> x+1`

(2) This leads us to a second possible approach: to distinguish between contexts where type names refers to type objects and contexts where they describe values of that type. There also has to be a way to override the default.

(3) The third approach is to always interpret type names as values of the type in question, never as type objects. As in previous case, there has to be a way to explicitly request the type object.

Currently, Kampa takes the second approach: it distinguishes between type expressions and value expressions. In a type expression, a type name refers to a value of that type. In a value expression, the type name is treated as any other name (thus evaluating to the type object). Function parameters, block headers, and declarations are parsed as type expressions, while regular expression-statements and function return values are parsed as value expressions. To include a value in a type expression (e.g. in a variable declaration), the programmer may specify a *default value*. Default value is a property of a type. A type with a default value is created using `=`. For example, the expression `Int = 5` when used as a type expression means "a new type that is identical to `Int`, but has a default value of `5`." The type may be omitted and is then inferred from the right-hand side.

This requires some special cases (e.g. for function declarations) and the behavior is counterintuitive in some cases, but the most important disadvantage becomes apparent when attempting to use generic types. Consider for example a simple type constructor `List: (?) -> (?)`. `List(Int)` works just fine. As usual, the function call argument is parsed as a value expression. The type name `Int` is thus not expanded and it continues being a type object, which is a valid argument for a `?` parameter. However, for the exact same reason, `(Int, Int)` will evaluate to a tuple of two types, not the type of a tuple. As such, it will not be a valid argument for `List` and will not compile.

To create an `List` of `(Int, Int)`, one has to write `List(?(Int, Int))`. The `(?...)` construction can be seen as a counterpart of `<...>` of C++ and its descendants. For consistency, it would then be reasonable to also write `List(?Int)` instead of `List(Int)`. If all user programs followed such a convention (or if they were forced to), it would mean that type names would never appear in value expressions at all.

However, instead of forbidding value expressions with type names in them, it is possible to give them a more useful definition. As described above, the main difference between type-expressions and value-expressions is their treatment of type names. If the definition of value-expressions is updated to match type-expressions in this respect, the main reason to distinguish between the two disappears. This change means essentially switching to the last of the three approaches outlined in the beginning of this section.

To summarize this and related changes:

- As before, the result of a `?expr` is always a type object. However, the `expr` inside of it can now be any value-expression. If it indeed is a value, the value is discarded (e.g. `?42` is an equivalent of `?Int`: the type object describing the integer type).
- Type names are now always expanded. The result of the expansion is an *invalid value* of the type in question. An invalid value can be used in the same way as a regular value of the same type, but only in a context where the value is not required (such as the `?...` expression). For example, `?f(Int)` or `?f(42)` is the return type of `f` when called on an integer argument.

2.1.2 Block expressions

Block expressions were originally intended as a multi-purpose tool. In addition to grouping statements, they could also be used to construct and initialize a value. Figure 2.1 shows a simple example of this. Furthermore, they were able to extend the value and its type with additional members (thus using the value as a prototype), as shown in Figure 2.2.

However, the full power of such initialization statements was never used. Instead, most if not all of the use-cases fell into one of the following three categories:

1. Plain blocks without any header, just the output declarations in the body (like in the `makePen` function in Figure 2.2).
2. Blocks with header whose body never adds any additional members, and which use the `return` statement instead of assigning to the fields individually. For example, the `Int` in `answer: () -> Int { return 42; }` is a block header.
3. Interface implementation, as in the expression-tree example in Figure 2.1.

The highest cost is paid by code in category 2. Specifying the type in the header introduces scope variables that the body is never going to assign or use. Worse, their names can be seen nowhere in the function (only in the definition of the type) and they will often collide with names used by the function itself. For example, a function like `makeStats: (min: Int, max: Int) -> Stats { return (min, max); }` will accidentally use the uninitialized `min` and `max` from its `Stats` return value instead of the passed parameters. This scope extension could be made explicit, which would solve the problem on the part of the developer writing the code. However, unless it explicitly states which variables are imported, it does not help with reading.

If we (for a while) only consider categories 1 and 2, we could remove the scope extension altogether. This basically separates the body from the header. Now, the header is absent in 1, and serves just as an explicit type annotation in 2. We can redefine the `expr { statements...; }` syntax to mean exactly that: an explicit type annotation of the block's value (it is, after all, how everybody would read it in 2). Now it is possible to convert all code from category 3 to category 1, preserving the header as a type annotation that confirms the block body indeed implements the interface correctly.

Figure 2.1: Initializing a newly constructed value

```
// type definition
Stats: ?(min: Int, max: Int);

// creating and initializing an instance
collectStats: (N: Int, values: [Int...N]) -> Stats {
    min = INT_MAX, max = INT_MIN;

    for(i: = 0; i < N; i += 1) {
        if(values[i] < min)
            min = values[i];
        if(values[i] < max)
            max = values[i];
    }
}
```

```
// type definition
Expr: ?(
    eval: () -> Value,
    toString: () -> String,
);

// implementation
makeTimes: (left: Expr, right: Expr) -> Expr {
    eval = () -> left.eval() * right.eval();
    toString = () -> "(" + left.toString() + "*"
                         + right.toString() + ")";
}

// implementation
makeConst: (value: Value) -> Expr {
    eval = () -> value;
    toString = () -> /* ... */;
}
```

Figure 2.2: Extending a type (example taken from Lieberman [2])

```
// prototype
makePen: (initialX: Num, initialY: Num) -> {
    . [x: = initialX, y: = initialY];
    . draw: (newX: Num, newY: Num) -> {
        /* ... */;
        x = newX;
        y = newY;
    }
}

// extending the prototype
makeTurtle: (initialHeading: Num) -> (= makePen()) {
    . [heading: = initialHeading];
    . forward: (d: Num) -> {
        draw(x + d*cos(heading), y + d*sin(heading));
    }
}
```

2.1.3 Statements and expressions

As promised in the syntax summary in Section 1.1, this thesis removes the distinction between expressions and statements. This allows the developer to use fewer mutable local variables, alleviates the need for additional braces around individual commands used in expression contexts, and also makes the language more regular.

As of now, Kampa has five kinds of statements (quoting the original thesis):

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.

There are two types of goto statements: one jumps before the labeled statement (similarly to `goto` in C), the other jumps after the labeled statement (similar to `break` *label* in Java or JavaScript). The restriction on the goto statement is that any label may only be used from inside of the labeled statement itself. This means that it cannot be abused as badly as an unrestricted goto. It will never skip over definition or jump into block (or worse, into a loop, making the function not only hard to read, but also hard to work with in an optimizing compiler).

To remove the distinction between expressions and statements, it is enough to define the value to be returned by each of them, and to redefine code blocks as lists of semicolon-terminated expressions, not statements.

Goto

The most precise type for goto is the empty type (a type that has no instances), since it never yields a value. Kampa does not have a dedicated empty type, although an empty type could be expressed in various ways, such as an array of a negative size. The availability of a value of this type could then also be used by the compiler as a proof of unreachability. Detecting unreachability is necessary to avoid rejecting programs for "missing" return values and similar – it is not just for optimization. However, detecting whether a type is empty is generally a hard problem,[3] and treating just some "randomly" selected cases (such as the array) specially is less elegant than just using control-flow analysis. Therefore, until a dedicated empty type is specified and implemented, the type-checker will use the control flow.

At this point, typing goto using an empty type is not necessary and we could use a more practically useful type, which is `(T: ?) -> T`, a function that takes a type variable and returns a value of that type.[1] On the other hand, there is no significant difference between calling the `(T: ?) -> T` function with the desired type as a parameter and just "casting" an arbitrary value to an arbitrary type, which is allowed by the type checker in unreachable code.

As such, using the empty type or the mentioned generic function as the type of goto has no advantages over any other type. For now, the type of goto will be just an empty tuple, as it is easier to generate the code for it.

Labels

Label is not just one mark in the code; it wraps an entire statement. If we replace statements with expressions, label becomes an expression that wraps another expression. As said above, there are two uses for labels – two types of goto. A goto-start (repeat) does not matter much, it could be seen just as a detail that does not influence its type. On the other hand, a goto-end (break) could leave the value uninitialized, so the label expression must not blindly return the type and value of the wrapped expression.

One possible type for a label expression that contains a break is the empty tuple. This covers the usual case in imperative programs, where the wrapped expression is likely a block with no output declarations. However, it discards the advantages of using an expression in the first place: to be able to return a value.

Another option is to require a value to be embedded in the break expression. The label expression would then evaluate either to the result of the wrapped expression (if it terminated normally), or to the value in the break that terminated it early. The type would then be the union of the types of the individual breaks ("union" in the sense of "the smallest common superset"), similarly to the type of a conditional expression.

[1]Note that this type is not empty – it is inhabited, e.g., by a function that recurses forever.

A third option is to (again) require the value in the break expression, but only use the type of the wrapped expression for the label expression. The type checker would then only verify each break expression separately to have the correct type, but not attempt to generalize in case of mismatch.

The first of the three options is evidently inferior to the remaining two. However, none of the latter two seems better than the other. Taking the union over all breaks can spare the user an unnecessary type annotation, but it involves non-local effects: a single added or removed break can change the type of the whole expression, and potentially the whole function. Using just the type of the wrapped expression may restrict the type too much, especially if the end of the expression is unreachable without a break.

Currently, the interpreter uses the third approach, particularly for its ease of implementation. If it turns out that this causes it to reject too many essentially correct programs, it can always be switched.

Macros

A macro is a shortcut for a statement of another kind. Once all other statement kinds are replaced with expressions, the change is purely syntactic.

Declarations

The only[2] difference between a declaration and an expression statement is that the declaration introduces a name to the scope. For example, `answer: 42;` introduces a variable `answer` with a value of `42`. It is on itself already a valid expression with a value of `(answer: 42)`.

2.1.4 Non-mutability

Note: As explained in subSection 1.2.2, the language has three main qualifiers: MUTABLE, IMMUTABLE, and READABLE. In the following few paragraphs, we will use the term *non-mutable* to refer to both IMMUTABLE and READABLE.

The language has deep non-mutability only. When a reference (box) is non-mutable, then all data it points to are non-mutable as well. Conversely, for the data to be mutable, the reference itself must be mutable too. This was a known negative consequence, but the only solution seemed to be adding more variants of the qualifier.

Qualifying pre-existing and new types

There main reason to use deep non-mutability was to allow creating an non-mutable parameter, variable, or other value using a pre-existing type definition that itself does not use any qualifiers (and may involve references). For example, consider a search tree structure. It might have a relatively complex type involving recursion and possibly several implementation details on which regular users of the structure should not depend. There will probably be functions that need to modify the tree and functions that only look up its keys. The second group

[2]After type- and value-expressions are unified, as per subSection 2.1.1

Figure 2.3: Initializing a newly constructed value

```
// type definition without any qualifiers
Type: ?[[[Int]]];
// using the type definition, with IMMUTABLE qualifier
object: \Type;

// writing the type in-place, with IMMUTABLE qualifier
object: \[[[Int]]];
```

should only view the structure as READABLE. This means that they should have a parameter that uses the type definition, just with an additional qualifier.

Now, qualifiers in Kampa are essentially type constructors: they take a type and yield another type. There is no difference between qualifying a type taken from a type definition and qualifying an ad-hoc type created in-place (see Figure 2.3). Therefore, if we expect the qualifier to apply deeply in the first case, we also have to expect it to apply deeply in the second case.

This thesis implements a better solution. Instead of considering qualifiers as type constructors, the parser now takes them as part of the expression at hand. Only a few selected expressions can have a qualifier (`□` here stands for any qualifier):

1. identifier (e.g. `□TypeName`)
2. member access (e.g. `□foo.TypeName`)
3. call (e.g. `□GenericName(args)`)
4. named (e.g. `□bar: TypeExpr`)
5. box (e.g. `□[TypeExpr]`)

Bullets 1, 2, 3 are exactly the cases where we need deep non-mutability to avoid repeating the type definition just to change all the qualifiers in it. Therefore the effect of the qualifier in these cases remains deep.

The remaining two cases are counterparts to values that can theoretically appear on the left-hand side of an assignment. The counterpart of named expression (4) is member access. For example, the qualifier on `□bar: Int` determines whether the corresponding variable/member assignment (`bar = 42` or `object.bar = 42`) is valid or not. The counterpart of a box expression (5) is unboxing (dereference). For example, the qualifier in `object: □[Int]` determines whether dereferencing and then assigning it (`object[] = newValue`) is valid or not.

The qualifier in bullets 4 and 5 is shallow; it does not influence the contents (i.e. the type `TypeExpr`). If the programmer needs to, they can always just add another qualifier to it.

This gives us all the flexibility it realistically can. When building a new type (whether in a type definition, or ad-hoc in a function parameter), we can fine-tune every detail as appropriate. When using a pre-existing type, we have to assume it is already well-designed, and only choose from a set of reasonable options (the three qualifiers provided by the language).

The default qualifier

Another motivation for deep non-mutability was to avoid mistakes where a completely immutable object is intended, and the IMMUTABLE qualifier is used, but inadverently not at all levels. Such errors would lead to unexpected type incompatibilities (e.g. a function that cannot be called with an IMMUTABLE object although it never modifies it anyway) or false assumptions when reading other programmer's code (e.g. an immutable list of lists, giving the impression of a completely immutable structure, despite the inner lists being mutable).

However, once we replace deep immutability with the design described above, we can change the default to READABLE. This was not practical originally, since almost no explicit MUTABLE would have any effect due to being overriden by some implicit READABLE from above (unless MUTABLE was dutifully used at all levels). Without deep immutability, this is no longer a problem.

2.1.5 Summary

This section summarizes the changes by comparing the old syntax (left) with the new syntax (right).

Local variable definition

```
// immutable is backslash
\answer: = 42;

// mutable is default
answer: = 42;
```

```
// immutable is default
answer: 42;

// mutable is at-sign
@answer: 42;
```

Import

```
// named
foo: = \"foo.kampa";

// into scope
: = \"foo.kampa";
```

```
// named
foo: \"foo.kampa";

// into scope
\"foo.kampa";
```

Interface definition

```
StringBuilder: ?(
    append: String @-> (),
    build: () -> String,
);
```

```
StringBuilder: ?{
  . append: String @-> ();
  . build: () -> String;
}
```

Note: The old syntax is still possible, but using a braced code block makes it consistent with the implementation (see the next pair).

Interface implementation (constructor)

```
// should work but doesn't
sb: () -> @StringBuilder {
   // ...
   append = String @-> {
       // ...
   }
   build = () -> String {
       // ...
   }
}
// workaround (unannotated)
sb: () -> {
   // ...
   . append: String @-> {
       // ...
   }
   . build: () -> String {
       // ...
   }
}
```

```
// works
sb: () -> @StringBuilder {
   // ...
   . append: String @-> {
       // ...
   }
   . build: () -> String {
       // ...
   }
}
// works but unnecessary
sb: () -> {
   // ...
   . append: String @-> {
       // ...
   }
   . build: () -> String {
       // ...
   }
}
```

In the old syntax, the `StringBuilder` already creates uninitialized `append` and `build`. The assignments (`=`) are meant to initialize them. The implementation is however too limited to allow it. The workaround ignores the interface and just creates a type that happens to be compatible.

In the new syntax, the annotation is truly just an annotation. Whether it is omitted does not affect how members are defined (always `:`).

Macro definition

```
// distinguish expressions
// and {statements}
?. "for"({INIT}COND';'INC)
                {BODY} = {
   INIT;
   while(COND) {
      continuable BODY;
      INC;
   }
}
```

```
// expressions everywhere,
// also simplified syntax
?. "for" (INIT; COND; INC)
                    BODY = {
   INIT;
   while(COND) {
      continuable BODY;
      INC;
   }
}
```

Note

The rest of this thesis will only use the new syntax, even when referring to the old implementation.

2.2 Implementation limitations

The language is hobbled by its implementation more than by any of the above deficiencies. Here we present just a brief summary. A more detailed description can be found in Chapter 3.

As already the original thesis states, arrays on the stack (i.e. in local variables) are not supported. The problem is more general, though. Objects on the stack must have constant size. This means that values of unknown types (specified by a type variable) cannot be placed in local variables. Likewise, it is not possible to implement a function that takes a parameter of unknown type, although it can take a concrete type and then be called in a context where the type is unknown (e.g. a comparator parameter – it is implemented for integer and then passed to a generic sorting function).

Then there is a whole class of limitations concerning dependent type checks. The element count of an array must always be an immutable variable (either a local variable or, in type definitions, a neighboring tuple member or function parameter). It is not even possible to use numeric literal expressions for array sizes, everything must have a location. The reason for this is that the type checker cannot verify value equivalence, just memory locations. It considers two values equal if and only if they have the same location. In value expressions, the location is just the position in the stack frame. In type expressions, it can additionally be another tuple member or a function parameter within the same expression (referred to as sibling type dependency in the implementation). Even allowing it and accepting the fact that it will not be compatible with anything (possibly even itself) would be unsafe. This is because the array element count must be remembered for later bound checks and size and offset calculations. If this restriction were not in place, a size calculation could accidentally re-evaluate a call, which could be an expensive, or even side effectful, operation.

Finally, while the implementation supports generic functions, it does not allow generic types. In Kampa, a generic type is written as standard function that returns a Type. However, when attempting to use the function's parameter in the returned type (something one would want to do in practically all cases), it is reported as an error: "Dependent type not allowed here". This is a known and intentional restriction, dictated by the internal representation of types.

Chapter 3 remedies all these limitations. Here, in the following sections, we design features for the improved implementation.

2.3 Missing language features

This section analyzes ways to extend the language. The language can be used without most of them, but not as efficiently.

Overloading and parameter inference refer to individual features, but the rest are more like use-cases. We will analyze what prevents these use-cases and what needs to be added in order to support them.

2.3.1 Overloading

Because member functions are only used for polymorphism, most functions are non-members. Even though this means there will be a large number of functions in the scope (from imports), this should not pose a problem. There is no significant difference between selecting functions based on the type they are member of, versus based on the first parameter type. Of course, this assumes functions can be overloaded.

The elements eligible for overloading must be expressions, not just identifiers. There are two reasons for this. First, the left-hand side of a call (i.e. the function) is generally an expression, not necessarily just an identifier. The scope lookup is done by the identifier expression itself, not by the call expression. Treating calls that happen to have an identifier (or even a member access?) on the left-hand side specially would lead to unnecessary irregularities in the syntax. It would therefore simplify neither the specification nor the implementation. The second reason to overload on expressions is that there are several other expression types we might want to overload:

Identifiers and **member accesses** (already mentioned and included just for completeness).

Partial applications. In `a + b` (i.e. `(a +) b`), there might be several alternatives of `+`, some of which might only differ in the second parameter. For example, we want to allow `timeDelta + timeInstant` and `timeDelta + timeDelta` to call different `+`.

Tuples. Tuple expressions are often used as arguments to functions. Some component of a tuple could be (in the simple case) an identifier referring to an overloaded function. The correct overload can only be chosen after the tuple is passed to the higher-order function.

Functions. Overloading could be used as a way to infer parameter types in lambda expressions. For example, the lambda in `list map((a: _) -> a.x)` could have its type inferred from the type of the second parameter of `map` (the `_` in the example is not actual Kampa syntax).

This does not mean all expressions should be overloadable: at the very least, values and their types must be unambiguous when entering the scope, since they can potentially be used multiple times in different contexts and these contexts must not influence each other. Then there are expression types where the distinction is not as clear. For example, allowing whole call expressions to be overloaded, that is, deciding what function to call according to the context, could allow overloading curried functions, but it could also make the code hard to understand. For some other expressions, there might not even be any reason to allow overloading (except that allowing it is as easy as forbidding it). This includes block expressions, assignments, member accesses.

2.3.2 Parameter inference

Calling functions with generic/dependent parameter types is currently unnecessarily verbose and redundant. For example, to apply a function defined as `qsort: (T: ?, N: Int, array: @[T...N], less: (T, T) -> Bool) -> ()` to the array `foo: @[Int...42]`, one has to write `qsort(?Int, 42, foo, <)`. Passing anything other than `?Int` and `42` for the first two parameters results in a type error. This check generally does not even have a significant potential to catch errors. It is just an excuse to "examinate" the programmer.

However, parameter lists are just tuples like any other and type/length parameters are just tuple items like any other. They cannot be just made optional; it would lead to ambiguities and force the type checker to backtrack too often. Requiring that they should always be omitted would in turn cause "spooky action at distance", where adding items to the end of a tuple (adding parameters to a function) could cause items in the beginning to become inferable, and thus required to be omitted. Additionally, it would disable forwarding whole parameter tuples, since they would always have to be stripped of inferable items. As such, the inference must be explicit.

One possible solution would be to make it explicit at call-site, adding a special expression, say `_`, that would mean a value that can be inferred from the context. Writing it is quite bearable in the above example: `qsort(_, _, foo, <)`, but it may be confusing for readers. Specifically, it tells them nothing about the function type anyway and only leaves them wondering why are there exactly two inferred parameters and what are they. To add to this, it becomes especially inconvenient in generic abstract data types. Consider for example a method of a generic list: `get: (T: ?, this: List(?T), index: Int) -> T`. To call it using the infix notation, one would have to write `(_, myList) get(5)`, compared to `myList get(5)` on a non-generic list.

For the above reasons, inference is made opt-in on the definition site. To make a parameter inferable, it has to be marked as such. The dot character (`.`) was chosen as the marker, as it is easy to type and does not visually clutter the definition. For example, after the `qsort` parameter is updated to `(.T: ?, .N: Int, array: @[T...N], less: (T, T) -> Bool)`, the function can be called using `qsort(foo, <)`.

There may still be reasons to call the function with explicit parameters, for example to force a conversion. As such, there needs to be a way for the callers to decide to override the `.`. Here, we use the fact that only named parameters can be marked as inferable (as there is no way to use unnamed parameters as type dependencies). To explicitly pass an otherwise inferable parameter, the call site can use the parameter name. For example: `qsort(T: ?Int, N: 42, foo, <)`. This also allows unambiguously specifying some parameters and leaving others for inference.

Implicit parameters

Kampa does not support type constraints and no such support is currently planned. When a function takes a type variable parameter such as `T: ?` in the `qsort` and `get` examples above, the parameter can represent any type of an invariant size. Nothing else is known about it. This means that, given an object of such a type, it is not possible to use it in any other way than copying it into memory locations of the same type and passing it as function parameters of the same type.

As a consequence, if a generic function needs to do anything else, such as accessing the object's members, using its methods, or doing anything else specific to that type, it needs to accept another parameter (usually a function) that performs this action for it.

An example of this is the `less` parameter in the `qsort` example above. `qsort` cannot directly compare instances of `T`, so it needs to accept a parameter that does the comparison.

Many languages support various forms of type constraints. In Java and C#, type parameters can be constrained to implement an interface. In Haskell, a similar function is served by type classes. In Rust, it is traits. In the following text, we will use the term trait to refer to all of these.

One property all these have in common is that there can be only one trait implementation per a (type, trait) pair. However, for many such pairs there are more possible implementations. Therefore there also needs to be a way to use a different implementation for an existing type. To continue the `qsort` example: there may be multiple possible orderings on one type, and thus multiple possible `less`s. In none of the mentioned languages can traits solve this. As such, the standard `sort` function always comes in at least two versions:

- Java has `Arrays.sort(Object[])` which requires all elements to implement the `Comparable` interface (checked at runtime on individual elements), and it has `Arrays.sort(T[], Comparator<? super T>)`.
- C# has `Array.Sort(Array)`, again checking at runtime all elements implement the `IComparable` interface, and `Array.Sort(Array, IComparer)`.
- Haskell has `sort :: Ord a => [a] -> [a]` (using the type class `Ord`) and `sortBy :: (a -> a -> Ordering) -> [a] -> [a]` (using a comparison function `a -> a -> Ordering` instead).
- Rust has `sort(&mut [T]) where T : Ord` and `sort_by(&mut [T], F) where F: FnMut(&T, &T) -> Ordering`.

Similarly, all other functions requiring an ordering must come in two variants. Java and C# will sometimes accept a `null` comparator in which case they cast the elements to the comparable interface.

As said above, in Kampa, there is no way to constrain a type variable to any equivalent of `Comparable` or `Ord`. Instead, the equivalent of `Comparator` or `a -> a -> Ordering` (i.e. the `less` function) is the only option. To accomodate for the cases where there is indeed a sensible default ordering and it is the one the caller of the library needs, the language should support selecting this default implicitly. This can be done by looking up a matching comparator in the caller's scope. As with inferred parameters, there must be a way to distinguish parameters that are always implicit from ones that are never implicit. The same notation as for inferable parameters (`.`) can be used, since it does not make sense to make parameters that can be inferable implicit instead. For a value in scope to be passed as an argument to an implicit parameter, its name must match and its type must be convertible to the parameter type.

A similar mechanism is found in the Scala programming language, where it can be considered complex, dangerous and unreadable when not used judiciously[4].

Figure 2.4: Encapsulated list using closures

```
makeList: (size: Int) -> {
    // a local variable captured by the functions below
    elems: @["N/A"...size];
    // three independent functions, returned as a tuple
    . getSize: () -> size;
    . get: (i: Int) -> String { return elems[i]; }
    . set: (i: Int, s: String) @-> { elems[i] = s; }
}
```

The most severe problems seem to be caused either by interactions between implicit parameters and implicit conversions, or by implicit conversions alone. These are not applicable to Kampa, since Kampa does not have any user-defined implicit conversions. Another subtlety of Scala implicits is shadowing, where even an incompatible non-implicit value can shadow another value (of a possibly compatible type). Probably because of this, the authors of the cited article suggest not to use the name in the resolution at all. However, the name can be used to significantly limit the set of candidates for the implicit argument. Kampa could solve the problem of shadowing by using overload resolution (Section 2.3.1) instead, thus never allowing an unrelated value to hide a valid candidate.

2.3.3 Encapsulation

Originally, the principal means of encapsulation in Kampa was closures. For example, a simple fixed-size string list could be implemented as shown in Figure 2.4. The type of such a list is a tuple of three functions, exposing no implementation details. The source code of the implementation can be modified at any time without breaking any users. It is even possible to choose between multiple implementations at runtime by using different constructors.

However, there are also some disadvantages:

- Even if we do not plan to use the advantage of the dynamic polymorphism, the virtual calls will still be there.
- Each closure has its own copy of `size` and two of them have their own copies of the `elems` pointer, which is relatively wasteful.
- There is no way to express the relations between the objects' implementations, or even that two objects share the same implementation. This is not important in the list example, but it becomes an obstacle in some others, see Figure 2.5.

A much more flexible solution is based on existential types[5]. An existential type in Kampa is a tuple with a type variable as one of its items and with other items using that type variable. Figure 2.6 shows the list example translated to use existential types (note that this is not a literal translation, but a useful one). Now we can use `ListsImpl.makeList(...)` to create a list and `ListsImpl.List` to

Figure 2.5: Filesystem interface using closures

```
FileSystem: ?{
    . getFile: (path: String) @-> File;
    . move: (from: File, toDir: File) @-> ();
    // ...
}

File: ?{
    . isSameFile: (other: File) @-> Bool;
    // ...
}
```

refer to the type. The type `ListsImpl.List` is a type variable. One cannot use it directly. Therefore, changing its implementation is again backwards compatible. Let us now compare this to the the closures approach:

- Each user can choose to refer to the `ListsImpl` constant directly or to take a parameter of type `Lists`. The former case can be easily compiled without any virtual calls, while the latter retains the possibility of dynamic polymorphism.
- An instance of type `ListsImpl.List` takes the same space as the plain tuple used as its definition. For the polymorphic case, passing the `Lists` parameter with it takes additional 6 pointers. (Ideally, these could be in a shared immutable box, but this is not currently supported.)
- The relations between objects' implementations can be expressed easily. See Figure 2.7. `File` types of different `FileSystem` implementations will be considered incompatible.

Limitations

One minor limitation was that the `Lists` interface in Figure 2.6 could not be written as a block, it was necessary to use the tuple syntax. This was fixed in Section 2.1.1 by unifying type- and value-expressions.

Worse, the conversion on the very first line of the implementation, which converts the whole block to `Lists`, is not allowed. This is because the type checker does not support existential generalization, i.e. replacing some/all occurrences of the concrete type `(size: Int, elems: @[String...size])` with the abstract type `List`. A workaround is possible by using its inverse, universal instantiation. One first has to define an identity function on the type `Lists` (the interface type), i.e. `Lists -> Lists`. It is then possible to pass the whole block with the implementation to the function. The type checker instantiates (specializes) the function for the concrete `List` type, `(size: Int, elems: @[String...size])`, and will thus accept the rest of the implementation as its parameter. In the body of the function, the whole parameter already has the type `Lists` and can

Figure 2.6: Encapsulated list using existentials

```
// the interface as an existential type:
Lists: ?{
    . List: ?;
    . makeList: (size: Int) -> @List;
    . getSize: (this: List) -> Int;
    . get: (this: List, i: Int) -> String;
    . set: (this: @List, i: Int, s: String) -> ();
}

// a concrete implementation:
ListsImpl: Lists {
    . List: ?(size: Int, elems: @[String...size]);
    . makeList: (size: Int) -> @List {
        return (size, @["N/A"...size]);
    }
    . getSize: (this: List) -> this.size;
    . get: (this: List, i: Int) -> this.elems[i];
    . set: (this: @List, i: Int, s: String) -> {
        this.elems[i] = s;
    }
}
```

Figure 2.7: Filesystem interface using existentials

```
FileSystem: ?{
    . File: ?;
    . getFile: (path: String) @-> File;
    . move: (from: File, toDir: File) @-> ();
    // ...
    . isSameFile: (this: File, other: File) @-> Bool;
    // ...
}
```

be just returned unchanged. Obviously, the above workaround is unnecessarily complicated and leads to large amounts of boilerplate code. This thesis adds existential generalization as an implicit conversion, meaning the cast in the example is allowed.

The need for the cast

There remains one unresolved problem. Without the cast triggering the existential generalization, the `ListImpl.List` type definition is essentially just an alias for the concrete type, and all the methods operate on the concrete type. In many cases, this does not matter since the cast *will* be in place anyway to ensure that the implementation satisfies the interface. However, in some cases, the programmer does not need a separate interface specification. For these cases, there should be a way to omit the cast and still end up with an existential type. However, there is currently no way to do this.

To understand why the current design is wrong, let us compare the following two “flavors” of existential types:

```
Array: ?{                          Object: ?{
  . N: Int;                          . T: ?;
  . new: () -> Int...N;              . new: () -> T;
}                                  }
```

These two types are very similar:

- Each of them is a tuple of two items.
- In both the type of the second item depends on the value of the first item.
- We do not see the actual value of the first item in either of the *types*.

An “implementation” of the above types could look like this:

```
ArrayImpl: {                 ObjectImpl: {
  . N: 3;                      . T: ?(Int,Int,Int);
  . new: () -> Int...N {       . new: () -> T {
      return 0 ... 3;              return 0, 0, 0;
  }                            }
}                            }
```

Note that the type of `N` is still just `Int`. The implementation does not leak its detail in its type. On the other hand, the type of `T` will be `?(Int, Int, Int)` (i.e. a Type type) instead of `?` (the Type Variable type). We can fix this inconsistency in the language. And fixing it does not even break the example. The `ObjectImpl` code would still be compilable even if the expression had the type `?` instead. The type checker could still see its definition locally, in the same way it can see that `N` = `3` locally.

But before we do away with Type types entirely, let us consider one more example. We will add parameters to the `N` and `T`. In the original `Lists` example, this would be necessary to make the lists generic. In this example, there is no real need for any type parameters, so we will just use empty parameter lists. The interfaces now look like this:

```
Array: ?{                          Object: ?{
  . N: () -> Int;                    . T: () -> ?;
  . new: () -> Int...N();            . new: () -> T();
}                                  }
```

And these are the corresponding implementations:

```
ArrayImpl: {
  . N: () -> 3;
  . new: () -> Int...N() {
      return 0 ... 3;
  }
}                ObjectImpl: ?{
                   . T: () -> ?(Int, Int, Int);
                   . new: () -> T() {
                       return 0, 0, 0;
                   }
                 }
```

`ArrayImpl` now does not compile, since `N()` cannot be proven equivalent to `3` (the function conceals this fact). On the other hand, `ObjectImpl` works, since the type of `T` is `() -> ?(Int, Int, Int)` (not just `() -> ?`), and thus the type of `T()` is `?(Int, Int, Int)`. There is no need to prove anything about values if all we need is already in the type. If we replaced Type types with the Type Variable type, neither would work.

This just confirms that there indeed is an inconsistency that needs to be rectified, in fact, two of them:

- The value of a parameterized definition (done using a function, `() -> ...`) is opaque to the type checker, while a definition without a parameter list (i.e. without a function) is transparent.

- As said above, integral constants are never part of the type, while type constants may or may not be part of the type (and they are by default). The reason is mostly historical. This can be fixed by just removing Type types. However, the previous must be fixed first.

The former was done intentionally, on the premise that functions are *the* units of code that should be opaque to the outside code. As indicated by the parameterized type definition, this is not necessarily true. And conversely, the `ObjectImpl` and `ListsImpl` definitions from the above examples should be opaque to the outside, but they are not. There are ways to cast the abstract data type away and access the internals, provided we do this from inside of the same function (and the same file).

We should consider other units (not functions) to take over the role of "opacifier". There are several options:

- Block expressions, including the top level (the file code block).

  Advantage: implicitly encapsulates every nontrivial piece of code. Disadvantage: it would already be the third feature specific to blocks (in addition to composing a tuple *and* type-annotating it).

- Files only.

  Advantage: whole files are always encapsulated, no restrictions within files. Disadvantage: any element of a file is open to the whole file.

- A new expression type.

  Advantage: does one thing and does it well. Disadvantage: significant chance of being underused (omitted out of laziness), no easy way to be put on a file block.

- The previous two approaches combined.

Except with the third option, there also needs to be some way to suppress the opacification. This is necessary to continue supporting type aliases, and it will also allow transparent constants.

Testing these approaches in practical applications will require further work. While the collection library (discussed later, in Section 4.1.2) is a good test case, it will also be useful to consider APIs with more interrelated types, such as some abstract filesystem interface.

2.3.4 Metaprogramming

Kampa already provides two ways to generate programs, but no way to inspect existing programs.

One form of program generation in Kampa is macros. This should probably *not* be extended to allow the inspection of macro parameters, since it would make the parsing of the language hard, especially when it needs to be done incrementally and continually, as in an IDE or language server. Additionally, during the macro expansion phase, the compiler also does not yet know the types, which means it cannot type-check macro definitions until expanded and the expansion cannot depend on the types of the arguments.

The other way to generate programs is doing so within the language itself. Like most general-purpose languages used today, Kampa allows regular programs to build complex functions out of simpler functions.[3] What should be noted here is that a program can take any value on its input, perform an arbitrary type-safe computation, and produce an arbitrary program on its output, as a function with a requested type. However, some metaprogramming techniques (including the macros mentioned above) can offer more: the functions (or generally any values)

[3]Whether this on itself constitutes metaprogramming is beyond the scope of this discussion.

generated by them are not restricted to predefined types. The metaprogram is able to generate a new type for the value it returns and this new type can be seen by the type checker.

While it would be possible to hand-write the type and then just generate the function(s) for it, there are many situations where the type itself is the most important part of the generated code. For example, automatically generated interfaces for libraries written in other languages[6] or according to interface description languages or even external data sources[7]. All this can be done in a type-safe manner in the language itself if it is powerful enough[8].

Generating types from values

As already mentioned in Section 2.2, it was originally not possible to write a function that returns a type that uses the function's parameters. This makes even the most trivial form of type generation (substitution of parameters) impossible.

This limitation is removed by this thesis, but others remain. Most importantly, a function that returns a type cannot branch (or, more precisely, if it branches, then the returned type becomes completely opaque) or recurse. On the other hand, parts of the type can be selected as elements from an array.

Inspecting types

The changes done in Section 2.3.2 (parameter inference) already enable trivial cases of inspecting types:

```
ParamType:  (.T: ?, .R: ?, ?(T -> R)) -> ?T;
ReturnType: (.T: ?, .R: ?, ?(T -> R)) -> ?R;
```

Both functions have only one non-inferable parameter, the function type. The first two parameters can be inferred from it. For example, if we passed the type of `+` to `ParamType` (i.e. `ParamType(?(+))`) we would get `(Int, Int)`. We could also write a function that returns both types in a tuple:

```
getInfo: (.T: ?, .R: ?, ?(T -> R)) ->
                (paramType: ?T, returnType: ?R);
```

Now, let us generalize `getInfo` to work on any type, not just functions. It has to return a type that can accomodate multiple options, because the current `(paramType: ?, returnType: ?)` does not make sense for non-functions. We will call this new type `TypeInfo`. Its definition is not important for the example.

For simplicity, we will only add support for `Int`. We will use overloading, as discussed in Section 2.3.1.

```
// a reflective representation for types
TypeInfo: ?/*not relevant for the example*/;

// constructors for the TypeInfo type
makeIntInfo: () -> TypeInfo;
makeFuncInfo: (T: ?, R: ?) -> TypeInfo;

// overloaded function that converts
// real types to TypeInfo instances
getInfo: (?Int) -> makeIntInfo();
getInfo: (.T: ?, .R: ?, ?(T -> R)) ->
                        makeFuncInfo(?T, ?R);
```

Like this, the `TypeInfo` for a function would refer to the real types of the parameter and return value. What if we wanted to use `TypeInfo` for them instead?

```
makeIntInfo: () -> TypeInfo;
makeFuncInfo: (T: TypeInfo, R: TypeInfo) ->
                                     TypeInfo;
getInfo: (?Int) -> makeIntInfo();
getInfo: (.T: ?, .R: ?, ?(T -> R)) ->
      makeFuncInfo(getInfo(?T), getInfo(?R));
```

This example would not compile. Neither of the `getInfo` calls on the last line is valid. This is because overloads are only resolved once, when the function that contains them is defined; not for each instantiation separately as in, e.g., C++. Nothing is known about `T` and `R` at that time.

This is similar to the reason we proposed implicit parameters in Section 2.3.2. While the current function does not know of any applicable `getInfo` for either `?T` or `?R`, the caller does in some cases. Let us make use of this fact and request these from the caller using implicit parameters.

```
makeIntInfo: () -> TypeInfo;
makeFuncInfo: (T: TypeInfo, R: TypeInfo) ->
                                     TypeInfo;
getInfo: (?Int) -> makeIntInfo();
getInfo: (.T: ?, .R: ?, ?(T -> R),
          .getInfo: (?T) -> TypeInfo,
          .getInfo: (?R) -> TypeInfo) ->
      makeFuncInfo(getInfo(?T), getInfo(?R));
```

Now, there is no overload resolution even taking place inside `getInfo`. More precisely, there is, but it is trivial. `getInfo(?T)` resolves to call the exactly matching (and the only matching) parameter `.getInfo: (?T) -> TypeInfo`. Analogically with `?R`. Figure 2.8 shows how calling such a function could work.

Still, there are several unknowns.

- How to write `getItem` for named types, e.g. `foo: Int`? Currently, parameter inference can be used to unwrap these, but only if the name (`foo`) is known in advance.

- How to support dependent types? For example, a dependent function does not match `?(T -> R)` for any `T: ?, R: ?`. Instead, it is, for some `T: ?, R: (?) -> ?`, `?(T -> R(T))`. It is almost certain that `R` cannot be inferred in this case.

- Can we somehow ensure that the expansion of the implicit parameter stops even for recursive types? That is, can the type convertor always expand to a recursive function instead of infinitely recursing itself?

- Should this be even done? There are several advantages, such as supporting abstract data types well (`getInfo` can be written for each), not enforcing any particular `TypeInfo` structure and even allowing some functions (e.g. hashing) to not use `getInfo` and work with types directly. On the other hand, it requires quite complex and powerful type conversions (with the open questions above).

Inspecting expressions

The "pattern matching" used in the previous section is based on the property of Kampa that (informally speaking) types are transparent by default, while values are always opaque. A more precise statement can be found on page 28 of the section on encapsulation. After this inconsistency is fixed, values can be pattern-matched in the same way as types.

For now, it is possible to pattern-match values using a hack similar to C++'s `std::integral_constant` [9, meta.help]. A value can be embedded into a type by making it a dependency, as shown in Figure 2.9.

Similarly to types, without implicit parameters, there is no way to inspect more than a constant number of hard-coded patterns.

2.3.5 Coroutines

A *coroutine* is a function which can pause its execution to be resumed at a later time. What this "later time" means depends on the coroutine. Usually the coroutine returns an object to its caller, which then electively does the resumption by calling a method (e.g. `resume` [9] or `next` [10] or `send` [10]) on the object. If coroutines are used to implement asynchronous function, the coroutine itself schedules the resumption to a specific event,[11] usually the completion of a *future* (e.g. the completion of a *task*, or a *promise*).

Figure 2.8: Type inspection using inference and implicit parameters

```
// example type definition
testCase: ?(Int -> (Int -> Int));

result: getInfo(?testCase);
// resolves to:

// finding the matching overload
// and inferring parameters
getInfo(T: ?Int, R: ?(Int -> Int), ?testCase,
        getInfo: (?T) -> getInfo(?T),
        getInfo: (?R) -> getInfo(?R));

// expanding the definitions of T and R
getInfo(T: ?Int, R: ?(Int -> Int), ?testCase,
        getInfo: (?Int) -> getInfo(?Int),
        getInfo: (?(Int -> Int)) ->
              getInfo(?(Int -> Int))
       );

// finding the matching overload
// in the last parameter
getInfo(T: ?Int, R: ?(Int -> Int), ?testCase,
        getInfo: (?Int) -> getInfo(?Int),
        getInfo: (?(Int -> Int)) ->
              getInfo(T: ?Int, R: ?Int, ?(Int -> Int),
                      getInfo: (?T) -> getInfo(?T),
                      getInfo: (?R) -> getInfo(?R),
                     )
       );

// expanding the inner definitions of T and R
// in the last parameter
getInfo(T: ?Int, R: ?(Int -> Int), ?testCase,
        getInfo: (?Int) -> getInfo(?Int),
        getInfo: (?(Int -> Int)) ->
              getInfo(T: ?Int, R: ?Int, ?(Int -> Int),
                      getInfo: (?Int) -> getInfo(?Int),
                      getInfo: (?Int) -> getInfo(?Int),
                     )
       );

// done: all getInfo calls now refer to either
// getInfo(?Int) or getInfo(T: ?, R: ?, ?(T -> R), ...)
```

Figure 2.9: Embedding a value in a type to make it transparent

```
Constant: (.T: ?, t: T) -> (?) {
    return ?(); // the actual type does not matter
};

// just for illustration - this could be more generic
unapply: (.F: Int -> Int, .X: Int, ?Constant(F(X))) ->
                                                   (f: F, x: X);
// example use
succ: 1+;
alsoSucc: unapply(?Constant(succ 7)).f;
seven:    unapply(?Constant(succ 7)).x;
```

The creation of a generator object, as well as the scheduling of the resumption (usually written as `await` and implemented by calling `then`) is usually done by the language itself, with no possibility of customization or extension.

On the other hand, Kampa aims to only provide the core mechanisms and entrusts libraries with providing convenient and high-level abstractions. In this case, there is no need to implement the `await` keyword as a call to `then`. This can be done by a library `await` macro. All the macro needs from the language is a way to stop and resume its enclosing function. The former is trivial: to stop a function execution, just return from it. The latter can be achieved with a *continuation*: a function object that contains the current execution state of the function (local variables, instruction pointer).

As a first approximation, the `await` macro would perform the following three steps (where **return** terminates the current function, not the macro):

```
Macro await (future):
    1. callback = current state
    2. future2 = future.then(callback)
    3. return future2
    4. ← future.value
```

The obvious problem here is that once *future* completes and calls *callback*, *callback* resumes at line 1. What we can do about this is to save a different instruction pointer:

```
Macro await (future):
    1. callback = current state but line 4
    2. future2 = future.then(callback)
    3. return future2
    4. ← future.value
```

We can now look into what the **current state** expression even is. It must be something that captures local variables. It should also be able to take arguments, so that we do not need to use (or even have) the value field in Future objects. Kampa already does have a primitive that can do both: plain old lambda functions.

Macro await (*future*):

```
1. $callback = \lambda\ value \rightarrow$ goto line 4
2. $future2 = future$.then($callback$)
3. return $future2$
4. $\leftarrow value$
```

Normally, a goto statement would not be allowed to use a label from a different function. On the other hand, if the type checker verifies that the inner function and the outer function have the same return type, there is no reason it should not be allowed – from the technical point of view, that is. The implementation just has to ensure that variables appearing in the continuation are captured as if they were in the lambda body itself.

Let us now rewrite the code into Kampa syntax. This also involves replacing unrestricted goto with a break statement. First, unrestricted goto does not exist in the language, second, a break will allow us to pass the *value*, which would normally be out of scope once the lambda is broken out of.

```
? "await" future = $breakLabel: TValue {
    callback: (value: TValue) -> TFuture2 {
        $breakLabel = value;
    }
    future2: future.then(callback);
    return future2;
}
```

The type `TValue` can be easily obtained from the type of the `future` expression. The type `TFuture2`, used for the return type of the lambda, is also the return type of the current function. Currently, it cannot be obtained in a general way. However, `await` will always appear in `async` blocks. `async` can be defined in a way that makes `TFuture2` available (possibly with a better name).

Notes

`async` was used here just as an example use of the general language feature and has been significantly simplified. We will discuss the design of the actual asynchronous programming library in Section 4.1.4. Another application of continuations, generators, is analyzed in Section 4.1.3 on iterators.

Also note that breaking out of a function only changes the instruction pointer. It does not manipulate the stack in any way. The example in Figure 2.10 will not work as intended (i.e. it will not stop the `map` function):

If the user is lucky, the program will not compile because the outer function does not return `ProcessedItem` as expected by the type annotation on the inner function. If the outer function happens to also return `ProcessedItem`, then the

Figure 2.10: Creating a continuation by accident

```
$breakLabel: {
    processedList: list
          map((item: Item) -> ProcessedItem {
              if(item.evil())
                  $breakLabel =;
              return process(item);
          });
    // ...
}
```

continuation will be executed for each `evil` item in the list and the `map` will always process the whole list.

To avoid mistakes like this, functions are not allowed to use nonlocal labels by default. This needs to be enabled by replacing the `->` arrow with `~>`. Internally, functions defined with `~>` are called malfunctions (because they usually break).

2.4 Missing runtime features

We mention these two points separately from the language features because they are not related to the language (its semantic or grammar) per se. Instead they are requirements that the runtime must implement.

2.4.1 Bytecode

Although currently the only implementation of Kampa is an interpreter (or a JIT compiler when used in conjunction with GraalVM[12]), the language is designed mainly to be compiled. This includes separate compilation, where each library is compiled on its own to be included in an application later.

Separate compilation should not impede type safety or optimization, and portability of compiled libraries is an advantage, so the libraries should be compiled to some form of intermediate code (bytecode) rather than machine code directly. Additionally, it can serve as a clear interface between the front-end (which analyzes the source and deals with all the high-level abstractions), possibly a general optimizer, and platform-specific back-ends.

The requirements on the bytecode are:

- The ability to represent any Kampa type unambiguously. This is necessary to allow compiling against bytecode files without the need of separate header files.

- The ability to represent any value, including cyclic references and values of dependent and existential types. The top level code block in a Kampa source file executes at compile time and may construct an arbitrary object, which needs to be preserved by the bytecode.

- Code representation that can be further manipulated, e.g. inlined or (in case of generic functions) specialized. This allows link-time optimization.

- It should avoid redundancies and ambiguities present (or planned) in the language, such as name lookups, implicit conversions, overloading, parameter inference, and type inference. This makes the analysis of the bytecode easier and more efficient, which is the main reason to use it as the input for an optimizer/backend instead of the source language.

On the other hand, being directly interpretable (e.g. as a stream of instructions that can be executed without any context) is not a requirement. Most virtual machines for existing bytecodes already do some preprocessing or verification anyway.

The need to represent objects on itself already rules out many existing formats, but the most restricting requirement is, of course, the type system. This is not a compatibility issue, since a translation step may always be introduced, which will use some lossy encoding (such as dropping the types and replacing member/element accesses with explicit offset calculations).

The bytecode format

The format consists of two layers. The upper layer is called IR (intermediate representation) and it represents all executable code, types, and constant data objects as a single object graph. The lower layer is called serialization (unrelated to Java serialization) and it encodes the graph in a stream of bytes.

The serialization layer supports three data types:

Long is an unsigned integer in the range $[0, 2^{64})$. It is encoded as a variable-length quantity similar to UTF-8.

Variant tag is an enum – unsigned integer in the range $[0, n)$ where n is the number of alternatives. It is encoded using $\lceil \log_{256} n \rceil$ bytes in network byte order.

Reference is a composite object that can be serialized once and referred to multiple times. The first occurrence is encoded as a zero byte followed by the object's fields, each encoded according to its type (which is again one of the three in this list). Any later occurrence of the same object is encoded as a back-reference, which is a Long followed by no fields.

The IR layer represents function code as a mixed control-flow and data-flow graph, drawing inspiration from the Sea of Nodes representation.[13, 14] However, there are two important differences. First, it lacks a representation of mutable local variables. In the Sea of Nodes, these are represented using static single-assignment form. We avoid the need for it by treating these variables as memory. This is a deficiency to be addressed by future work. Second, the control-flow edges are directed away from the entry node, while in Sea of Nodes, they are directed towards the entry. This divergence makes the back-end and especially the front-end more straightforward.

Types are represented by the IR layer using a type graph. Each node corresponds to a type (e.g. an array type, a function type, the number type) and may refer to other types (e.g. an array type refers to its element type, a function type refers to its parameter type and its return type) and even data-flow nodes (e.g. an array type refers to a node that computes its element count).

Finally, constant data objects are represented by the IR layer just by themselves. That is, numbers are serialized as the two's complement longs, boxes are serialized as references, tuples/arrays are represented as concatenations of their items/elements. The representation of a named type is the same as that of the type wrapped in it. A function object is represented as a tuple of the function's code (as described above) and its data (usually a tuple of the variables in its closure). A Type Variable value is represented as a type graph node, as described above, while Type values have an empty representation.

Neither of the layers includes any field names or tags or other metadata. There is generally no way to deserialize an object without knowing its type in advance. This is not a problem in the function code and types, where both the encoder and the decoder know the schema. However, with constant data objects, the type of the object must come first. That is, types act as the schema for data objects.

It is also possible to describe the schemata for function code and types themselves in terms of types, which means that types and function data could be used as constant data objects. This is currently not allowed by the implementation, since it needs to use a different runtime representation for each of the three categories.

Notionally, a bytecode file just holds a single constant data object. This will usually be a tuple of functions and other definitions for non-executable, library, files. Executable bytecode files are required to hold a single function value. Physically, a bytecode file consists of three parts: the magic number (`29 2f 28 0a`), the type of the object, and the object itself.

Usage

The bytecode files can be used exactly in the same way as source files, i.e. imported by other source files (using the same syntax) or executed (using the same command). Additionally, there is a command that can only execute bytecode but not the source files (i.e. it is just the runtime without the compiler). The command-line interface is described in Section 3.1.

2.4.2 Null

Many data structures contain space that is allocated but not yet populated. Perhaps the simplest example is an array list (a growable array), whose capacity (physical size) is generally larger than the notional size.

In memory-unsafe languages, this space would just be left uninitialized. The implementation of the data structure must then ensure it never reads the uninitialized data. Kampa is memory-safe, and as such, it must not allow a program to ever use uninitialized memory.

The array list implementation still ensures it never uses an uninitialized value (by checking the size field first), it just cannot prove it. Using the dependent types provided by the type system is unfortunately not an option, since the size field (which would have to be used as a dependency) is mutable.

Therefore, we have to make sure the elements are always initialized with some default value. Obtaining this default value cannot easily be done in Kampa. There is no generic way to create a default value for an unknown type. There even cannot be, since there is no feasible default value for boxes (references). Kampa does not have null pointers, and allocating an actual pointer just to fill a field that will never be read is a nonsense.

The array elements thus have to be of a different type, `Nullable(?T)`. This type can represent any value of the original type `T` and has at least one value which is easy to generate (`null`). Additionally, it must satisfy the following requirements:

Round-trip conversions. It is fairly obvious that converting a `T` value to `Nullable(?T)` and back should always succeed and give the original value.

Soundness. Converting a `Nullable(?T)` to `T` should *never* return anything other than `T`. It has to either return some `T` or abort.

Minimal overhead. Null checks will be unavoidable, but at least space overhead should be zero.

On the orher hand, we do *not* require the ability to discriminate between null and non-null. This is already handled by the user (such as array list) itself. The conversion of `null` to `T` is not required to crash, if it can return a valid `T`. This is essential to represent nullable integers without any overhead.

The interface

The nullable type and functions for working with it cannot be implemented in the library, they must be a part of the runtime. However, they do not need to be a part of the language and there is little reason to do so (unlike, e.g., functions). As such, they are added to the same place as other functions with a regular type but "magic" implementation, such as the arithmetic operators: a pre-generated bytecode file called `base_base.kbc`.

The new functions are:

- `Nullable: (?) -> ?`, a type constructor. It takes a type variable and returns a type variable.
- `null: (T: ?) -> Nullable(?T)` creates a null value for the particular `Nullable(?T)`.
- `asNullable: (T: ?, T) -> Nullable(?T)` converts a given `T` value to `Nullable(?T)`. We need a conversion function because the type is not built-in. This is probably fine, given that nullable types are only to be used in the "gory internals" of standard (and possibly few other) library data structures.
- `asNonnull: (T: ?, Nullable(?T)) -> T` converts a `Nullable(?T)` to a `T`. The result of `asNonnull(?t, asNullable(?t, t))` is always `t`. The result of `asNonnull(?T, null(?T))` is intentionally undefined and

may depend on `T`. It may crash or return an arbitrary (but valid) value of type `T`. As with the previous bullet, this is a function and as such it must be explicitly applied. In this direction, it is certainly a good thing, as the conversion might fail.

- `isNullable: (T: ?) -> Bool` returns whether null is a valid value of `T`, i.e. it returns true iff `asNonnull(?T, null(?T))` *should not* crash. This is only intended for testing of the runtime.

- There is no `isNull: (T: ?, t: Nullable(?T)) -> Bool`. This is intentional.

Currently, `isNullable` is true e.g. for integers and `Nullable` itself, and false e.g. for boxes, function objects and type objects. As a consequence, if `Nullable` is nested, only the innermost `asNonnull` may fail (depending on the base type).

2.5 Summary

To summarize, in addition the implementation fixes, we have drawn up the following list:

- Syntax modifications (2.1)
- Overloading (2.3.1)
- Inferred parameters (2.3.2)
- Implicit parameters (2.3.2)
- Opaque types by default (for encapsulation – 2.3.3)
- (De)constructing named & dependent types (for metaprogramming – 2.3.4)
- Function and conditional expressions transparent to the type checker (for both encapsulation and metaprogramming)
- Continuations (for coroutines – 2.3.5)
- Bytecode (2.4.1)
- Nullability (2.4.2)

This is an extensive list and completing all its items will take time. Some items must therefore be prioritized to the exclusion of others, as decided by the practical need for them.

The first item must be done first, as postponing it would slow down the development of the library and would also mean that all code will at some point have to be rewritten.

Nullability is literally indispensable for the collections library. Inferred parameters are not absolutely necessary, but their absence still makes using these collections extremely inconvenient. As such, these two will also need to be implemented.

The remaining items can be done without, and thus their relative ordering is not so clear.

Coroutines are explicitly suggested by the guidelines, albeit just as an example. Additionally, they can be used to implement async/await, which is becoming a standard feature of modern general-purpose programming languages.

The bytecode currently does not enable much, but the support for it has a significant impact on the architecture of the implementation. The current implementation is an AST interpreter (transformed to a compiler by Truffle). Having to produce a lower-level representation makes it closer to a compiler. Due to its impact on the architecture, it should again be done early to avoid later rewrites.

Making types opaque by default is not difficult, but replacing valid uses for the current behavior is. It does not enable almost anything new. Therefore, the implementation of this feature is postponed for now. The code of the library will act as if this restriction was already in place.

The most open-ended part of the analysis was metaprogramming. It will require further analysis and a lot of experimentation. It has also the most prerequisites: implicit parameters with overloading (for `getInfo` or its equivalent) and expressions transparent to the type checker. Additionally, it is possible and quite practical to use a language with only syntactic/textual macros (such as C), or not using metaprogramming at all (for example, the implementation attached in this thesis does not use Java reflection except for instance checks). As such, implementing the two features necessary for metaprogramming is left for future work. The most noticeable consequence (although by far not the only one) is that there will be no auto-generated by-value comparison/hashing and debug printing. Interestingly, this is a very common limitation, present in a wide range of languages, such as C, C++, and Java/C# before 2020.[4]

Implicit parameters can be made up for by just always passing the parameter. This is usually trivial, except for the recursive cases (which are only required by the metaprogramming as designed in 2.3.4). Similarly with overloading.

The author intends to finish these items in the future.

[4]In that year, both Java 14 and C# 9 coincidentally introduced a feature called records, which addresses this.

3. Implementation

A prototype implementation has already been attached to the original thesis. However, large portions of it had to be rewritten to allow better static analysis. This analysis is necessary for dependent type checks, as explained below, in Section 3.5.2.

3.1 Usage

This section documents the command-line interface of the compiler and bytecode interpreter. Details regarding building the compiler and other setup can be found in the README file in Attachment A.1. The command-line interface consists of four main classes (listed below). This separation is significant from the perspectives of both the user and the implementation, which is why we also describe it here.

- `cz.cuni.mff.d3s.kampa.exec.CompileOnly` takes a single source file, compiles it and produces a single Kampa bytecode file. The source file may import other files (source or bytecode), in which case these are resolved, compiled in case of source files, and included in the resulting bytecode file.
- `cz.cuni.mff.d3s.kampa.exec.ExecuteCompiled` takes a single bytecode file and executes it. It cannot compile source files. It also never imports anything (since the bytecode does not have a notion of import).
- `cz.cuni.mff.d3s.kampa.exec.CompileAndExecute` is roughly a composition of the previous two. It does not serialize to the bytecode and then deserialize, but it still uses the IR.
- `cz.cuni.mff.d3s.kampa.exec.GenerateBaseBase` takes no arguments, reads no sources, and produces a bytecode file at a constant file path. The resulting bytecode file contains functions that are hard or impossible to implement in the language itself, such as arithmetic operations.

A bytecode file can be used everywhere a source file can. This means not only imports, but also arguments to `CompileOnly` and `CompileAndExecute`. However, the utility of this is limited. `CompileOnly` just reencodes the file, and `CompileAndExecute` behaves like `ExecuteCompiled` except that it involves the compiler.

3.2 Structure of the compiler

The part that compiles Kampa source files to the bytecode consists of several passes described below. Each file is processed separately, going through all the passes at once.

Imports are recognized and processed in the first pass. Before the analysis of a file can continue, the imported file must be fully processed.

Lexical and syntactic analysis

This pass is mostly preserved from the original implementation. The lexical analyzer is extremely simple. The parser is more involved, since it has to expand macros and parse operators with user-defined precedence. Refer to the original thesis for a more detailed description.[1]

The result of the syntactic analysis is an immutable abstract syntax tree (AST) that closely matches the source but has all macros expanded and parser directives removed.

Conversion from AST to annotated AST

This step converts to a representation that allows storing the type, value, and possibly other information in its nodes. It allows non-tree links between the nodes, corresponding to variable references. The annotated AST representation is mutable, which allows types to be assigned in the order required by dependencies rather than in a simple bottom-up order. This is necessary in order to break cycles in nodes that support it (e.g. box types and function types).

Semantic analysis

This pass adds annotations to the AST nodes. It is implemented in the individual node classes.

In addition to resolving and checking types, it also generates the IR for values and verifies that all recursive definitions are valid. Kampa is eagerly evaluated, and therefore only functions and boxes are allowed to recursively reference themselves. However, forward references are generally allowed, which necessitates scheduling.

Once the semantic analysis completes, any malformed program should have been rejected. There is currently one exception. As said in Section 2.1.3, type annotations are not be checked in unreachable code. However, the semantic analysis is not able to detect whether a particular piece of code is reachable. As such, when it detects an incorrect annotation, it intentionally generates an invalid IR value (`IRInvalid`). This causes a later phase to crash and report an internal error. This hack is in place until reachability detection is implemented in the semantic analysis (as proposed in Section 2.1.3).

Control-flow construction

This pass uses the schedule generated by the previous pass to generate a control-flow graph. It inserts branches and edges for conditions, loops, and gotos.

IR cleaner

This pass removes information that is only used by the semantic analysis to perform generalization (see Section 3.5.1). This pass also changes the representation of types (described in the same section).

IR control-flow cleaner

This pass removes nodes that have no side effects and no users from the control-flow graph. Now, it is just an optimization, but previous versions of the bytecode interpreter depended on it.

Closure generator

Up until this pass, inner functions can use IR nodes from their enclosing functions implicitly. This pass locates all such uses and replaces these uses with a special type of IR node (`IRCapture`). It also updates the function's definition to specify the type of its closure (a tuple type of the captured IR nodes). Finally, it updates all the nodes that create function objects of this function to specify a tuple of IR nodes to be captured.

This pass cannot be executed before the IR cleaner because it requires the lower-level representation of types. It could be easily updated to be able to execute before the IR control-flow cleaner, but this could only result in unused nodes to be captured.

Doing the closure already during scope lookup only works for ordinary inner functions, not continuations. It would also complicate the type checker, which should be able to compare values regardless of where they are defined.

3.2.1 Bytecode generation

Bytecode generation is *not* a pass comparable to the above. While the above passes execute on each file separately, the bytecode generator is called directly from `CompileOnly` once all files have been processed. Also, in the case of `CompileAndExecute` , the bytecode generator is not used at all.

As said in Section 2.4.1, the bytecode format consists of two layers: serialization and IR. Since the previous passes already generate the appropriate IR, the bytecode generator only needs to serialize it.

3.3 Structure of the bytecode interpreter

The interpreter does not interpret the bytecode directly. Instead, it first deserializes it, getting the IR graph. This step is only necessary in `ExecuteCompiled` . In `CompileAndExecute` , no bytecode is even generated.

Not even the IR graph is interpreted directly. It requires some lowering. This lowering happens in a single pass that does the following:

- It converts the mixed control-flow/data-flow graph into sequences of "instructions" arranged in basic blocks.
- It replaces member accesses and array element accesses with offset calculations and loads/stores.
- It drops all type information. (Except when "slow types" are enabled, which is the case only when compiling to bytecode, not at ordinary run-time.)

At runtime, every function remembers its IR. When executed for the first time, it is lowered and the result of the lowering is cached for all later calls. This means that the interpreter essentially simulates a JIT compiler. Just instead of compiling to machine instructions, we compile for a naive and inefficient virtual machine, whose code is represented as collections of objects implementing some `Instruction` interface, and whose memory consists of `java.lang.Object` arrays (as detailed in the next section).

3.4 Memory representation

The primary purpose of the implementation is to test programs written in the language and to evaluate what it involves to compile it to a low-level representation. Performance is not the main objective, and it can for now be sacrificed for easier debugging and faster development.

As such, we use Java's implicit checked casts, null checks, and array bound checks to detect at least some errors in the implementation, but we never directly use the information available (e.g. using `instanceof`, `== null`, `.length`). Therefore, raw memory with pointers would be enough to execute the same "instruction set" but it would catch fewer bugs.

The memory representation of a value consists of zero or more Java objects, whose classes are determined by the value's type:

- **Number:** a single `java.lang.Long` instance.
- **Tuple:** the concatenation of its items.
- **Function:** a single `FunctionObject` instance (see below).
- **Named:** in the same way as its wrapped type.
- **Box:** a single `java.lang.Object[]` instance. The size of the array and types of its elements depend on the box content type.
- **Array:** the concatenation of its elements.
- **Type Variable:** a single `TypeObject` instance (see below).
- **Type:** nothing (same as an empty tuple).
- **Nullable:** if non-null, same as its wrapped object,
  if null, a sequence of `java.lang.Long(0)` of the same size.

Function objects

Function objects are represented using `FunctionObject` instances. The class has two read-only fields. One refers to the function IR, the other is an `Object[]` holding the closure data. In a practical native implementation, the representation would probably consist directly of two `size_t`s:

```
struct FunctionObject {
    void *code;
    union { void *ptr; size_t val; } data;
}
```

Type objects

Type objects are represented using `TypeObject` instances. The class has four read-only fields, but only the first two of them are normally used: `int size` and `bool nullable`. The former is the size of the values of the type represented by this type object. The latter is a flag is exactly the value returned by `isNullable` (Section 2.4.2).

The size is in OOPs (JVM ordinary object pointers), i.e. the number of elements the value takes in an `Object[]`. In a practical native implementation, it would be in bytes. It would also require additional flags besides `nullable` for alignment information. Still, the size of a type object would fit in 64 bits on both 32-bit and 64-bit platforms.

Variable-length stack allocations

We address the problem of local variables whose size is not a compile-time constant. The original implementation did not support them at all. This was due to the stack frame being represented as a constant-sized object. Its size could depend on the function but not on anything else. The original implementation used a relatively efficient scheme where tuples could be split among multiple frame slots. This allowed each tuple item to be stored directly in the stack frame.

On the other hand, the current implementation stores every value in just one slot. This means that values consisting of multiple objects must be wrapped in an additional `Object[]`. However, it allows variable-sized values in the stack. There is still an optimization that values with a constant size of 1 are stored directly. The splitting of larger values into multiple slots, when combined with supporting non-constant allocations, was too complex. In summary, the current implementation trades efficiency for generality and simplicity. This allows local variables of unknown types and also aligns with the purpose of the implementation.

3.5 Dependent types

The handling of dependent types pervades the whole implementation. The original implementation disallowed them in some contexts to avoid this (including for example stack variables, as said in the previous section).

The redesigned semantic analyzer instead tries to reduce the complexity associated with them. This allowed significantly extending its capabilities, but also caused some problems. We will describe both sides in this section.

3.5.1 Representation

Original solution

In the original type representation, types never referred to values directly. Instead, they referred to parameter indices. These indices pointed into an immutable list of values that was stored separately (we call it the *argument list*). For example, the array type `Int...N` becomes `Int...Param0` and an argument list with one element, ⟨ `N` ⟩. If we use a type with such an external dependency in a tuple, then the tuple itself will have that dependency. It also remembers which items needed which dependencies and explicitly forwards these. For example, the tuple `(Int...N, Int...M)`'s items are (as per the previous) both `Int...Param0`, and its argument list is ⟨ `N` , `M` ⟩. To ensure each item refers to the correct argument, the tuple forwards the arguments to each item separately. In the array pair example: `(Int...Param0 ⟨ Param0 ⟩, Int...Param0 ⟨ Param1 ⟩)`. The `Param`s in each item type refers to the item's own list. Each of the lists refers to the tuple's own parameters, ⟨ `N` , `M` ⟩. This forwarding can have many levels. For example, the function type `(Int...N, Int...M) -> (Int...L)` will become: `(Int...Param0 ⟨ Param0 ⟩, Int...Param0 ⟨ Param1 ⟩) ⟨ Param0 , Param1 ⟩ -> (Int...Param0) ⟨ Param2 ⟩` with an argument list of ⟨ `N` , `M` , `L` ⟩. When a tuple contains both a type that depends on a value and that value. For example, the sized array type is a tuple of a number and an array that depends on that number. Written as `(N: Int, Int...N)` in the source code, it is represented as `(N: Int, Int...Param0 ⟨ Sibling0 ⟩)`. As before, the `Param0` just points into the item's argument list, ⟨ `Sibling0` ⟩. The `Sibling` refers to an item of the same tuple (as opposed to a parameter as before). `0` is the index of that item. The argument list of the whole tuple is empty, ⟨⟩. Analogically with functions, where a `Sibling` in the return type refers to the function's parameter.

The advantage of this representation is that whenever a substitution is needed (e.g. during generalization and instantiation), it is enough to process just the list items, but the type can stay the same. When extracting an item from a tuple, one just takes the item's argument list and replaces each element with the corresponding element from the tuple's list.

Original solution limitations

The disadvantage is that this representation requires that when a type is created, all values it uses are already known. This is not necessarily true. For example, the arguments and return types of functions are processed lazily. They are not known at the time of the construction of the function type. In fact, they cannot be known in the case of recursive types (the type would require to be processed before itself).

The implementation attached to the original thesis used many workarounds and restrictions to cope with this. One of the consequences was that type definitions (i.e. Type types) were not allowed to refer to any values at all. Doing so would result in the type error "Dependent type not allowed here."

Type types depending on values are more important than they seem, since they are the way to define generic types. A generic type is a function that takes a Type Variable and returns a Type. If the Type cannot use the Type Variable, it is almost useless.

Current solution

After a short-lived experiment with list bindings replacing the immutable argument lists, the representation based on explict argument lists was given up entirely.

The immutable lists are still used by the IR, and list bindings are used to build them in the IR cleaner (Section 3.2), but they are no longer used in the semantic analyzer.

Instead, each tuple type and each tuple expression is identified using a `NodeID` instance. A type uses the same ID as the expression from which it was created. `NodeID` is a Java class with no fields and no methods, whose objects are only used for identity comparisons.

When a type needs to refer to a value, such as the array type `Int...42`, it just contains the IR of the value directly, `IRConst(42)` in this example. The value could again be a variable reference, `Int...N`, in which case the IR contains information about both its location and its value. For example, if `N` is defined as `42`, the length of `Int...N` will be `IRSibling(nodeID, i, IRConst(42))`. `nodeID` here refers to the `NodeID` of the lowest common ancestor of the AST node defining `N` and the current array AST node. `i` is the index of `N` within the ancestor.

Note that there already is a `Sibling` information even though the array is not yet a part of the tuple. When the type is processed, for example by a type checker, free `NodeID`s (i.e. those not found in enclosing tuples) are ignored. That is why `IRSibling` also contains the definition (`IRConst(42)` in the example).

When the array is put into a tuple next to its length (`(N: Int, Int...N)`), it does not have to be changed. The only thing that changes is the meaning of the array length, the `IRSibling(nodeID, i, IRConst(42))`. `nodeID` is no longer free. It is bound by the enclosing tuple type. As such, the value of the `IRSibling` is ignored and only the sibling relation is taken into account (e.g. in type checks).

Although `IRSibling` is used as an IR node, it is only relevant in the semantic analysis. The IR uses the explicit argument lists and does not need any `IRSibling`. As such, they are removed by the IR cleaner (Section 3.2). Depending on the context, the IR cleaner replaces them with either the value (`IRConst(42)` in the example) or a `Param` pointing to a `Sibling` (as shown in the previous section).

Problems with the current solution

As said above, this representation makes joining values into tuples easy. The analysis does not have to traverse type trees or even the argument lists. On the other hand, the extraction of tuple items is hard, since the item type needs to be traversed and all `IRSibling`s using the tuple's `NodeID` must be substituted. Similarly when calling functions: the return type must be traversed to substitute references to the function parameter.

The substitution also has to ensure it does not accidentally bind a `NodeID`. As such, all `NodeID`s defined in the return type must be renamed first. All this is already inefficient, but it becomes unusable in case of recursion. Consider the following example:

```
LinkedList: (T: ?) ->
            ?[head: T, tail: LinkedList(?T)];
```

Conceptually, this is a recursive generic type. To the semantic analysis, it is a function that returns a type. To infer its return type, we need first to infer the type of the `LinkedList(?T)` call inside of it. To do this, the semantic analyzer must:

1. Obtain the type of `LinkedList`. This is not a problem. The analyzer is designed to work well with recursive types by using lazy evaluation and caching.

2. Obtain its return type. Again, this is fine. The return type will already be available when it is first queried for.

3. Rename all `NodeID`s in the return type. Renaming in recursive types is all right. Caching and lazy evaluation is used so that types that have already been seen are not processed again.

4. Substitute the current `T` into the definition. Same with renaming, this works with recursive types.

On itself, none of the components can recurse indefinitely. The problem with a definition like this is that there will be an instance of the renaming visitor and an instance of the substituting visitor, each doing its caching, but each feeding its results to the other. Starting with just one type, one of the visitors creates a new type not seen by the other, which prompts the other to create a new type not seen by the first one.

Even if we complicated the analysis by merging these two, they would still hang on a pair of mutually recursive functions like this:

```
LinkedListA: (T: ?) ->
            ?[head: T, tail: LinkedListB(?T)];
LinkedListB: (T: ?) ->
            ?[head: T, tail: LinkedListA(?T)];
```

The only way to reliably prevent infinite recursion in all places in the semantic analyzer is to never create types from other types (e.g. by renaming and substitution). The only acceptable way to create a type should be creating it from an expression in the AST, since there is a finite (and fairly limited) supply of AST nodes.

As such, the argument lists as in the original representation are necessary. To avoid the need to know each type completely to be able to use it, the argument lists can be "preventively" populated with all variables available. In the common case the type does not need all of them, it will just not use all its parameters. There needs to be an efficient way to represent sets of variables (the argument list cannot just contain an element for every variable in scope). This should probably be something that copies the structure of the lexical scope, just not using variable names but the forwarding shown in the original type representation. A concrete design is left for future work.

For now, the following workaround can be used to describe recursive generic types (the result is exactly the same but does not hang the compiler):

```
// non-recursive generic type
LinkedListImpl: (T: ?) -> {
    // non-generic recursive type
    .Type: ?[head: T, tail: Type];
}
// non-recursive generic type
LinkedList: (T: ?) -> LinkedListImpl(?T).Type;
```

3.5.2 Value equivalence

Checking the equivalence (or compatibility) of dependent types requires deciding whether two values will be equal at runtime. This is known to be impossible, so the type checker has to use a well-defined approximation. When it cannot prove two values to be equivalent, it rejects the program.[1]

Originally, two values were only considered equivalent when their memory location was the same. There were not many other options. While the generated program consisted of value nodes, these could not be inspected by the type checker, only executed. As said above, this thesis replaces the intermediate representation with one that is not useful for direct execution, but can be inspected instead. As such, comparing two values is now a matter of comparing two expression-like structures. The most important consequences include:

- Two instantiations of the same generic type with the same arguments are considered equal. For example: `ArrayList(?Int)` and `ArrayList(?Int)` or `ArrayList(?T)` where `T` is known to be `Int`.
- Arrays can have in-place constant lengths and be considered compatible if their lengths are equal.

[1]Another possible approach would be trying to prove the values can differ and only rejecting in that case. However, the type system of Kampa is intended to be sound and thus does not honor the presumprion of innocence.

4. Evaluation

In this section, we evaluate Kampa's generality, expressivity, and ease of use. Regarding expressivity: the language on itself intentionally does not attempt to be expressive. Instead, it allows libraries to define macros that look like built-in constructs in other languages.

4.1 Kampa Runtime Library

In this section, we design a standard library that uses the language well (while not artificially overusing its specifics). We compare the tools provided by Kampa with their equivalents provided by other languages where applicable. We will also compare the design decisions with other languages' libraries.

4.1.1 Optional

The *optional* type (also called *maybe* or *option*) is a type whose value can be either a wrapped value of another type (*some*, *just*) or a special value representing the absence of any value to wrap (*none*, *nothing*, *empty*).

We have already discussed a similar feature, nullable types, in Section 2.4.2. However, nullability is a low-level feature for cases where no optional value is required from the program logic point of view. It is only used to locally opt out from some guarantees provided by the language. It only makes sense to use nullability as an implementation detail.

On the other hand, `Optional` can be used when there is real need for a value that can be absent, i.e. both alternatives have some meaning in the program's logic. It is not specific to pointers and it can be meaningfully used for interfaces as well as for implementation. The main use case for the optional type is in return values of functions that may, even under normal circumstances, produce no result, such as looking up a matching item/key in a collection. The program using the library can then branch or carry out additional operations depending on whether `some` or `none` was returned.

In Kampa, there are currently roughly four possible ways to represent an optional value of type `T`:

1. `present: Bool, value: T...present` – an array whose length is a bool. `1, t` stands for `some` (where `t` is the wrapped value), `0, ()` stands for `none`. Note that this type has a variable length, and thus cannot be used as an array element type or a type variable.

2. `[present: Bool, value: T...present]` – same as previous, except that it adds indirection. This solves the previous problem (a reference has a constant size), but it may harm performance due to additional heap allocations and random accesses to memory. This is the approach taken by Java's Optional (except that Java can share its empty optional among all `T` by using unchecked casts, reducing memory footprint and allowing presence query without dereference).

3. `present: Bool, value: [T...present]` – a variation on the previous alternative. This allows even faster presence query, and with a slight modification also the sharing of empty among all `T`. However, the additional reference in the (presumably average) case of `some` remains.

4. `present: Bool, value: Nullable(?T)` – always allocating space for the value, even for `none`. Like the first alternative, this one involves no references. Unlike the first alternative, the size is invariant. This is practically always an advantage, since the possible memory savings of the first alternative are nullified by the fact that it cannot be stored in an array (scrimping bits on singular instances just leads to overhead). This is the approach taken by C++ (with the default value being undefined).

In alternatives 1, 2, 3, it is possible to replace `T...present` with a union of `()` and `T`. This is useless for alternatives 1 and 2, since it just adds complexity. In alternative 3, though, it can be used for the sharing of empty values.

Alternative 2 can be eliminated, as it has absolutely no advantage in comparison to alternative 3. It occupies the same amount of space, it has slower presence checks and it cannot share its empty instances.

Similarly, the space-saving advantage of 1 in comparison to 4 is rarely significant and will more often lead to unnecessary overhead. While 1 can still be useful, it should not be the main optional type.

Finally, `Optional3(?T)` is similar to `Optional4(?[T])`. In fact, the latter is more general, since it allows `T` to have variable length. And avoiding the one global empty box, albeit insignificant in terms of memory consumption, is definitely cleaner.

Functions

The library offers `optUnwrap`, which checks the `present` flag and either returns the value or aborts the program. Querying the flag and then using `optUnwrap` to obtain the value is the most universal way to use `Optional`, but it is not very reliable (a flaw in a program could result in unwrapping a non-present `Optional`).

As such, the library also offers a functional API, which currently consists of the following functions: `optMap`, `optFlatMap`, `optElse`, `optElseGet`, `optOr`, `optOrGet`. Their implementation relies on `optUnwrap`.

4.1.2 Collections

A classical example of a standard library collection is a growable array. In our library, it is called `ArrayList` after Java's `ArrayList`. Java (as well as dotnet) uses a definition similar to `class { Object[] data; int size; }`, or `[@data: [length: Int, @items: T...length], @size: Int]` in Kampa syntax (`length` is the capacity). Kampa offers slightly more freedom with respect to the layout.

First, we may use an optimization for short lists, where the elements can be stored in the list object directly. This is not possible in Java and C#, since arrays are always reference types (and using e.g. eight named fields would only add complexity).

Second, the outer box (pointer) can be dropped in case the list has only one owner (and no other references), which might be a common case. This is possible in C# but not Java. This should not be the main implementation of `ArrayList`, since it is slightly restrictive; the user must ensure no copies are made, or more precisely, that only the latest copy is used.

For now, we stick with the plain approach, since the performance is dominated by other factors anyway. There already is a theoretical improvement over Java. In Kampa (similarly to C#), generics do not use boxing[1]. This means that in `ArrayList(?Int)`, the integer values can be stored directly in the array, not as individual heap objects.

The library also implements `ArrayDeque` – a queue that allows addition and removal from either end in $O(1)$ amortized time, backed by a growable array (but not using `ArrayList` internally because the growing must work differently). The reason it was added was the need for a queue data structure in async (which will be discussed in Section 4.1.4).

4.1.3 Iterators

There are several ways to represent an iterator. Practically every language/environment has its own:

- Python `iter` has a single method, `next`, which returns an object and updates the internal state of the iterator. The end is signalized by the method throwing a special exception.
- Java `Iterator` is similar to the previous. Additionally, since exceptions are considered expensive, not well suited for this purpose, and more difficult to handle when using an iterator manually, there is also the `hasNext` method, which generally does not update the state, and returns `true` iff `next` will not throw.
- C# `IEnumerator` also has two methods, `get Current` and `MoveNext`. Their responsibilities are slightly different than in Java iterators. `MoveNext` returns the a boolean (like `hasNext`), but it updates the state (like `next`). `MoveNext` should always be called before the first access to `Current`.
- C++ iterators have three methods. `operator++` moves the iterator (and returns the iterator itself). `operator*` (unary) returns the current element. `operator==` can be used to test whether the iterator ended (by comparing to another object, the end iterator, which might not even be an iterator).

Implementing an iterator

Python's approach has the advantage that the iterator does not have to either store the element or consider edge cases where one of the methods is called repeatedly without calling the other (although one edge case remains: calling `next` on an already-ended iterator). This means fewer ways an iterator implementation can deviate from the specification or the convention.

[1]The implementation, being written in Java, will use boxing anyway. See Section 3.4 for details.

Ways to implement iterators vary. In all of the mentioned languages, it is possible to implement them directly by writing a class. In Java (and C++ until C++23), this is also the only way. Python, C#, and C++23 offer an alternative in the form of generators. A generator is a function/method that, when called, does not execute and instead returns an iterator. The body of the function is only executed by using the iterator. The body contains *yield* statements, which pause the execution and return a value to the user of the iterator, which can then use the iterator again to resume the body. This repeats until the body ends (using return or reaching its end).

Writing a generator is at least as easy as writing the class. Formally: rewriting an iterator class to a generator is trivial: `for(x in existing_iter) yield x`. The opposite direction is not hard, but it can be annoying and lead to less readable code.

Using an iterator

As shown in passing in the previous paragraph, an iterator can be used in a for-each loop. This is true in the current versions of all four mentioned languages (albeit with varying syntax).

Some iterator objects support other operations. Some C++ iterators support pointer-like arithmetic. Java iterators support element removal. Some C# enumerators can also be reset to the initial state using the `Reset` method. However, this throws an exception when used on enumerators implemented as generator methods.

Interestingly, C++ iterators and C# enumerators can in some cases be copied in the middle of an iteration, creating two iterators that can be used independently. Again, this does not work with C# enumerators written as generators, since those are always references.

Iterators in Kampa

Kampa's support for continuations (Section 2.3.5) can be used to implement generator functions. The state of the generator can be represented using just the continuation (since it contains both the code and local variables).

We will start with an interface similar to Python's, i.e., just a `next` method. Instead of throwing an exception (not available in Kampa), the method could return an `Optional`. When a generator method is called, it just creates an iterator object in its initial state and returns immediately. The first call to `next` enters the body and runs until a `yield` statement is reached. The `yield` statement now has to take the continuation, save it to the iterator object, and return a `some` to its caller. Following calls to `next` just call the continuation (i.e. closure) saved in the iterator state. The initial state can also be represented as a closure, making `next` as simple as a single dereference and call.

Generators are often implemented using state machines, where states correspond to `yield` statements in the body (with some states being initial, final, failed, etc.). The current state is stored in the iterator object, together with a record of the generator's local variables. Notice that this is very similar to storing the closure, just instead of the numeric state index, we store a code pointer.

Back to the interface: this design can be further improved by replacing shared mutable state with local (mutable) state. The iterator object is not necessary if the `next` method returns a new iterator (together with the element) instead. In other words: `yield` does not modify anything and returns a `(head, tail)` pair, where `head` is an element and `tail` is another iterator. The iterator now does not need to *store* the closure. It can *be* the closure.

This leads us to the iterator type: `Iter(?T)` is a function returning an optional head/tail pair, `() -> Optional(?(head: T, tail: Iter(?T)))`.

We make one final modification that is rather pragmatic than theoretic. An iterator should probably not be directly callable. In most cases, using an iterator like a function (`iter()`) would be done accidentally, yet it would be valid and evaluate to the abovementioned `Optional(...)`. To avoid this, we wrap the function in a named type. The syntax necessary to the optional thus becomes `iter.start()`, which is unlikely to be done by accident. We define the for-each loop to always use `.start()`.

As said above, `yield` is responsible for creating the head/tail pair. It wraps it in `some` and returns it. We still need something that (i) pauses the execution in the beginning, returning a `start: () -> ...` and (ii) handles the common case, where the generator reaches the end of its body (and returns `none`). Both is done by one macro, called `generate`, in which the whole body is wrapped. Since `generate` creates a function on its own (to pause in the beginning), it does not place any requirements on the enclosing function. As such, `generate` is just an ordinary expression that returns an iterator.

Nondeterminism

Besides `yield`, generators can also contain `select` expressions. The syntax is `select from foo`, where `foo` is an iterator of any type (regardless of the current iterator type). The result of the expression is a single element of `foo`. The `select` expression forks and returns multiple times, once for each `foo`. In other words, the rest of the generator body runs once for each item of `foo`. All the items yielded by all the forks are included in-order in the resulting iterator.

`select from` can be compared to `<-` in Haskell `do` notation and its implementation is similar. In Haskell, `elem <- foo` calls `foo >>= (\elem -> ...)`, where `...` is the rest of the do block. In Kampa, `elem: select from foo` calls `foo iterFlatMap (elem -> ...)`, where `...` is the rest of the generator body (i.e. a continuation).

Other functions and macros

In addition to the `Iter` type, `generate`, and `yield`, the library provides:

- `emptyIter`, `iterFilter`, `iterMap`, `iterFlatMap` – simple functions that are themselves implemented as generators
- `for(`*`var`*` in `*`iterator`*`) `*`action`* – a macro that simplifies iterator usage
- `yield from `*`iterator`* – a macro for use in generators, it is a shortcut for this loop: `for(elem in `*`iterator`*`) yield elem;`
- `yield break` – another macro for use in generators, it does not take any value and terminates the generator

4.1.4 Asynchronous programming

We have already touched `async` and `await` when designing coroutines in Section 2.3.5, and ended up with this example:

```
? "await" future = $breakLabel: TValue {
    callback: (value: TValue) -> TFuture2 {
        $breakLabel = value;
    }
    future2: future.then(callback);
    return future2;
}
```

This intentionally omitted an important detail. The value called `future` by the example is often something returned by another `async` block. It contains code that must be executed at some point. However, it is not clear who ensures this and how. The executor cannot be a global object – Kampa intentionally does not support global mutable state. The executor must be “found” in some way. There are several options:

1. Async blocks start themselves. They run immediately and synchronously until the first await that would suspend. To run synchronously, they do not yet need access to the executor. The await then takes a future object, which will presumably fulfill at some point. The fact that it fulfills means that it was run by the executor and thus can have access to it.

2. Async blocks do not start themselves. They are suspended from the beginning. They are only executed when they are awaited. Note that the awaiter can always already have access to the executor; this directly follows from the fact that it is an async function that is already running (and thus already past its initial suspend).

3. The future objects could track dependencies. Using the names from the example: normally, `future` has a reference to `future2` (to be able to call it back). In the case of this option, there would (also?) be a reference from `future2` to `future`. The chain of the latter type of references leads from the async main function, whence all the remaining async code can be reached.

4. Avoiding the “problem” of missing global state by introducing a context object. It would be necessary to have a context object in scope in order to create an async block. The async block would be scheduled for asynchronous execution already when created.

Option 4 was the first one we experimented with. It seems to offer the most elegant design, but only in conjunction with some currently missing features (existential generic types – see Section 2.3.3). Until these are supported, we abandon this option, since the complexity added does not pay for itself.

Option 3 is probably very similar to option 2. Just instead of actively traversing the dependency graph, the executor is passed along its edges. However, in deeply nested async calls, there might be a significant difference. If a leaf function suspends, this information has to be propagated up the stack to the main function, only to be tracked again to the bottom just to discover what the leaf function waits for.[2]

Options 1 and 2 are much more distinct from each other, even from the user perspective. Yet the decision seems much harder and different environments choose different paths. In the Kampa library, we choose option 2, mostly for its consistency with generators and the (necessarily subjective) intuition that if it is an async(hronous) block/function, it should not run synchronously.

As such, the asynchronous programming library currently uses option 2. Asynchronous blocks return unstarted `Task`s which have to be started using the `.start(AsyncExecutor)` virtual function. This function already returns an object on which `then` can be called. `start` is called automatically by `await`, but a task can also be started manually without yet awaiting it.

The library also provides two simple implementations of `AsyncExecutor`. Both are single-threaded and based on a queue. However, there is currently no support for multithreading or asynchronous IO (or any IO for that matter). As such, asynchronous programming in Kampa can currently not be used for anything practical. It is just meant to demonstrate that the language itself can support it once the system interface provided by the runtime catches up.

4.2 Summary

The library makes heavy use of many of the additions. For some of the additions, this is inevitable. For example, without support for generic types, it would be nearly impossible to implement most of the utilities (unless using macros or specializing the definition for each type manually). Other additions, such as continuations, could be done without, but code using them can be significantly more expressive.

While the additions were only designed with `ArrayList` and asynchronous coroutines in mind, they proved to be general enough to be used for iterators, generators, optional, a string library. Furthermore, the features already present in the language enabled choices not offered by many other languages, such as the memory layouts.

Ease of use is necessarily subjective and it depends on practice. However, there are some factors that work in favor of Kampa (e.g. memory safety, use of automatic memory management, regularity). Probably the most visible limitation is now the lack of overloading, which requires the user to explicitly state which function to use, as can be seen from almost any code working with iterators and optional (e.g. `iterMap` and `optMap`).

All code described in this chapter can be found in Attachment A.2.

[2]This is similar to nested generator calls using `yield from`.

Conclusion

This thesis improves the Kampa programming language and its experimental implementation. The main objective was to make the language more practically usable. There are many aspects to practical usability of a language, such as tooling, availability of libraries, and even external factors (adoption, community). Addressing all of them is beyond the scope of a single thesis.

Consequenly, this thesis primarily focused on the language (syntax and semantics), its compiler/interpreter, and basic runtime libraries. Where necessary, design decisions were made while considering the needs of other tools.

Chapter 2 discusses a number of shortcomings in the language and suggests possible solutions, of which the most important ones were also implemented in this thesis:

- More consistent and readable syntax (2.1)
- Addressing several implementation limitations (2.2)
- Parameter inference (2.3.2)
- Continuations (for coroutines – 2.3.5)
- Compilation to Kampa-specific bytecode (2.4.1)
- The Nullable type (2.4.2)

The following shortcomings were left for future work, because they were not considered essential for this thesis:

- Overloading (2.3.1)
- Implicit parameters (2.3.2)
- Further type system improvements, especially its consistency (2.3.3, 2.3.4)

Chapter 3 describes the implementation of the language. Even though the language was designed to be compiled to native code, programs are currently interpreted. Large portions of the original interpreter had to be reworked to enable better static analysis needed for dependent type checks. The implementation is still considered a prototype quality – there are still are a few corner cases in which the implementation fails to detect some kinds of errors or detects them too late (Section 3.2, Semantic Analysis). It also exposed some deficiencies of the internal representation of types which could not be redesigned in the scope of this thesis (Section 3.5.1).

The language and interpreter improvements were necessary to enable implementation of a more extensive runtime library, as shown in Chapter 4. In terms of number of provided functions and types, the library is orders of magnitude smaller compared to libraries found in languages such as C#, Java, or Haskell. However, the provided functions and types are comparable in nature, ease of use, and generality with their counterparts in other languages. Some of the interfaces that programs can use to interact with the environment are still rudimentary and need to be replaced. But the improvements in this thesis have made it possible.

In closing, even though there is still a considerable room for improvement (and the author continues to work on Kampa with unabated interest), the improvements presented in this thesis have pushed Kampa towards practicality and made it possible to write code that is comparable to other modern programming languages.

Additionally, the implementation is currently very unreliable, failing to detect some errors or detecting them too late (Section 3.2, Semantic Analysis). Its internal representation of types needs a redesign, too (Section 3.5.1).

Bibliography

[1] L. Rozsypal. Kampa: an experimental programming language. Bachelor thesis, Charles University, Praha, 2020.

[2] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. *SIGPLAN Not.*, 21(11):214–223, 1986.

[3] P. Urzyczyn. Inhabitation in typed lambda-calculi (a syntactic approach). In Philippe de Groote and J. Roger Hindley, editors, *Typed Lambda Calculi and Applications*, pages 373–389, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[4] Filip Křikava, Heather Miller, and Jan Vitek. Scala implicits are everywhere: A large-scale study of the use of scala implicits in the wild. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019.

[5] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Syst.*, 10(3):470–502, 1988.

[6] Matthias Blume. No-longer-foreign: Teaching an ML compiler to speak C "natively". In *BABEL*, 2001.

[7] D. Syme, K. Battocchi, K. Takeda, D. Malayeri, D. Fisher, J. Hu, T. Liu, B. McNamara, D. Quirk, M. Taveggia, W. Chae, U. Matsveyeu, and T. Petricek. Strongly-typed language support for internet-scale information sources. *Technical Report MSR-TR-2012–101, Microsoft Research*, 2012.

[8] D. R. Christiansen. Dependent type providers. In *Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming*, WGP '13, page 25–34, New York, NY, USA, 2013. Association for Computing Machinery.

[9] ISO/IEC JTC 1/SC 22. ISO/IEC 14882:2020: Information technology – programming languages – C++. Standard, International Organization for Standardization, Prague, 2020.

[10] G. van Rossum and P. J. Eby. Coroutines via enhanced generators. PEP 342, 2005.

[11] D. Syme, T. Petricek, and D. Lomov. The F# asynchronous programming model. In *Practical Aspects of Declarative Languages*, pages 175–189, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[12] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One vm to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, page 187–204, New York, NY, USA, 2013. Association for Computing Machinery.

[13] Cliff Click and Michael Paleczny. A simple graph-based intermediate representation. *SIGPLAN Not.*, 30(3):35–49, 1995.

[14] C. Click. *Combining Analyses, Combining Optimizations.* PhD thesis, Rice University, Houston, Texas, USA, 1995.

List of Figures

A. Attachments

A.1 Kampa compiler and bytecode interpreter

The first attachment is a new version of the implementation attached to the Bachelor's thesis[1]. The description of the new version can be found in Chapter 3, including the command-line usage in Section 3.1. For build instructions, please see the `README.md` in the attachment.

A.2 Examples

The second attachment is a folder containing example programs. It includes updated versions of the original examples, together with the programs written as part of the evaluation (Chapter 4).