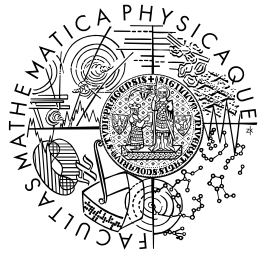

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Maya Mückenschnabel

Combining effects with dependent types

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: Tomáš Petříček

Study programme: Computer science

Prague 2024

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I want to thank first and foremost my supervisor, Tomáš Petříček, for having the patience with me, and for providing me with great guidance. I would also like to thank my cat Fazolka, for helping me with some of the more difficult theory, as she slept on my lap when I was trying to solve it. The thanks belongs to my partner as well, as they were very accepting and supportive when I talked to them about something they didn't understand. I wanted to thank my girlfriend for helping me formulate my thoughts and for her support. And lastly my thanks belongs to all queer and trans people for existing.

Title: Combining effects with dependent types

Author: Maya Mückenschnabel

Department: Department of Distributed and Dependable Systems

Supervisor: Tomáš Petříček, Department of Distributed and Dependable Systems

Abstract: Dependent type systems provide a novel way of reasoning about program correctness, by embedding behavior of the program into the more expressive type system. Correctness is achieved by not allowing incorrect states to be representable. Languages like Idris show that dependent type systems are practically useful, not only for formal proofs, but also for creating fewer bugs in production. But the purity of computation poses a problem for composability of stateful computations and of side effects. Effect handlers provide one possible solution for this problem. In this thesis we propose an effect extension of dependent type systems. The resulting system not only makes it possible to provide guarantees about correctness of a program, but also make it easy to compose such guarantees using effects. We formalize the type system and present a prototype implementation.

Keywords: dependent types, effect handlers, type systems

Název práce: Combining effects with dependent types

Autor: Maya Mückenschnabel

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Tomáš Petříček, Katedra distribuovaných a spolehlivých systémů

Abstrakt: Závislostní typové systémy poskytují nový způsob dokazování správnosti programů pomocí vkládání chování programu do výstižnějšího typového systému. Správnosti je dosaženo tím, že není možné neplatné stavy reprezentovat. Jazyky jako Idris ukazují, že závislostní typové systémy jsou prakticky použitelné nejen pro formální důkazy, ale i pro vytváření programů s méně chybami v produkci. Avšak čistota výpočtu představuje problém z hlediska složitelnosti operací se stavem a vedlejších událostí. Událostní lapače nabízí jedno z možných řešení tohoto problému. V této práci navrhujeme událostní rozšíření závislostních typových systémů. Výsledný systém poskytuje nejen garance správnosti programu ale rovněž činí složitelnost těchto garancí jednodušší pomocí událostí. V této práci představujeme takovýto typový systém a jeho prototypovou implementaci.

Klíčová slova: dependent types, effect handlers, type systems

Contents

Chapter 1: Introduction

In which we describe dependent types and effects

In programming, it is often vital to provide guarantees about correctness. These guarantees can be either expressed by contracts or by dependent typing. Unlike contracts, dependent types require a new type system. Such system is more flexible and allows writing programs that are correct by construction.

In this thesis we propose a combination of a dependent type system with effects. This introductory chapter explains what dependent types and effects are, how they are useful, and how can one understand them from the practical perspective.

1.1 Dependent types

Polymorphism has become ubiquitous in languages such as Java, C# and C++. It allows the definition of generic methods without manual casts.[1] In Java and C# the polymorphic abilities of functions and classes can only be expressed as parametrization per type. For example a type `List<T>` defines a collection of all items that are of type `T`. In dependent systems, such polymorphism is not restricted only to types. The nature of such parametrization is fully generic, allowing for types that are dependent on any computation. We can add compile-time information about the length of the collection by defining a type `List<T, n+m>`. As such, the type conveys richer information about the value being presented.

It may be tempting to showcase such systems using more mathematical languages, however dependent types are something that the everyday developer may have already utilized, if they used, for example, C++ templates.

Let us define a function append on `std::array<class T, size_t>`, the array that has a statically known length. Such array is itself a dependent type, but let us consider a function `concat` (See figure 1.1) which guarantees that the length of the resulting array is the sum of the lengths of the provided arrays with types.

Dependent types not only allow for greater correctness of the algorithms.[2] They may also provide hints to the compiler of available optimizations. In C++ the type-checking for dependent types is done with a certain degree of naïveté. The type system can still be subverted easily and the inference is lacking.

[1] As from the programmers perspective. In reality Java generics are type erased references, that are automatically cast upon manipulation. Such polymorphism is called dynamic. In C++ every template instantiation generates a new version of the code in the resulting binary, and thus the polymorphism is static.

[2] For example, giving arrays of length 2 and 3, and trying to save them into an array of length 6 will result in a compile error.

```
template <std::size_t N, std::size_t M, typename T>
auto concat (std::array<T, N> a, std::array<T, M> b)
  -> std::array<T, N+M>
{
  std::array<T, N+M> result;
  for (std::size_t i = 0; i < N; ++i)
    {
       result[i] = a[i];
    }
  for (std::size_t i = 0; i < M; ++i)
    {
       result[N+i] = b[i];
    }
  return result;
}
```

Figure 1.1: C++ dependent concatenation function.

Languages like Idris and Agda use more complex systems for type-checking their dependently typed programs. Those do provide better guarantees and also better inference. In this thesis, we use the bidirectional type-checking strategy proposed by B. C. Pierce and D. N. Turner in their article *Local type inference*[1] in the definition of our type system.

1.2 Effect handlers

Most programming languages provide some guarantees about correctness; either a static one, i.e. type-checking, or a dynamic one, i.e. throwing an exception. When using the static approach, we want the compiler to guarantee that incorrect state is unpresentable, and as such our program will never enter an incorrect state.

The safety guarantees of a language can be plotted on two axes, dynamic and static. On one end some languages do not provide any static guarantees, but they at least guarantee that the program will terminate gracefully without executing arbitrary code in memory. This is the area of languages like Python and JavaScript.

On the other end of the spectrum are languages like Coq, Lean, Idris, and Agda that can provide proofs about program correctness. This comes with a downside. Since the computations can be proven to be correct mathematically, they must themselves be pure. That is, they always produce the same output for each input.[3] Typically these languages provide a way to introduce side effects in a controlled manner, through a mechanism called monads.[2] That sidesteps the problem by making each computation that includes side-effects implicitly take a state and pass the possibly modified

[3]The computations can still not terminate or crash. In this thesis we do not provide mathematical purity, as the problem of totality is undecidable. However we plan to release an article on restricting such behavior into effects, without using heuristics.

```
int addPositiveValue(int a, int b)
  throws NotPositiveException {
  expensiveComputation();
  if (b < 0) {
    throw new NotPositiveException("b is not positive!");
  } else {
    return a + b;
  }
}
void main() {
  int a = 1;
  int b = -2;
  int result = Integer.MIN_VALUE;
  try {
    result = addPositiveValue(a, b);
  } catch (NonPositiveException e) {
    System.out.println(e.toString());
    System.exit(1);
  }
  System.out.println(result);
}
```

Figure 1.2: Java exception handling.

state to the next computation. The state passing introduces a temporal dependency between monadic computations, and models the changing state of the environment. For example, this makes it possible to interact with the operating system.

The problem with monads is that they cannot have generalized composition. There have been many methods proposed in, for example, *Composing monads*[3][4] and *Monad transformers and modular interpreters*[4][5]. However these methods do not provide a generalized mechanism of composition without the use of auxiliary functions. Either directly, as it is the case in the former article or through the `lift` operation which is the case in the latter article.

As language designers, we want to provide some trade-off between static guarantees and simplicity. Therefore the composability may not be worth it for some designs. In cases where composability is desirable, effects provide both generalized composability *and* simplicity[6].

To introduce effects for the everyday developer, we consider exception handling code in Java (See figure 1.2). This program expects that `b` is positive. If not, it throws an exception and exits with an error code.

What if we want this code to instead invert the `b` when it is not positive

[4]This method "depends on the existence of an auxiliary function linking the monad structures of the components".

[5]This method is the preferred by the Haskell programming language.

[6]We are using the term "simple" as is defined in the talk *Simple made easy* by Rich Hickey[5].

so that `addPositiveValue(1, -2)` returns 3? And what if the function `addPositiveValue` is not ours, but is it is of from another library which we do not have access to? In such cases we could certainly perform a second call to `addPositiveValue`, but that will perform `expensiveComputation` twice.

This is a contrived example, but the point is illustrated quite well. Ideally we would want to resume such computation instead of rerunning the whole function again. In that case, Java exceptions prove not to be expressive enough. But the extension is rather simple. We introduce a second argument to `catch`, a so called *resumption operation*, a function that when called, will resume the computation from where the exception was thrown, possibly with a new value. (See figure 1.3)

```
int addPositiveValue(int a, int b)
  throws NotPositiveException {

  expensiveComputation();
  if (b < 0) {
    return a +
      throw new NotPositiveException("b is not positive");
  } else {
    return a + b;
  }
}
void main() {
  int a = 1;
  int b = -2;
  int result = Integer.MIN_VALUE;
  try {
    result = addPositiveValue(a, b);
  } catch (NonPositiveException e,
           Resumption<Integer, Integer> r) {
    System.out.println(e.toString() + " inverting");
    result = r(-b);
  }
  System.out.println(result);
}
```

Figure 1.3: Java with exception resumption.

What we have shown here is the example of effect handlers in Java-like language. In languages with effects we would call `throw`, `raise` and `catch` a `handler`. As can be inferred by the reader, the resumption operation may not be invoked. In that case, we have the standard Java exception semantics.[7]

[7]The only difference is that Java allows for exceptions inherited from `RuntimeException` not to be apart of the function type. This is generally not the case for effect systems.

This is not the only method of introducing effects. If the developer is familiar with coroutines or Python's generators, the actions of `yield` and `resume`/`next` mirror what `raise` and `resume` operations do in languages with effects. This suggests that even `async` `await` operations are in fact, a specialization of effects, and that is indeed the case.[6]

1.3 Outline

In the following chapters we will:

- Formally describe lambda-calculus, its simply typed extension, introduce dependent types and effect systems, and explain and justify our decision for using bidirectional type-checking. (Chapter 2)
- Formally define the type system, show the necessary typing rules and introduce constraints, a solution for language constructs with non-syntax-driven types. (Chapter 3)
- Introduce Kelp, the proof-of-concept language with compiler and interpreter based on the aforementioned type system and show a few examples of programs written in the language. (Chapter 4)
- Conclude our findings and propose further work. (Chapter 5)

Chapter 2: Background

In which we describe the basis of our thesis

In this thesis we formally describe a theoretical type system. Since this may be a topic unfamiliar to many readers, this chapter is used as an introduction to the basic theory that is built upon in chapter 3.

2.1 λ-calculus

There are several formal models of computation. A model that is widely understood is the Turing machine, which models a program as a state machine with linear memory. In each step, the machine proceeds by reading a symbol from the memory at the current position, writing a symbol to the memory, incrementing or decrementing the memory position and updating the machine state.

λ-calculus is a different approach to describing computation introduced by Alonzo Church[7]. Computations are modeled as reductions of terms according to specified reduction rules. Terms are either variables, abstractions or applications. Formally we define the term t as follows:

$$\begin{array}{rll} t := & x & \textit{Variable} \\ | & \lambda x\ .\ t & \textit{Abstraction} \\ | & t_1 t_2 & \textit{Application} \end{array}$$

There are two reduction rules, α-reduction and β-reduction. We can omit α-reduction if we add a requirement that all bindings of variables happen only once. We can, for example, rename all variables to unique names.[1]

β-reduction is the interesting rule. It is applicable to sub–terms of the form $(\lambda x.t_1)t_2$, that is a case where a term of the abstraction form is applied to an argument t_2. The abstraction term $(\lambda x.t_1)$ is also often referred to as the lambda term. β-rule reduces the application term by taking the body of the lambda t_1 and substituting every occurrence of the variable x within t_1, with the term t_2.

$$(\lambda x.t_1)t_2 \rightarrow_\beta t_1[x := t_2]$$

It may be surprising that this is all we need to describe any computation. And in fact λ-calculus is Turing-complete[8]. It may be even more surprising that we don't need any other values than lambdas, not even numbers or

[1] α-reduction only performs the renaming of variables, i.e. $\lambda\ x\ .\ t[x] \rightarrow_\alpha \lambda\ y\ .\ t[y]$.

strings. All can be simulated by giving a semantic meaning to the structure. For example, we can define numbers as follows:

$$
\begin{aligned}
0 &:= \lambda s_0 \,.\, s_0 \\
1 &:= \lambda s_1 \,.\, 0 \\
2 &:= \lambda s_2 \,.\, 1 \\
&\dots
\end{aligned}
$$

2.2 Simply-typed λ-calculus

To extend the λ-calculus with types, we first modify the grammar. Now every lambda annotates its argument x with a type τ. Types can be either primitive types or composition of those types into function types $\tau_1 \to \tau_2$.

$$
\begin{array}{rll}
t := & x & \textit{Variable} \\
| & \lambda x : \tau \,.\, t & \textit{Abstraction} \\
| & t_1 t_2 & \textit{Application}
\end{array}
$$

The reductions remain unchanged. But before they are performed, the program is type-checked. We define the algorithm as a set of rules. If no rules apply, the type-checking has failed and the program is incorrect. For that we introduce a context Γ that holds all currently known variables and their types.

$\vdash$ denotes that based on the left side, the right side is true. The horizontal line splits the proposition and the consequence.

If $x : \tau$ is in the typing context Γ, then it is true, that x is of type τ in the context Γ.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

If we have an extended context $\Gamma, x : \tau_1$ and in this context, it is true that $t : \tau_2$, then the type of the lambda form $\lambda x \,.\, t$ is $\tau_1 \to \tau_2$.

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash (\lambda x : \tau_1 \,.\, t) : \tau_1 \to \tau_2}$$

If we have an application of a lambda with type $\tau_1 \to \tau_2$ on a term of type τ_1, it is of type τ_2.

$$\frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1 t_2 : \tau_2}$$

2.3 Dependent type theory

Formally the extension of the simply-typed λ-calculus into a dependently typed system is rather trivial, in that we only relax the posed restrictions.

We extend types to be not only primitive types and function types. We allow any of the former *or* any term.

We write t, ρ to denote terms. We use ρ to denote that this term produces a type, and t for regular terms. This is only a convention used throughout this thesis and bears no syntactical meaning.

$$
\begin{array}{rll}
t, \rho := & x & \textit{Variable} \\
| & \lambda x : \rho \,.\, t & \textit{Abstraction} \\
| & t_1 t_2 & \textit{Application} \\
| & \tau & \textit{Type} \\
| & \mathcal{U} & \textit{Type universe} \\
| & \prod_{x:\rho_1} \rho_2 x & \textit{Dependent function space}
\end{array}
$$

If we want to reason about dependent types, we need to introduce a new concept, the dependent function space. One such space is the universe $\mathcal{U}$ type space, denoting every possible type. The dependent function space behaves similarly to normal lambdas, but we use a different syntax and the dependent lambda is now instantiated during type-checking.

$$\prod_{x:\rho_1} \rho_2 x \quad \textit{can be understood as} \quad \{\rho_2 x \mid \forall x \,.\, x : \rho_1\}$$

Generally, dependent function spaces are a set of all possible types that can be produced. The rule for this production is as follows. If ρ_2 is the lambda that produces the types and ρ_1 is the input of such space, then the dependent function space is the set of all types that are produced by taking every possible $x : \rho_1$ and applying ρ_2 on such x.[2]

$$\frac{\Gamma \vdash \rho_1 : \mathcal{U} \qquad \Gamma \vdash \rho_2 : \rho_1 \to \mathcal{U}}{\Gamma \vdash \prod_{x:\rho_1} \rho_2 x : \mathcal{U}}$$

Every type can be described by using this notation. If the type is not dependent, it simply means that it does not depend on x, and as such ρ_2 is a lambda that ignores its argument $\rho_2 := \lambda\_ \,.\, \rho_3$.[3]

When we want to instantiate a function with a dependent type, we pick the argument to the call and apply on it ρ_2.

$$\frac{\Gamma \vdash t_1 : \rho_1 \qquad \Gamma \vdash t_2 : \prod_{x:\rho_1} \rho_2 x}{\Gamma \vdash t_2 t_1 : \rho_2 t_1}$$

[2] In practice we do not generate such sets in the type-checking algorithm. We instead produce the resulting types on demand. For example this approach is used for template instantiation in C++.

[3] Convention throughout this thesis is that _ denotes an ignored variable, that can be never referred to. If it could be referred to, we would need the α-reduction that was previously skipped.

2.4 Bidirectional type-checking

In languages that do not support dependent types, the choice of type-checking algorithms is vast. One of the stronger algorithms is the Hindley-Milner type system [9, 10]. In such system the program does not need *any* type annotations. All type information can be inferred from the context of the caller and the callee.

By introducing dependent types we now require some annotations to appear. It may be argued that some type annotations are desirable from the language designer's perspective. However it is evident that type inference is still desirable, and that type annotations should only appear in places where there are ambiguities.[4]

The bidirectional type system provides a facility to provide a type inference algorithm in places where there are no ambiguities (like binding a local variable) with the price that functions must always be annotated. This type system is able to type-check even dependently typed programs and thus it was chosen to be the type system that was used as a basis for our own extensions.

We split the type annotation from simply-typed λ-calculus into two rules during type-checking, the $\Rightarrow$ *syntesis* rule and the $\Leftarrow$ *checking* rule.

If $x : \tau$ is in the type context Γ, then x synthesizes the type τ.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x \Rightarrow \tau}$$

Every term t can be checked to be of type τ, if it can be synthesized to be of type τ.

$$\frac{\Gamma \vdash t \Rightarrow \tau}{\Gamma \vdash t \Leftarrow \tau}$$

We can also annotate the expression to get from checking to synthesis.

$$\frac{\Gamma \vdash t \Leftarrow \tau}{\Gamma \vdash t : \tau \Rightarrow \tau}$$

If we have an extended context $\Gamma, x : \tau_1$ and in it, it is true that we can check $t \Leftarrow \tau_2$, then the type of the lambda $\lambda x \,.\, t$ checks $\tau_1 \to \tau_2$.

$$\frac{\Gamma, x : \tau_1 \vdash t \Leftarrow \tau_2}{\Gamma \vdash (\lambda x \,.\, t) \Leftarrow \tau_1 \to \tau_2}$$

If we have an application of lambda that synthesizes type $\tau_1 \to \tau_2$ on a term that checks to be of type τ_1, it synthesizes the type τ_2.

$$\frac{\Gamma \vdash t_1 \Rightarrow \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 \Leftarrow \tau_1}{\Gamma \vdash t_1 t_2 \Rightarrow \tau_2}$$

[4] There is a difference between ambiguities from the type-checking algorithm perspective and from the human perspective, however bidirectional type-checking strikes a decent balance by requiring only function type annotations.

2.5 Effect systems

The notion of encoding side effects into the language semantics has been proposed numerous times [11]. These systems only encode the side-effects as manual additions to the type. As such, they offer a practical addition but one that cannot be reasoned about, and does not conform to the notion of computation that is described in λ-calculus.

In *Notions of computation and monads*[12] Moggi describes a generalized approach to define side-effects as monadic operations, including nondeterminism, interactive input and output and exceptions. This extends the λ-calculus with the notion of side-effects and can be reasoned about.

Moggi describes two approaches to monadic side-effects. First, for reasoning about programming languages, Moggi introduces a metalanguage for a category, and treats monads as unary type-constructors. Second, for reasoning about programs, Moggi uses only one monad and the programming language itself for the term language.

We describe the extension of the Moggi's former approach by not encoding monadic operations using `bind` operation, as it is used in the case of the latter and Haskell, but rather, using the implicit binding between stateful monadic operations and operations without side-effects, need not to be wrapped explicitly. This approach is much more permissive in terms of composition and easier to grasp for beginners.

This thesis is not the first implementation of such systems. Prior work includes that of Leijen[13], Plotkin and Pretnar[14] and Brachthäuser et al.[15]. Our contribution is the extension of an effect system with a dependent type system.

As the combination of a dependent type system and an effect system is the subject of chapter 3, in this chapter we will only describe a basic effect system that is an extension of the simply-typed λ-calculus.

We extend the calculus by introducing two new operations the **raise** operation and the handler **with** operation.

The resumption operation can be used multiple times and thus this system is, in implementation, a multi-prompt delimited continuation[5] system as described by Felleisen[16].

Assume e is an effect, the terms are then defined as follows:

$t :=$	x	*Variable*
	$\lambda x : \tau \,.\, t$	*Abstraction*
	$t_1 t_2$	*Application*
	raise e t	*Effect signaling*
	with $e := \lambda r.\lambda x.t_1$ **in** t_2	*Effect handling*
	$()$	*Unit value*

In **raise** e t the t represents the argument passed to the handler. The

[5]Multi-prompt means that continuations can be called multiple times and delimited means that such continuations can return with a value.

r and x in the **with** handler represent the resumption operation and the argument that was passed to the handler $t_1[x := t]$.

The unit value $()$ is the value of nothing. Its type is $\top$, the empty type. It is used in places, where argument must be provided, but its value has no meaning.

With this framework we can show an example of how effect systems can encode side-effects. More specifically, let us take a look how state can be encoded. This is analogous to the `State` monad.

We assume two primitive effects, get and set, with following types:

$$\begin{aligned} \text{get} &: \top \to \mathbb{N} \\ \text{set} &: \mathbb{N} \to \top \end{aligned}$$

To simplify the example, we define abbreviations for the get and set handlers:

$$\begin{aligned} \text{get-handler} &:= \lambda s : \mathbb{N} \,.\, (r\ s)\ s \\ \text{set-handler} &:= \lambda \_ : \mathbb{N} \,.\, (r\ ())\ i \end{aligned}$$

Now a sample stateful computation can be written as:

$$\begin{aligned} \text{computation} := (&\textbf{with}\ \text{get} := \lambda r : (\mathbb{N} \to \mathbb{N} \to (\mathbb{N}, \mathbb{N})) \,.\, \lambda \_ : \top \,.\, \text{get-handler}\ \textbf{in} \\ &\textbf{with}\ \text{set} := \lambda r : (\top \to \mathbb{N} \to (\mathbb{N}, \mathbb{N})) \,.\, \lambda i : \mathbb{N} \,.\, \text{set-handler}\ \textbf{in} \\ &\quad \textbf{raise}\ \text{set}\ ((\textbf{raise}\ \text{get}\ ()) + 1) \\ &\quad f\ 1 \\ &\quad \textbf{raise}\ \text{set}\ ((\textbf{raise}\ \text{get}\ ()) + 1) \\ &\quad \lambda s : \mathbb{N} \,.\, (s, 1)) \end{aligned}$$

And run with the initial state of 0 as follows:

$$(\text{computation}\ 0)$$

This computation reduces to $(2, 1)$. The second element is the return value of the computation and the first is the final state.[6] We had to manually wrap the last line into the monad, which we do by wrapping it in a lambda manually, not unlike `return` in Haskell.

The computation translated into the State monad in Haskell (Appears in figure 2.1). We can directly translate the two handler cases, they correspond 1:1 with the `set` and `get` operations, the last statement is equivalent to `return`.[7]

[6] Note that f 1 is temporally dependent to happen after the first statement and before the second, even though there is no explicit passing of the state.

[7] Note that we are skipping over some details, usually all operations done on the monad are wrapped into the `State` type and not just tuples. Also this implementation is missing the definition of the monadic bind operation.

```haskell
type State s a = s -> (s, a)

return :: a -> State s a
return x s = (s, x)

get :: State s s
get s = (s,s)

set :: s -> State s ()
set x s = (x, ())

f :: State Int Int
f = do
  x <- get
  set (x + 1)
  lift $ f 1
  x <- get
  set (x + 1)
  return 1

runState f 0
```

Figure 2.1: Haskell State monad

To understand why encoding of effects in Haskell is not compositional, note that the function call f 1 is not manually lifted in the effect example and is in the Haskell example. Consider that f is a monadic operation on a different monad with different scope. Then the Haskell example would need to both define a monadic transformer and define an ad-hoc lifting operation between the two monads. In effect systems no such lifting is needed and effects can be transparently propagated through handlers. This is the biggest strength of effect systems. We can transparently compose impure computations.

Chapter 3: κ-calculus

In which we formally define the type system

In this chapter we introduce the κ-calculus. This is a small, formally tractable programming language that combines dependent types and effects. The κ-calculus system uses bidirectional typing rules. We follow the basic structure and notation from David Raymond Christiansen[17]. A system derived from the simply typed λ-calculus, that splits the classical rule $\Gamma \vdash t : \rho$ into two judgments: a *synthesis* judgment $\Gamma \vdash t \Rightarrow \rho$ (read as "t can synthesize a type ρ in the context Γ") and a *checking* judgment $\Gamma \vdash t \Leftarrow \rho$ (read as "t can be checked to have given type ρ in the context Γ").

In this thesis we extend this system with rules for effects. We also provide more synthesis rules for list head and tail syntheses, and a new kind of judgment for language constructs that cannot be handled using a syntax-driven type system. For example, in Kelp, it needs to be possible to check `[1 1 1]` as both a tuple and a three element list.

Let us define the formal syntactic system. Let us assume that every program is syntactically valid, and that no variables are of the same name. This is always possible using renaming or, for example, the technique known as de Bruijn indices[18].

3.1 Terms

The terms of the κ-calculus are those of the λ-calculus with effects and dependent types, extended with collections and conditionals.

$t :=$	$x, y, \ldots$	*Variables*
$\mid$	$e_1, e_2, \ldots$	*Effects*
$\mid$	$t_1 t_2$	*Application*
$\mid$	$\lambda x\ .\ t$	*Abstraction*
$\mid$	$()$	*Unit value*
$\mid$	**if** t_1 **then** t_2 **else** t_3	*Conditional evaluation*
$\mid$	$[t_1\ t_2\ t_3\ \ldots]$	*Collections*
$\mid$	$t\ :\ \rho$	*Type annotation*
$\mid$	**raise** e t	*Effect signaling*
$\mid$	**with** $e := \lambda r.\lambda x.t_1$ **in** t_2	*Effect handling*
$\mid$	ρ	*Type term*

Types follow an infinite hierarchy of $t : \tau : T_1 : T_2 : \ldots$. The $\mathcal{U}$ type

universe is introduced as a type, that describes every possible constructible type. It is needed for specifying types of dependent functions.

raise e t is the effect raising construct described in section 2.5. e specifies the effect raised, and t is a term describing its argument. As such effect raising looks like calling a regular function.

with $e := \lambda r.\lambda x.t_1$ **in** t_2 is an effect handler. e is the effect being handled. $\lambda r.\lambda x.t_1$ is the effect body. When the effect is raised, this function is applied. The variable r is bound to the resumption operation of the effect[1]. The variable x is bound to the argument provided to **raise**. The term t_2 is the handler body, from which the effects will be handled. The resumption has the signature of $\rho_1 \to \rho_2$, thus resumption can be invoked multiple times and the result of a handler is the result of applying the effect body recursively on the handler body.

3.2 Effects

$e :=$	$(\textbf{Effect}\ \rho_1\rho_2)$	*Effect construction*
$\varepsilon :=$	$(e_1, e_2, \ldots)$	*Effect collection*
$\mid$	$\varepsilon_1 \boxplus \varepsilon_2$	*Effect collection composition*
$\mid$	$\varepsilon_1 \boxminus \varepsilon_2$	*Effect collection subtraction*

We denote an effect collection with ε and E. E is reserved for the current effect context and ε is used in function type signatures.

3.3 Types

The types of κ-calculus are:

[1]The return to the place where the effect was raised.

$$
\begin{array}{lll}
\rho := & t & \textit{Term} \\
| & \top & \textit{Unit type} \\
| & \bot & \textit{Empty type} \\
| & \tau & \textit{Type literal} \\
| & \rho_1 \to \rho_2 & \textit{Function type} \\
| & \rho_1 \to^{!e} \rho_2 & \textit{Effect type} \\
| & \rho_1 \to \rho_2 \uparrow^{!} \varepsilon & \textit{Function type with an effect} \\
| & \mathcal{U} & \textit{Type universe} \\
| & \prod_{x \Leftarrow \rho_1} \rho_2 x & \textit{Dependent function space} \\
| & (\rho_1\ \rho_2 \cdots \rho_n) & \textit{Tuple} \\
| & (\textbf{C-List}\ \rho\ n) & \textit{List with known length} \\
| & (\textbf{List}\ \rho) & \textit{List}
\end{array}
$$

The rules for t and ρ can both contain types and terms. In this thesis we use ρ instead of t for a term, to signal that it will be used as a type. As discussed before, this is simply to ease the reading and bears no syntactic meaning in the language itself.

The *unit* type represents a one possible value type, sometimes in other languages like C, called `void`, the value represented is $()$. However unlike C, this type is not "incomplete" and it is allowed to be put into structures and bound.

The *empty* type represents an impossible state. In C it would be represented in a function as `__attribute__((noreturn))`. Or type of anything in an `if(false)` block.

The effect type $\rho_1 \to^{!e} \rho_2$ denotes the type when raising an effect e. Where ρ_1 is the accepted argument type and ρ_2 is the return type. This behaves the same as the function type. The difference is that each effect type contains the unique identifier e of its definition and as such is unique and not equal to any other effect.

The function type $\rho_1 \to \rho_2$ is now optionally extended with ε to denote the function's effects. As such the full notation for function types is $\rho_1 \to \rho_2\ !\ \varepsilon$ optionally with $!\ \varepsilon$ omitted for brevity.

The dependent function space $\prod_{x \Leftarrow \rho_1} \rho_2 x$ denotes the type of a generic function. The signature denotes that for each x that can be checked to be of type ρ_1, we can create a type that is the result of applying ρ_2 term to x. Such example is a function that returns a type. Types that are not dependent, are also a dependent function space, but the result ρ_2 does not depend on x.

Equality of types is defined with structural equivalence. Types can also be sub-typed, that is, one type ρ_1 is a sub-type of another type ρ_2, if in every place where term of type ρ_2 can be placed, a term of type ρ_1 can be placed as well. We denote $\rho_1 :> \rho_2$ as the sub-typing relation.

3.4 Typing rules

Recall that in simply typed λ-calculus we have a type annotation $x : \rho$. In bidirectional type-checking, we split this rule into two, the aforementioned *synthesis* and *checking* rules. The type annotation is the simplest example of the *synthesis* rule.

Now we begin to diverge from Christiansen's article, as effects now need to be considered when applying a function. Effects are produced as a part of the raise operation and composed with application. Expressions are not allowed unless the effect their application produces is in the effect context. Right now the only change is the addition of E, the effect context.

Γ denotes the context of defined variables and their types. The type system is thus specified using judgments of the form $\Gamma, E \vdash t \Rightarrow \rho$ and $\Gamma, E \vdash t \Leftarrow \rho$. Typing of variables is as before:

$$\text{Var}\frac{(x : \rho) \in \Gamma}{\Gamma, E \vdash v \Rightarrow \rho}$$

Checking and type annotations also behave as before. Every term t can be checked to be of type ρ if it can be synthesized to be of type ρ.

$$\text{Check}\frac{\Gamma, E \vdash t \Rightarrow \rho}{\Gamma, E \vdash t \Leftarrow \rho}$$

If term t can be checked to be of type ρ, then the annotation $t : \rho$ synthesized type ρ. This is the inverse operation of "Check". It is not a bijection however, we require an explicit annotation to switch from checking to synthesizing. Therefore synthesis is a stronger requirement.

$$\text{Annotate}\frac{\Gamma, E \vdash t \Leftarrow \tau}{\Gamma, E \vdash t : \tau \Rightarrow \tau}$$

If we have an extended context $\Gamma \boxplus x : \rho_1$[2] and in it, it is true that if we can check $t \Leftarrow \rho_2$, then the type of the lambda $\lambda x \,.\, t$ checks $\rho_1 \to \rho_2$.

$$\text{Argument body check}\frac{\Gamma \boxplus \{x : \rho_1\}, E \vdash t \Leftarrow \rho_2}{\Gamma, E \vdash \lambda x \,.\, t \Leftarrow \rho_1 \to \rho_2}$$

Effects can be raised and effects can be handled. Raising an effect is an operation that introduces side-effects, all operations not containing any unhandled effects are side-effect-free. All functions either have a handler for each effect, or the effect is a part of their signature, and such function is not side-effect-free. However if the main entry point of the program does not have any unhandled side-effects, the whole program is side-effect-free.

This allows for the existence of programs with local self-contained side effects. Such as ones containing a mutable state, which are still side-effect-free. Such programs are pure, meaning that for the same input, the same output is always produced. Note that this is a stronger guarantee than that

[2]We have previously used $\Gamma, x : \tau$ to denote Γ extension. With the introduction of the E effect context this is no longer practical. We use $\boxplus$ to denote the extension from now on.

of Haskell's purity. Haskell functions are considered pure, but still can call `fail` which aborts the execution. This is not possible in our type system. Functions can only crash due to out-of-memory events and not terminate.

$$\text{Effect raising}\frac{\Gamma, E \vdash t_1 \Rightarrow e \qquad \Gamma, E \vdash t_2 \Leftarrow \rho_1 \qquad e : \rho_1 \to^! \rho_2 \in E}{\Gamma, E \vdash \textbf{raise}\ t_1\ t_2 \Rightarrow \rho_2}$$

When raising an effect, the effect e needs to be defined in the effect context E. This also determines the type of its argument and result. Thus effect context E holds not only allowed effects, but also their type signatures.

Effect handling has an interesting property. The result of the handler is the type of t_1. If no e effects are raised, then it must be true that $\rho_3 :> \rho_4$. The resumption operation has the type $\rho_2 \to \rho_3$ where ρ_2 is the effect's raising result type, and ρ_3 is the result type of the t_2 computation.

$$\text{Effect handling}\frac{\begin{array}{c}\rho_3 <: \rho_4 \\ \Gamma, E \boxplus e : \rho_1 \to^! \rho_2 \vdash t_2 \Leftarrow \rho_3 \\ \Gamma, E \vdash t_1 \Rightarrow (\rho_2 \to \rho_3) \to \rho_1 \to \rho_4\end{array}}{\Gamma, E \vdash \textbf{with}\ e := t_1\ \textbf{in}\ t_2 \Rightarrow \rho_4}$$

Effects do not participate in expressions, they only do appear in function types, as functions and handlers are the only expressions that operate on effect context. This does pose some limitations when generic effects are required, as effects cannot be simply retrieved from expression type.

Note that we require the function type to be synthesizable here. This differs from the Hindley-Milner type systems and can roughly be translated as "every function needs to be annotated with a type". Nevertheless, this is true for most programming languages.

Application behaves the same as in section 2.4. The only difference is that now we require that ε is in the effect context at the point of application.

$$\text{Application}\frac{\Gamma, E \vdash t_1 \Rightarrow \rho_1 \to \rho_2 \uparrow^! \varepsilon \qquad \Gamma, E \vdash t_2 \Leftarrow \rho_1 \qquad \varepsilon \subseteq E}{\Gamma, E \vdash t_1 t_2 \Rightarrow \rho_2}$$

Wrapping a term producing an effect in a lambda shadows it; effects only appear as a part of the lambda type, as well as in the lambda application. This means that it is side-effect-free to define a side-effect producing lambda.

$$\text{Lambda effect encapsulation}\frac{\Gamma \boxplus \{x : \rho_1\}, \varepsilon \vdash t \Rightarrow \rho_2}{\Gamma, \emptyset \vdash \lambda x.t \Rightarrow \rho_1 \to \rho_2 \uparrow^! \varepsilon}$$

3.4.1 Dependent types

The rules for dependent types are the same as described in section 2.3, but now we consider the bidirectional type-checking algorithm and thus we need

to refine the rules to distinguish between checking and synthesis. We also add the context E even though it does not actively participate in any rules.

Dependent type is an image of a function $\rho_2 : \rho_1 \to \mathcal{U}$, where the argument of type ρ_1 can be any term, including a type. We can refine the construction rule by requiring synthesis only for ρ_2. The refined rule is isomorphic to the lambda application rule.

$$\text{Dependent space construction}\frac{\Gamma, E \vdash \rho_1 \Leftarrow \mathcal{U} \qquad \Gamma, E \vdash \rho_2 \Rightarrow \rho_1 \to \mathcal{U}}{\Gamma, E \vdash \prod_{x \Leftarrow \rho_1} \rho_2 x \Leftarrow \mathcal{U}}$$

If a term t_1 synthesizes dependent type $\prod_{x \Leftarrow \rho_1} \rho_2 x$, and t_2 checks to be ρ_1, then the application of $t_1 t_2$ synthesizes the type $\rho_2 t_2$.

$$\text{Dependent space instantiation}\frac{\Gamma, E \vdash t_1 \Rightarrow \prod_{x \Leftarrow \rho_1} \rho_2 x \qquad \Gamma, E \vdash t_2 \Leftarrow \rho_1}{\Gamma, E \vdash t_1 t_2 \Rightarrow \rho_2 t_2}$$

3.4.2 Sub-typing

It is beneficial to define a hierarchy between types. For example an $(\mathbf{Int}\ 32) <: \mathbf{Integer}$. The bounded 2^{32} integer type is sub-type of the big-number integer, i.e. in every place where the big integer is required, a bounded integer is allowed.

We provide only a pair of rules for sub-typing, as it is not a subject of this thesis. This sub-typing is partial ordering as shown by Pfenning[19].

If we have a sub-typing of $\rho_1 :> \rho_2$ and the term t synthesizes ρ_1, then we can check t to be of type ρ_2.

$$\text{Sub-typing}\frac{\rho_1 :> \rho_2 \qquad \Gamma, E \vdash t \Rightarrow \rho_1}{\Gamma, E \vdash t \Leftarrow \rho_2}$$

$$\text{Function sub-typing}\frac{\rho_1 :> \rho_2 \qquad \rho_3 <: \rho_4}{\rho_1 \to \rho_3 <: \rho_2 \to \rho_4}$$

Dependent types are also a part of sub-typing. If a type term ρ is in the dependent function space, it is the sub-type of such space.

$$\text{Dependent sub-typing}\frac{\rho \in \prod_{x \Leftarrow \rho_1} \rho_2 x}{\rho <: \prod_{x \Leftarrow \rho_1} \rho_2 x}$$

As sub-typing is a partial ordering, equality holds if both types are sub-types of each other.

$$\text{Sub-typing and equality}\frac{\begin{array}{c}\rho_1 <: \rho_2 \\ \rho_2 <: \rho_1\end{array}}{\rho_1 = \rho_2}$$

3.5 Collections

In Kelp the expression `[1 1 1 1]` can be either a list of integers, a compile-time known list of four integers, or a tuple of four integers. As such it is generally unknown if such expression synthesizes one of the types that it will later be checked against. To address this, we introduce the notion of a **Collection**. **Collection** is a union of the **List**, **C-List** and tuple dependent spaces. Collection constraints allow us to handle collection-related languages constructs that are not syntax-driven.

We introduce a new pair of judgments $\Gamma, E \vdash_c t \Rightarrow (\textbf{Collection}\ \rho\ n)$ and $\Gamma, E \vdash_c t \Leftarrow (\textbf{Collection}\ \rho\ n)$. It allows us to perform judgments about collections without the possibility of collections leaking to the final program.

If t can synthesize $(\textbf{Collection}\ \rho\ n)$ then t can be checked to be of any of the subsequent concrete types.

$$\frac{\Gamma, E \vdash_c t \Rightarrow (\textbf{Collection}\ \rho\ n)}{\Gamma, E \vdash t \Leftarrow (\textbf{List}\ \rho) \qquad \Gamma, E \vdash t \Leftarrow (\textbf{C-List}\ \rho\ n) \qquad \Gamma, E \vdash t \Leftarrow \overline{\rho}^n}$$

We define a sub-typing hierarchy of collection types as follows:

$$\begin{aligned} \forall i \in \mathbb{N} : (\textbf{List}\ \rho) &:> (\textbf{C-List}\ \rho\ i) \\ \forall i \in \mathbb{N} : (\textbf{C-List}\ \rho\ i) &:> (\textbf{Collection}\ \rho\ i) \\ (\overline{\rho}^n) = (\rho, \rho, \ldots \rho) &:> (\textbf{Collection}\ \rho\ n) \end{aligned}$$

In type-checking we might encounter a situation in which we either synthesize the type of the first element or synthesize the type of the second element. However we cannot at this time provide the concrete type for this expression, as that may depend on further context.

We introduce two new rules, which we either synthesize the head or the tail of the expression. These formalize an algorithm for type-checking programs working with collections where only partial sections of the type are known.

Head tail collection type synthesis
$$\frac{\begin{array}{c}\Gamma, E \vdash (\textbf{cons}\ t_1\ t_2) \\ \Gamma, E \vdash t_1 \Rightarrow \rho_1 \\ \Gamma, E \vdash_c t_2 \Leftarrow (\textbf{Collection}\ \rho_1\ n)\end{array}}{\Gamma \vdash_c (\textbf{cons}\ t_1\ t_2) \Rightarrow (\textbf{Collection}\ \rho_1\ n+1)}$$

Tail head collection type synthesis
$$\frac{\begin{array}{c}\Gamma, E \vdash (\textbf{cons}\ t_1\ t_2) \\ \Gamma, E \vdash_c t_2 \Rightarrow (\textbf{Collection}\ \rho_1\ n) \\ \Gamma, E \vdash_c t_1 \Leftarrow \rho_1\end{array}}{\Gamma \vdash_c t \Rightarrow (\textbf{Collection}\ \rho_1\ n+1)}$$

It is important to show that if both of the rules are applicable, the resulting types are the same.

Lemma 1. *If the head tail (HT) synthesis produces a type, and the tail head (TH) synthesis produces a type, the produced types are equal.*

Proof. Let us define that the synthesized types are HT $\Rightarrow \rho_{HT}$ and TH $\Rightarrow \rho_{TH}$. WLOG let us say that $\rho_{HT} <: \rho_{TH}$ while $\rho_{TH} \not<: \rho_{HT}$. If neither $\rho_{HT} <: \rho_{TH}$ or $\rho_{HT} :> \rho_{TH}$ are true, then the check rule must have failed, and because of that, there exists a sub-typing relation between ρ_{TH} and ρ_{HT}.

If $\rho_{HT} <: \rho_{TH}$, then one of the synthesized types is ρ_{TH}, as neither TH or HT syntheses can produce a new type. That means that there is a term of type ρ_{TH} that cannot be checked to be ρ_{HT}. That means that the HT must have failed. And we a have a contradiction. □

If an expression is a tuple, we can either synthesize a collection or only a tuple, depending on whether the types of tuple elements have a common super-type. This is exactly the aforementioned head tail/tail head synthesis.

$$\text{Collection specialization}\frac{\Gamma, E \vdash \overline{t_i \Rightarrow \rho_i}^n \qquad \forall \rho \in \overline{\rho_i}^n \; \exists \rho_0 : \rho <: \rho_0}{\Gamma, E \vdash_c [\overline{t_i}^n] \Rightarrow (\textbf{Collection } \rho_0 \; n)}$$

$$\text{Tuple specialization}\frac{\Gamma, E \vdash \overline{t_i \Rightarrow \rho_i}^n \qquad \forall \rho \in \overline{\rho_i}^n \quad \nexists \rho_0 : \rho <: \rho_0}{\Gamma, E \vdash [\overline{t_i}^n] \Rightarrow (\overline{\rho_i}^n)}$$

3.6 Constraints

Now that collections have shown how our system can deal with not-syntax-driven language constructs, it is useful to provide a generalization of such extended type-checking. We introduce constraints as the generalization of collections, types that cannot be created but which can be checked in the special $\vdash_c$ judgments.

Constraints are similar to dependent types. Unlike dependent types, constraints cannot be constructed. They lack the $\rho_2 : \rho_1 \to \mathcal{U}$ function. Rather they represent the union of such spaces. As such they are not spaces themselves. As they are not a part of the type system in the general sense, and at the end of type-checking, no program contains any constraints, we introduce a new kind of judgment $\vdash_c$, which behaves similar to the regular judgment but allows for constraints to be a part of the judgments.

Let us define $\mathbb{C}$ as the mapping between strings $\mathcal{K}$, aliases for type constructors, and in language constructs f. This allows us to introduce aliases for complex expressions. We include the strings $\mathcal{K}$ in the definition of terms.

Now we can formalize the aforementioned **Collection** as the union of the three concrete types inside $\mathbb{C}$.

$$f(\rho, n) := \{(\textbf{List } \rho), (\textbf{C-List } \rho \; n), \overline{\rho}^n\}$$
$$\mathbb{C} := \mathbb{C} \cup \textbf{Collection} \rightarrowtail f$$

We can construct constraints by first creating a constructing function f. Then the type is constructed as the application of f on the parameters $t_1, t_2, \ldots$.

$$\text{Constraint construction}\frac{\mathcal{K} \rightarrowtail f \in \mathbb{C} \qquad \Gamma, E \vdash_c t \Rightarrow (\mathcal{K}\ t_1\ t_2\ \ldots) \qquad \rho \in (f\ t_1\ t_2\ \ldots)}{\vdash t \Leftarrow \rho}$$

If the term t synthesizes the constraint C_{ρ_1,ρ_2}, then it checks to be of both ρ_1 and ρ_2.

$$\text{Constraint instantiation}\frac{\Gamma, E \vdash_c t \Rightarrow C_{\rho_1,\rho_2}}{\Gamma, E \vdash t \Leftarrow \rho_1 \qquad \Gamma, E \vdash t \Leftarrow \rho_2}$$

Implicitly we assume, that every judgment from the normal checking is available in the constraint checking. In logic theory we would say that the constraint judgment theory is a conservative extension of the aforementioned type theory.

$$\text{Synthesis extension}\frac{\Gamma, E \vdash t \Rightarrow \rho}{\Gamma, E \vdash_c t \Rightarrow \rho} \qquad \text{Check extension}\frac{\Gamma, E \vdash t \Leftarrow \rho}{\Gamma, E \vdash_c t \Leftarrow \rho}$$

Remark 1. *No well typed program contains constraints at the end of type-checking. All constraints are either discarded or checked into proper types.*

Let us assume that no syntax can name the constraint, that is, terms of such constraint are possible, but the constraint itself cannot be put in a place of a type in the program. Assume also that each program is only contained in a body of a function with declared return type.

- *This implies that every function can only accept proper types as its arguments, and only return a proper type.*
- *This implies that every statement that is of a constraint type is inside a body, and never crosses into another function, as at that point it is replaced by the required constraint specialization into a proper type.*
- *And finally, each statement either appears in a function call – then it is constraint specialized, in a return statement – then it is constraint specialized by the return type of the containing function, or it appears in a body and it is ignored, and as such can be ignored during compilation, and only its side effects can be considered.*

Chapter 4: Kelp

In which we introduce the language

The Kelp language is based on the traditional Lisp syntax. Unlike most traditional Lisps it is statically typed in nature and does not feature a macro system. Instead it is planned to include a compile-time reflection system symbiotic with the κ-calculus type system.

Kelp variables are lexically scoped and immutable. Any mutable state or side-effect can only be achieved through the effect system. Code is organized into records which also act like namespaces, and each file implicitly creates such record. As such declarations are provided not as a special form, rather as a field value.

The only distinction of compile-time and runtime expressions is the presence of effects. As such, the runtime provides two top level handlers, the compile-time handler (for compile-time side-effects like warnings and errors) and a runtime handler. The compile-time handler is used every time an expression is tried to be compile-time evaluated. If the effect is not handled even by the top-level handler, the interpreter backs out, as such expression requires runtime evaluation. The runtime handler only wraps the main function.

The following example (figure 4.1) introduces Kelp. It shows a basic program that prints the addition of two numbers.

```
;; package namespace import
:core (import "core")

;; function declaration and definition
:add (lambda
    [a :: Integer ;; argument type declaration
     b :: Integer]
    :-> Integer   ;; return type declaration
    :! []         ;; effect declaration
    (+ a b))      ;; last statement is the return value

:main (lambda []
    ;; namespaced symbols are accessed with :
    :! [core:io:Stdout-Write]
    (core:io:print (integer->string (add 1 2))))
```

Figure 4.1: Example Kelp code.

The form `[a b c]` is a shorthand for `(list a b c)` and is used instead of the usual Lisp `` `(a ,b c) ``.

4.1 Calculating the Fibonacci numbers

The following program (See figure 4.2) calculates the 10th Fibonacci number. This algorithm is a naïve $\mathcal{O}(n^2)$ implementation, however it does currently produce the correct result when run through the Kelp interpreter.

```
:core (import "core")

:fibonacci
(lambda [n :: Integer]
  :-> Integer
  (if (= n 0)
      0
      (if (= n 1)
          1
          (+ (fibonacci (- n 1))
             (fibonacci (- n 2))))))

:main
(lambda []
  :! [core:io:Stdout-Write]
  (let [n :: Integer 10]
    (core:io:print "The ")
    (core:io:print (integer->string n))
    (core:io:print " fibonacci number is ")
    (core:io:print (integer->string (fibonacci n)))))
```

Figure 4.2: Kelp code calculating the 10th Fibonacci number.

4.2 FizzBuzz

A more complex example is the implementation of the FizzBuzz children's game. In the program, we count from 1 to n and every time the number is divisible by 3 we yell `Fizz`, and when it is divisible by five we yell `Buzz`. If the number is divisible both by 3 and 5, we yell `FizzBuzz`, otherwise we just print the number.

The following example (See figure 4.3) is more complex than the previous one, as we have to implement our own `map` function. The function is not generalized. The generalization is left for future work. `match` is a pattern matching built-in that uses the structure of the provided expression to choose a path.

```
:core (import "core")

:map
(lambda [f :: (-> [Integer] [] :! [core:io:Stdout-Write])
         lst :: (List Integer)]
  :! [core:io:Stdout-Write]
  :-> []
  (match lst
    [] []
    [head :: Integer . tail :: (List Integer)]
    (begin
       (f head)
       (map f tail))))

:fizzbuzz
(lambda [n :: Integer]
  :-> []
  :! [core:io:Stdout-Write]
  (if (= (remainder n 15) 0)
      (core:io:print "FizzBuzz")
      (if (= (remainder n 3) 0)
          (core:io:print "Fizz")
          (if (= (remainder n 5) 0)
              (core:io:print "Buzz")
              (core:io:print (integer->string n)))))
  (core:io:print "\n"))

:main
(lambda []
  :! [core:io:Stdout-Write]
  :-> []
  (map fizzbuzz (range 1 100)))
```

Figure 4.3: Kelp FizzBuzz.

4.3 Handler

Until now we have only seen the invocations of the top-level runtime handler. We can also define our own handlers. In this example (See figure 4.4.) we override the `print` function. We redirect the standard output into the standard error output.

```
:core (import "core")

:main
(lambda []
  :! []
  :-> [core:io:Stderr-Write]
    (with [core:io:Stdout-Write
      (lambda [r s]  ;; r is the resumption, s is the argument
        (core:io:print-err s) ;; print the into stderr
        (r))] ;; resume the operation
    (core:io:print "Error")))
```

Figure 4.4: Kelp handler.

4.4 Variables

As an optimization technique, some functional languages allow a for temporarily mutable state (Clojure's transient data structures for example.[20, 21]). In Kelp we allow interior mutability with effects. This theoretically still is a pure operation, and can be modeled using monads (see section 2.5). But for the compiler this can be lowered as a simple read/write operation on memory. If other properties are desired, such as thread synchronization, they can be achieved easily without changing the syntax.

```
:core (import "core")

:main
(lambda []
  :! []
  :-> [core:io:Stdout-Write]
  ;; with-var is syntactic sugar over with
  (with-var [counter 0]
    ;; Modify the value by applying the function
    ;; on the old value
    (swap! counter (+ 1))
    ;; Set the value of counter to the @counter + 1
    ;; @x retrieves the value of a variable
    (set! counter (+ 1 @counter))
    ;; Prints 2
    (core:io:print (integer->string @counter))))
```

Figure 4.5: Kelp Variables.

4.5 Architectural specifics

The current interpreter uses heap-allocated reference-counted stack frames. As the control flow is non-linear, the runtime cannot free stack frames on return. As such the reference-counting approach was chosen as it is perhaps slow, yet it is correct and easy to implement. It is planned to use a more sophisticated algorithm for expressions containing effects such as the proposed *Type directed compilation of row-typed algebraic effects* by Daan Leijen[6].

4.6 Current limitations

The current Kelp implementation is limited. It does not provide a way to create and instantiate generic functions, the code can return a new function at compile-time that is parameterized by the types, however it is not done implicitly and requires two invocations.

The type system is the aforementioned dependent type system with effects, but the algorithm of type-checking currently differs. In the implementation the system is ad-hoc and does require type annotations for each variable. This is to be solved in future upgrades by moving to the proposed type-checking algorithm.

It is possible to lower Kelp code to native machine code, but currently the code is first semantically analyzed, which includes interpreting compile-time expressions, and later run through the same tree-walking interpreter at runtime. The stages are separate and the latter can easily be lowered into a better representation.

Chapter 5: Conclusions

In which we conclude our findings

We have combined a dependent type system with effects. To achieve this, we have used a bidirectional type system. We have modified the type system to allow for dependent types and effects. We have defined the combination of the two.

The bidirectional type-system allowed us to extend the type-checking algorithm with new inference rules for list type synthesis and checking. We have proven that these rules about inference reach the same inferred type, if both inferences succeed.

We introduced a brand new type of judgment which we symbolized as $\vdash_c$. This judgment extends the type-checking and inference algorithm with new kind of pseudo-types we named constraints. These constraints allow us to reason about non syntax-driven language constructs, such as tuples and lists. We have shown that such constructs never escape into the final program, and that we can type-check more programs without the requirement of type annotations.

We have shown a prototype language named Kelp which implements these systems. We have shown examples of how this language would use these systems to aid general programming.

5.1 Future work

The next problems in this domain are:

- How would the system look, if we extended using Quantitative Type Theory[22]?
- Currently the system no notion of non-termination when type-checking. Could the system be extended to be both mathematically sound and strongly normalizing?
- Can such system be effectively compiled to platforms with no support for multi-prompt delimited continuation using translation to tail calls?

Bibliography

1. Pierce, Benjamin Crawford; Turner, David N. Local type inference. *ACM Trans. Program. Lang. Syst.* 2000, vol. 22, no. 1, pp. 1–44. ISSN 0164-0925. Available from DOI: 10.1145/345099.345100.
2. Petříček, Tomáš. What we talk about when we talk about monads. *The Art, Science, and Engineering of Programming*. 2018.
3. Jones, Mark P.; Duponcheel, Luc. *Composing monads*. 1993. Research Report YALEU/DCS/RR-1004. Yale University. Available also from: http://web.cecs.pdx.edu/~mpj/pubs/RR-1004.pdf.
4. Liang, Sheng; Hudak, Paul; Jones, Mark. Monad transformers and modular interpreters. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Francisco, California, USA: Association for Computing Machinery, 1995, pp. 333–343. POPL '95. ISBN 0897916921. Available from DOI: 10.1145/199448.199528.
5. Hickey, Rich. Simple Made Easy. In: Strange Loop Conference, 2011. Available also from: https://www.youtube.com/watch?v=SxdOUGdseq4.
6. Leijen, Daan. Type directed compilation of row-typed algebraic effects. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. Paris, France: Association for Computing Machinery, 2017, pp. 486–499. POPL '17. ISBN 9781450346603. Available from DOI: 10.1145/3009837.3009872.
7. Church, Alonzo. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*. 1936, vol. 58, p. 345. Available also from: https://api.semanticscholar.org/CorpusID:14181275.
8. Turing, Alan Mathison. Computability and λ-definability. *Journal of Symbolic Logic*. 1937, vol. 2, no. 4, pp. 153–163. Available from DOI: 10.2307/2268280.
9. Hindley, Roger. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society* [online]. 1969, vol. 146, pp. 29–60 [visited on 2024-04-19]. ISSN 00029947. Available from: http://www.jstor.org/stable/1995158.
10. Milner, Robin. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*. 1978, vol. 17, no. 3, pp. 348–375. ISSN 0022-0000. Available from DOI: https://doi.org/10.1016/0022-0000(78)90014-4.

11. Lucassen, John M.; Gifford, David K. Polymorphic effect systems. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Diego, California, USA: Association for Computing Machinery, 1988, pp. 47–57. POPL '88. ISBN 0897912527. Available from DOI: 10.1145/73560.73564.

12. Moggi, Eugenio. Notions of computation and monads. *Information and Computation*. 1991, vol. 93, no. 1, pp. 55–92. ISSN 0890-5401. Available from DOI: https://doi.org/10.1016/0890-5401(91)90052-4. Selections from 1989 IEEE Symposium on Logic in Computer Science.

13. Leijen, Daan. Koka: Programming with Row Polymorphic Effect Types. *Electronic Proceedings in Theoretical Computer Science*. 2014, vol. 153, pp. 100–126. ISSN 2075-2180. Available from DOI: 10.4204/eptcs.153.8.

14. Plotkin, Gordon; Pretnar, Matija. Handlers of Algebraic Effects. In: Castagna, Giuseppe (ed.). *Programming Languages and Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 80–94. ISBN 978-3-642-00590-9.

15. Brachthäuser, Jonathan Immanuel; Schuster, Philipp; Ostermann, Klaus. Effect handlers for the masses. *Proc. ACM Program. Lang.* 2018, vol. 2, no. OOPSLA. Available from DOI: 10.1145/3276481.

16. Felleisen, Mattias. The theory and practice of first-class prompts. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Diego, California, USA: Association for Computing Machinery, 1988, pp. 180–190. POPL '88. ISBN 0897912527. Available from DOI: 10.1145/73560.73576.

17. Christiansen, David Raymond. Bidirectional Typing Rules: A Tutorial. 2013. Available also from: https://davidchristiansen.dk/tutorials/bidirectional.pdf.

18. de Bruijn, Nicolaas Govert. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*. 1972, vol. 75, no. 5, pp. 381–392. ISSN 1385-7258. Available from DOI: https://doi.org/10.1016/1385-7258(72)90034-0.

19. Pfenning, Frank. *Lecture Notes on Bidirectional Type Checking*. 2004. Available also from: https://www.cs.cmu.edu/~fp/courses/15312-f04/handouts/15-bidirectional.pdf.

20. Hickey, Rich. A history of Clojure. *Proc. ACM Program. Lang.* 2020, vol. 4, no. HOPL. Available from DOI: 10.1145/3386321.

21. Bagwell, Philip Sidney; Rompf, Tiark. RRB-Trees: Efficient Immutable Vectors. In: 2011. Available also from: https://api.semanticscholar.org/CorpusID:15763144.

22. Atkey, Robert. The Syntax and Semantics of Quantitative Type Theory. In: *LICS '18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, July 9–12, 2018, Oxford, United Kingdom*. 2018. Available from DOI: 10.1145/3209108.3209189.