CHALMERS

Monadic Intermediate Language for Modular and Generic Compilers

Master of Science Thesis in the Programme:
Computer Science – Algorithms, Languages and Logic

DMYTRO LYPAI

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, May 2016

Monadic Intermediate Language for Modular and Generic Compilers

DMYTRO LYPAI

Examiner: JOSEF SVENNINGSSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Typeset in Palatino and Source Code Pro using Pandoc and XeLaTeX.

Department of Computer Science and Engineering
Göteborg, Sweden, May 2016

Abstract

Monadic Intermediate Language for Modular and Generic Compilers

DMYTRO LYPAI

Department of Computer Science and Engineering

Chalmers University of Technology and University of Gothenburg

Most modern compilers perform a number of sophisticated program analyses and transformations, with their correctness and safety being of extreme importance. Intermediate representations used in compilers play a crucial role in defining how these procedures will be performed: what is possible to express and a level of effort required. A variety of computational effects present in modern programming languages imposes additional challenges on compiler writers.

This thesis starts with a brief introduction to monads and monad transformers and their relation to programming languages. A survey of the existing intermediate languages based on monads as well as highlights of the current developments in programming with effects are presented.

It continues with the main contribution of this work: Monadic Intermediate Language (MIL), which is a statically and explicitly typed functional language, which uses monads and monad transformers to express different computational effects and their combinations. To evaluate the designed intermediate language, two source languages representing two major programming paradigms: object-oriented and functional, have been designed and compilers targeting MIL have been implemented. Strengths and weaknesses of MIL uncovered during the implementation of the source programming languages are described in detail.

We conclude with describing code transformations and optimisations implemented for MIL, including very general effect-independent algebraic equivalences, such as, for example, monad laws, as well as a number of effect-dependent transformations. Relations to some of the well-known state-of-the-art optimisation techniques are outlined.

The ideas described in this thesis are implemented as three separate Haskell packages (`mil`, `funlang` and `oolang`), which are open source and freely available online.

Keywords: *compilers, intermediate representations, intermediate languages, computational effects, monads, monad transformers*

Acknowledgements

I owe a big thanks to the Swedish Institute, who made my studies in Sweden possible, effectively helping me to make one of the biggest changes in my life.

I am really grateful to my supervisor Josef for the initial idea of this thesis, for all his guidance along the way, for challenging me to always make this work better.

I would like to thank many of my friends and colleagues for their general support and for their help with finding an opponent. I am very thankful to Henrik Edström for doing the opposition.

And last but not least, I would like to thank my wife Liza, who believes in me much more than I do.

Contents

Chapter 1

Introduction

The vast majority of modern compilers follow more or less the same structure known for many years, which is called *compiler pipeline*. Compiler pipeline is a chain of phases, which together form the compilation process. They can be divided into two big parts: *analysis* and *synthesis*. Analysis (often referred to as the *front end* of the compiler) works with a program in a source language. Different analysis phases collect information about the source program and build it's representation together with checking different properties, such as lexical and grammatical structure, correct usage of data types etc. Synthesis (often called the *back end* of the compiler) uses information produced during the analysis to generate a program in a target language, usually performing a lot of program transformations and optimisations along the way [Aho et al., 2006]. A typical compiler pipeline is pictured in Figure 1.1. These phases do not necessarily happen one after another in a sequence, some of them might be combined, for example, but it is a convenient way of thinking about a compiler structure.

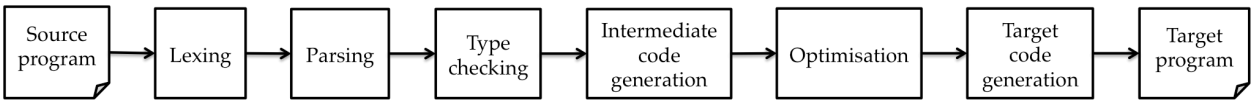

Figure 1.1: Compiler pipeline

One of the central parts of many compilers and, in particular, optimising compilers is their *intermediate representation(s)* (IR) or *intermediate language(s)* (IL). Intermediate representation is a data structure that represents the program being compiled during the compilation process. Many intermediate representations are also called languages, because they have a symbolic representation. For this reason we will mostly use the term intermediate language in this text. One of the simplest examples of an IL is a syntax tree built by parsing and *desugaring* a program in a source language (substituting convenient, but not essential language constructs with more basic ones). Well-designed intermediate language should allow to abstract away unnecessary details while expressing the necessary ones at certain compilation stages. It should be fairly easy to produce as well as translate further. Many compilers use several different ILs, where they diverge from the source language and resemble a compilation target's language or architecture more and more as compilation progresses. Some ILs are quite independent of both the source and the target, which allows them to be used in retargetable compilers (compilers which can generate code for several

different target languages/architectures) by combining different front ends and back ends which work with the same IL [Aho et al., 2006].

A major task performed by most modern industrial compilers is code optimisation. Compiler writers try to make compiled programs perform as good as possible regarding certain properties. The most common optimisations target code execution speed (performance), code size and it's energy efficiency (which becomes increasingly important given the advances in development of mobile devices and their ubiquity). Code optimisation is also a very complex task, since it needs to achieve good results, but even more importantly, not to change program's observable behaviour. Combining these two aspects is extremely difficult given a myriad of effects that modern programming languages allow to perform. It includes, but is not limited to input/output, throwing exceptions, modifying state, non-termination. Since IL is a compiler's main source of information about the program being compiled (together with some other data collected during the analysis) as well as the object the compiler transforms to finally produce the target program, it can play a significant role in the optimisation process.

This work tries to design an intermediate language based on monads and monad transformers to allow powerful transformations and optimisations in the presence of effects. The problem with many ILs is that a lot of valuable information about programs is not captured in the IL itself, but instead a compiler needs to track this information somehow. One of the examples is the effects a program may perform. The compiler has to infer information about effects in order to decide whether it is safe to perform certain optimisations or not and then record this in some separate data structures. Moreover, these data structures need to be updated as transformations happen. Having an IL based on monads we can encode effects right in the program and apply algebraic properties and laws to perform code transformations.

Another area which this thesis tries to address is providing a flexible and generic way of expressing parts of programming language semantics by combining monads using monad transformers. In a usual setting to change the semantics of a programming language, one needs to rewrite the code generation part of the compiler for this language. As it will be described later, with monad transformers it is possible to change certain parts of the semantics by just changing the order in which monads are combined.

Finally, this work attempts to implement a programming library for working with the introduced intermediate language and a set of effect-aware code transformations and optimisations, which can be reused by compiler writers targeting the designed IL. To evaluate and examplify the usage of the intermediate language, two programming languages (object-oriented and functional) are designed and compilers for them, which target the IL are implemented.

If we consider the current state of compiler technology, we can see that there are quite many different intermediate languages that are in use. Some of the most notable are:

- Three-address code (TAC or 3AC). A simple language, which is based on the idea, that there is at most one operator on the right-hand side of an instruction. This implies that all instructions are quite simple (for example, there are no built-up arithmetic expressions), which is beneficial for optimisation and target code generation [Aho et al., 2006].
- Languages based on static single assignment form (SSA) [Alpern et al., 1988], the main idea of which is that every variable in a program is assigned exactly once. It is similar to TAC in

the way that the result of every operation is bound to a variable. One of the main benefits of the SSA form is that many useful and powerful optimisations can be expressed in simple and efficient ways [Cytron et al., 1991].

- Compilers for functional languages often use CPS (continuation-passing style) as an intermediate language [Appel, 2007]. CPS is a style of programming where functions take an extra argument called *continuation*. Continuation is a function, which has one parameter and is called with the return value of the function that is given the continuation as an argument. It has been shown that there is a correspondence between CPS-based intermediate languages and the SSA form, however they are not equivalent: "some CPS programs cannot be converted to SSA" [Kelsey, 1995].
- A-normal form (ANF) introduced by Sabry and Felleisen [Sabry and Felleisen, 1992] and further developed in [Flanagan et al., 1993] is a notation that resembles CPS. ANF requires all arguments to functions to be trivial (constants, lambda abstractions and variables) and that the result of any non-trivial expression is let-bound or is returned from a function. ANF is essentially a different notation for the same idea as the one behind the SSA form [Kelsey, 1995], [Appel, 1998].
- Intermediate languages based on monads. These will be described in more detail in the later chapters.

There are several specific examples of ILs used in industrial compilers that are worth mentioning here, namely:

- Core, STG and C-- (Cmm) in GHC (The Glasgow Haskell Compiler). Core is a simple functional language (much smaller than Haskell) that is used for optimisations and further code generation [Eisenberg, 2015]. STG is an intermediate language produced from Core and it's intention is to define how to efficiently implement Haskell on standard hardware [Peyton Jones, 1992]. C-- is a C-like portable assembly language [Peyton Jones et al., 1999] and is an example of the three-address code.
- Register transfer language (RTL) in GCC (GNU Compiler Collection) is a low-level intermediate language with a syntax inspired by Lisp lists (https://gcc.gnu.org/onlinedocs/gccint/RTL.html). It is a language for an abstract machine with virtual (pseudo) registers and is another example of the three-address code. RTL is based on the idea described in [Davidson and Fraser, 1980].
- LLVM Intermediate Representation. It is a language used in the LLVM compiler infrastructure (http://llvm.org), which aims to provide a language independent compiler toolchain with a number of optimisations. LLVM IR is a language for an abstract machine with an infinite set of registers, it makes use of SSA form, has a simple type system, and abstracts away some low-level details such as a calling convention [Lattner and Adve, 2004].

The IL designed as a part of this work is quite different from most of the ILs described above. None of the mentioned ILs encodes different program effects and they don't try to allow to express parts of the language semantics on the high level either. Probably, the closest IL to the one we present is GHC Core, which is also a rather high level functional language.

A lot of inspiration and intuition for this work comes from the Haskell programming language [Marlow, 2010], thus some familiarity with Haskell is assumed.

Chapter 2

Monads, semantics and functional programming

This chapter introduces monads – the central concept of this thesis. Applications of monads in semantics and functional programming are described. At the end, we talk about a way to combine different monads – monad transformers. More details in relation to this work is left for one of the following chapters.

2.1 Monads introduction

Monad is quite an overloaded word. It is used in philosophy, linear algebra, music, biology and probably some other areas. Monads related to this work is a concept, which comes from category theory. But here it is more useful to use a definition adopted in functional programming.

Monad is a triple of a type constructor `M` and two operations: `bind` and `return`. Type constructor `M` has one type parameter. In Haskell, the monad concept is represented by a type class, which also has some additional operations. Monad operations have the following types:

$$\texttt{return} :: \texttt{a} \to \texttt{M a}$$

$$\texttt{bind} :: \texttt{M a} \to (\texttt{a} \to \texttt{M b}) \to \texttt{M b}$$

The `return` operation takes a value of type `a` and returns a monadic computation that just returns that value and does not do anything else. The `bind` operation takes a monadic computation that can produce a value of type `a` and a function that can consume that value and give a computation of type `M b`. So, `bind` gives a way to apply such a function to a computation of type `M a` to get a computation of type `M b`. Since `bind` is very often used as an infix operator, it's type usually has the argument before the function.

Monad operations must satisfy the following three laws:

- *Left identity*:

$$\texttt{bind}\ (\texttt{return}\ \texttt{x})\ \texttt{f} = \texttt{f}\ \texttt{x}$$

  It means that feeding the result of the return computation which just returns a value to function `f` is the same as just applying function `f` to that value.

- *Right identity*:

$$\texttt{bind}\ \texttt{m}\ \texttt{return} = \texttt{m}$$

  This law means that computing the result of `m` and then returning that result is the same as just computing `m`.

  Both identity laws can be connected to the identity laws for multiplication with 1 and addition with 0.

- *Associativity*:

$$\texttt{bind}\ (\texttt{bind}\ \texttt{m}\ \texttt{f})\ \texttt{g} = \texttt{bind}\ \texttt{m}\ (\lambda \texttt{x} \rightarrow \texttt{bind}\ (\texttt{f}\ \texttt{x})\ \texttt{g})$$

  Associativity law has a similar idea to associativity laws in arithmetic, namely that the order of parentheses does not matter [Wadler, 1995].

2.2 Examples of monads

In this section we are going to look at examples of several monads.

- The simplest of all monads is the `Identity` monad. It does not decorate computations with any effect or information. The `bind` operation for the `Identity` monad is basically just a function application. We provide a definition of `Identity` in Haskell below:

```haskell
newtype Identity a = Identity a

instance Monad Identity where
  return = Identity
  (Identity x) >>= k = k x
```

  It might not seem very useful at first, but the `Identity` monad helps to combine several monads by being a base on top of which others build up. It is also useful when a computation is abstracted over a monad and a user of the computation does not want to use any specific effects with this computation, so she can choose the `Identity` monad as the monad instance in this case.

- The next example is the `State` monad for stateful computations. Since in most functional languages immutability is encouraged by default, the usual technique to deal with state is to pass it around as function arguments. One of the possible `State` monad implementations pretty much makes this technique implicit. Here is such an implementation in Haskell:

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

instance Monad (State s) where
  return x = State (\s -> (x, s))
  (State h) >>= f = State (\s -> let (a, s') = h s
                                     (State g) = f a
                                 in g s')
```

One can think about a stateful computation as taking a state and producing a value and the new state. This is what the type definition above captures. The `return` function produces a stateful computation that just gives the given value back without modifying the state. The `bind` operation chains stateful computations together and passes the result and states between them.

There are usually some useful operations associated with a monad. For the `State` monad those are, for example, `get` and `put`, which are used to read and write the state value, respectively. Below is an example of their implementation in Haskell:

```haskell
get :: State s s
get = State (\s -> (s, s))

put :: s -> State s ()
put x = State (\s -> ((), x))
```

- Another ubiquitous effect is error (exception) handling. A monad which can be used to capture this effect is the `Either` monad:

```haskell
data Either e a = Left e | Right a

instance Monad (Either e) where
  return = Right
  Right x >>= f = f x
  Left e >>= f = Left e
```

Data constructor `Left` represents an error (represented by a value of type `e`) that occured during the computation, while `Right` denotes a successful computation with a value of type `a` as the result. In the `Either` monad `return` is simply `Right`. The `bind` operation passes the result of a successful computation to the next one, but in the case of an error, it stops and just propagates the error further.

Examples of useful operations for the `Either` monad are `throwError` and `catchError`. The former is used to produce an error value, and the latter allows to *handle* an error value and potentially produce some other value:

```haskell
throwError :: e -> Either e a
throwError = Left
```

```haskell
catchError :: Either e a -> (e -> Either e a) -> Either e a
catchError (Left e) h = h e
catchError (Right a) _ = Right a
```

- Very famous in the Haskell world `IO` is also an example of a monad. We will skip a representation of the `IO` monad in Haskell, but one can think about it as a `State` with "real world" as a storage. Since `IO` wraps arbitrary side-effects and interaction with the outside world, considering such a computation as the one that changes some very global state (of the world) is a rather useful metaphor.

2.3 Monads in semantics

One of the first applications of monads to programming languages was done by Eugenio Moggi. Moggi applied monads to structure the denotational semantics of programming languages [Moggi, 1991]. He was studying the notions of computations such as partiality, nondeterminism, side-effects, exceptions, continuations, interactive input and interactive output and proving equivalence of programs with those effects using a calculi based on a categorical semantics. There is also a "modular approach" to denotational semantics proposed in [Moggi, 1990].

2.4 Monads in Haskell

Later, following and building on top of Moggi's work, Philip Wadler incorporated monads as a way of structuring pure functional programs in Haskell. First, monads were used to combine input/output and lazy evaluation, and then they became an integral part of Haskell and are used for expressing state manipulation, error handling, environment reading, collecting output, nondeterminism, continuations and other [Wadler, 1990], [Peyton Jones and Wadler, 1993], [Wadler, 1995]. Monads were also successfully used for building parsers [Hutton and Meijer, 1998] and working with concurrency using Software Transactional Memory [Harris et al., 2005].

Nowadays, monads is a very important and popular programming pattern in Haskell. They are used extensively for a wide variety of applications, for example backtracking [Kiselyov et al., 2005] and parallelism [Marlow et al., 2011] among others.

2.5 Monad transformers

Monad transformer is a type constructor `T` which takes a monad `M` and returns a monad, or in other words, if `M` is a monad, so is `T M` [Liang et al., 1995].

Monad transformers are used to add new operations to a monad without changing the computation in that monad, for example state manipulation can be added to collecting ouput. They also compose easily, meaning that it is possible to apply a monad transformer to another monad transformer. The resulting structure is usually called *monad transformer stack*. There is one caveat,

though, namely, that effects produced by some of the monad transformer/monad combinations depend on the order in which they are combined.

Monad transformer comes with `lift` operation, which embeds a computation in monad `M` into monad `T M`. It has the following type:

$$\texttt{lift} :: \texttt{M a} \rightarrow \texttt{T M a}$$

The main intention for `lift` is to be used to specify on which level a certain monadic operation is performed.

Monad transformers must satisfy the following two laws:

- Lifting a monadic computation which only does `return` results in the same monadic computation.
$$\texttt{lift . return} = \texttt{return}$$
- Lifting a sequence of monadic computations is the same as lifting them individually and then combining lifted computations in the new monad.
$$\texttt{lift (bind m k)} = \texttt{bind (lift m) (lift . k)}$$

Monad transformers and their application to building modular interpreters are introduced and described in detail in [Liang et al., 1995].

To give a better understanding of monad transformers, we provide several examples of them:

- `StateT s m` is a transformer that adds an effect of a stateful computation to monad `m`. An implementation in Haskell could be the following:

  ```
  newtype StateT s m a = StateT (s -> m (a, s))
  ```

  Using the `StateT` monad transformer and the `Identity` monad, `State s` monad can be expressed as `StateT s Identity`.

  Every monad transformer also has to be an instance of `Monad`, but we will skip these definitions here. They would be quite similar to the example definitions in the "Examples of monads" section above, but a bit more involved.

  To make `StateT` a monad transformer in Haskell, we need to make it an instance of the `MonadTrans` type class, which contains the `lift` operation (note the `Monad` constraint in the type of `lift`):

  ```
  class MonadTrans t where
    lift :: (Monad m) => m a -> t m a

  instance MonadTrans (StateT s) where
    lift m = StateT (\s -> do
                a <- m
                return (a, s))
  ```

The `do` block above uses the underlying monad's `bind` and `return` and just returns the given state together with the underlying computation's result.

- `ErrorT e m` adds a notion of failure to monad `m`. One way to represent it is using the `Either` data type:

```
newtype ErrorT e m a = ErrorT (m (Either e a))

instance MonadTrans (ErrorT e) where
  lift m = ErrorT (do
    a <- m
    return (Right a))
```

To make the computation `m` to be an `ErrorT` computation we just bind its result in the underlying monad and then return the result as success using `Right`.

- `ReaderT r m` adds a layer of interaction with a read-only environment of type `r`:

```
newtype ReaderT r m a = ReaderT (r -> m a)
```

By analogy with `State` above, `Reader r` monad is `ReaderT r Identity`.

`lift` implementation below returns a computation in `ReaderT`, which just disregards the given environment and gives back the computation in the underlying monad:

```
instance MonadTrans (ReaderT r) where
  lift m = ReaderT (\r -> m)
```

One of the disadvantages of monad transformers for practical programming is the problem of lifting. When one has a stack of several monad transformers and wants to use an operation that is defined with the type containing only one of them, like `get` or `put` for `StateT`, for example, it is not possible to just use them directly (unless `StateT` is on top) because the types do not match. In this case lifting needs to be used, to add necessary layers on top of a computation in one of the monads deeper down in the stack. There are attempts to solve the lifting problem, for example by using type classes to abstract different types of computations and use these abstractions instead of specific monad transformers. This idea is based on [Jones, 1995] and incorporated in Haskell packages like `mtl`. It is rather convenient for a user, but has quite an overhead for a library writer, because the number of instances that needs to be provided is quadratic to the number of monads [Liang et al., 1995]. Another, more recent idea, is presented in [Jaskelioff, 2009], where "a uniform lifting through any monad transformer" is defined. There is also work on an alternative to monad transformers based on *algebraic effects* and *effect handlers* being done. This approach is covered in more details in the following chapter.

As it was mentioned earlier, the order in which different monads are combined can be significant. If it is the case, then it is said that these two effects do not *commute*. Probably, the most famous example is the ordering of `State` and `Error`. One can see this by "unrolling" the types:

- `StateT s (ErrorT e Identity) a` ⇒ `s -> ErrorT e Identity (a, s)` ⇒
  ⇒ `s -> Identity (Either e (a, s))` ⇒ `s -> Either e (a, s)`
- `ErrorT e (StateT s Identity) a` ⇒ `StateT s Identity (Either e a)` ⇒
  ⇒ `s -> Identity (Either e a, s)` ⇒ `s -> (Either e a, s)`

In the first case, when `StateT` is on top of `ErrorT` the semantics of a computation is that it either fails and returns an error or it succeeds and returns a value and a new state. One can think of this as a kind of rollback. On the other hand, in the second case, when the order is reversed, the computation always returns a new state, regardless of whether it gives an error or a result.

Chapter 3

Related work

Usage of monads in compiler's intermediate languages is quite unexplored. At the same time, nowadays the interest in reasoning about effects of a program and their control is significant. First, we will look at several intermediate languages which use monads. Then we will describe examples of monad transformers' usage to achieve modularity with effects. And finally we will cover some related work in programming with effects, which may serve as a source of inspiration for intermediate languages as well as reasoning about program effects in general.

3.1 Intermediate languages based on monads

3.1.1 Common IL for ML and Haskell

First, we will describe some of the ideas behind an intermediate language proposed in "Bridging the gulf" [Peyton Jones et al., 1998]. The main intention of the work was to design a common intermediate language for compiling ML [Milner et al., 1997] and Haskell [Marlow, 2010] programming languages. One of the main differences between these two languages is that ML has strict evaluation and Haskell is a lazy language. In addition to this, Haskell is a pure language, while ML is not. Therefore, the main challenge was to have an IL that will work equally well for both. Note that this setting and the problem are slightly different from what this thesis tries to address, but nevertheless it is still very interesting to see one of the not so many monadic ILs and the considerations authors had when designing it. Certain ground rules for an IL are given, namely: ability to translate both core ML and Haskell, explicitly typed with a type system that supports polymorphism (a variant of System F [Girard, 1986]), well-defined semantics, efficiency of compiled programs comparable to those produced by good ML and Haskell compilers. Authors propose two designs of such an IL. Both of them are based on monads. The idea behind the first IL is to be explicit about things that are usually implicit in the IL, but are explicit in the denotational semantics of it. There are two monads in this IL, lifting (for denoting possibly-diverging computations) and the combination of lifting with the state transformer `ST` (to distinguish between pure and stateful computations). The main reason for using the lifting monad is to express whether a function takes an argument, which is evaluated or not. Semantics of the monads is made implicit

in the language. Unfortunately, the first proposed design turned out to be vague about the timing and degree of evaluation and solutions to these in the presence of polymorphism are complicated and therefore, authors conclude that it is unsuitable as a compiler IL. The second IL design introduces a distinction between value types (for variables) and computation types (for expressions), let expression becomes eager and new syntactic forms (evaluation suspension and forcing) are introduced, thus making the control of evaluation explicit. The `Lift` monad becomes implicit (and covered by types distinction and new syntactic forms) and so there is only the `ST` monad left. Still having this monad explicit allows to express that certain computations are free from side-effects and perform useful transformations based on this information, which is recorded in the program itself. The second design seems to address the problems found in the first one, but as the authors highlight, the ability of the first IL design to capture certainly-terminating computations is very useful and they would like to incorporate it into the second design. The problem of combining monads was not addressed in "Bridging the gulf". Several monad transformation rules are given, but they were not in the main focus of this paper.

3.1.2 Optimizing ML using a hierarchy of monadic types

Next, we will look at an intermediate language from "Optimizing ML Using a Hierarchy of Monadic Types" [Tolmach, 1998]. The main property of this IL is that it has a fixed hierarchy of monadic types (every next monad includes the effects of the previous ones):

- `ID` (for pure, terminating computations)
- `LIFT` (for potentially non-terminating computations)
- `EXN` (for computations that may raise an exception)
- `ST` (for stateful computations).

In addition to the classical monad operations (`bind` and `return`), an embedding operation named `up` is introduced, which corresponds to the `lift` operation for monad transformers, introduced in the previous chapter. An interesting decision problem about the combination of state and exception arises (it is an example of a general problem of monad transformers' composition mentioned in the previous chapter). The author chooses exception handling that does not alter the state (as opposed to reverting the state on exception handling). There is a number of generalised monad laws and code motion laws for monadic expressions given in the paper. Another valuable contribution of Tolmach's work is a monad inference algorithm for computing the minimal monadic effect of subexpressions, which allows to perform many more useful transformations comparing to assigning the maximal effect. The author makes several supporting claims for a monadic intermediate language: its usage in organising and justifying a variety of optimising transformations in the presence of effects, reflecting the results of the effect inference in the program itself as well as keeping them up to date as different optimisations progress (as opposed to using some separate data structure for this). At the time of the paper's appearance there was no evidence on whether the results help to achieve any significant performance improvements for generated ML code, but some measurements were ongoing. At the time of this thesis writing no published results of these measurements have been found.

3.1.3 MIL-lite

The next work we are going to describe is an intermediate language MIL-lite from "Monads, Effects and Transformations" [Benton and Kennedy, 1999]. MIL-lite is a fragment of the monadic intermediate language used in the MLj compiler. A big part of the paper is devoted to semantics of effects, but we are going to concentrate on the IL design and the approach to expressing effects. Most terms of the language are straight-forward terms found in functional languages. MIL-lite is quite minimal and some syntactic sugar is expressed in terms of this core. There is a novel construct that is introduced: try-catch-in. One can view it as a blend of monadic bind (let) and exception handling. The authors claim that this new construct is suitable for expressing many optimising transformations and is very general. let expression is actually expressed in terms of try-catch-in. A distinction between value types and computation types is made similarly to [Peyton Jones et al., 1998]. A computation type is effectively a value type with effects. The following effects are covered:

- non-termination
- allocating a new reference
- reading from a reference
- writing to a reference
- raising an exception

Note how fine-grained the effects for stateful computations are. Effects are combined using sets and inclusion of these sets introduces a subtyping relation. All possible exceptions are also included in the set of effects. Authors provide a number of effect-independent and effect-dependent equivalences and use the reasoning about semantics of effects to prove some of them.

3.1.4 The GRIN project

The last intermediate language described in this section is the one from "The GRIN project" [Boquist and Johnsson, 1997]. GRIN (Graph Reduction Intermediate Notation) is a monadic intermediate code that is used in a back end of a compiler for lazy functional languages. GRIN resembles three-address code mentioned in the Introduction. For the monadic part, it has `unit` operation and ; (semicolon) as a monadic bind. The monad is a kind of State monad with a heap as an underlying storage. There are `store`, `fetch` and `update` operations in the monad. The authors highlight that the monadic structure gives GRIN a very "functional flavour" and therefore a nice setup for doing analysis and transformations.

3.2 Monad transformers and modular effects

In this section we will look at work that has been done in the area of monad transformers and using them to combine different effects. Note that in these cases monad transformers are used mainly in the implementation language, rather than in an IL, as described below. It is a bit different area

of monad transformers' application compared to this thesis, but it still serves as a great source of inspiration.

Probably, one of the most influential papers in this area is "Monad Transformers and Modular Interpreters" [Liang et al., 1995]. It describes how to structure an interpreter for a programming language, where language features are "pluggable" and the evaluation of each feature is implemented separately (in different type class instances in Haskell, in this case). Features have a strong relation to the effects performed (where effects correspond to monad transformers). By adjusting the order of monad transformers in the stack, one can choose different semantics of the language. Another key idea that allows this kind of implementation is *extensible union types*, which is used to specify different parts of the language's AST (abstract syntax tree) in a modular way. A continuation of this work "Modular Denotational Semantics for Compiler Construction" [Liang and Hudak, 1996] applies the results to define a modular monadic semantics and use it for a compiler. Authors describe usage of monad laws to transform programs and reason about them.

What was not addressed in the work described above in particular is more low-level code generation. So, later, Harrison and Kamin in [Harrison and Kamin, 1998] demonstrate a compiler structured in blocks for every feature, where a block is a compilation semantics for a feature and its associated monad transformers. They use partial evaluation of monadic expressions to generate code. Harrison's thesis extends this work and includes correctness proofs for such compilers [Harrison, 2001].

Another notable work on modular compilers is by Day and Hutton [Day and Hutton, 2012]. To structure the language's syntax in a modular way, the authors used an approach know as *"data types á la carte"* [Swierstra, 2008], which is somewhat similar to extensible union types mentioned earlier. On top of this they build a modular evaluator using monad transformers, but the decision about the underlying monad, to which transformers are applied is deferred until the application of top-level evaluation function. Then a modular compiler for a stack machine is described. The interesting point here is that the compilation scheme does not utilise a monad, since the compilation process itself is not connected to program effects. The authors conclude with a modular virtual machine that can execute compiled code.

3.3 Programming with effects

There is a growing interest in bringing controlled and expressive effects into programming languages. This section is by no means a comprehensive survey of the current state of effect systems, supporting libraries and programming languages, but rather a few examples of different directions that are being explored.

3.3.1 Koka programming language

First, we will look at the Koka programming language [Leijen, 2013], which is a function-oriented language with JavaScript-like syntax. One of the main features of Koka is that the effect of every function is automatically inferred. The supported effects are the following:

- `total` (pure mathematical functions)
- `exn` (throwing exceptions)
- `div` (non-termination)
- `ndet` (non-deterministic functions)
- `alloc` $<$ `h` $>$ (memory allocation in a heap `h`)
- `read` $<$ `h` $>$ (reading of a heap `h`)
- `write` $<$ `h` $>$ (writing to a heap `h`)
- `io` (input/output operations)

The effect system is based on row polymorphism. This enables the fact that effects can be combined into rows, for example $<$ `exn`, `div` $>$ means that a function can throw an exception and can diverge, which is basically the Haskell's notion of purity, but more than that, effects can be polymorphic, like $<$ `exn`, `div`|`E` $>$, where `E` is an effect variable.

3.3.2 Polymonads

One of the most recent ideas in expressing effects is a *polymonad*, which is a generalisation of monads [Hicks et al., 2014]. The main idea is that polymonads give the monadic `bind` a more general type:

$$\texttt{polyBind} :: \texttt{L a} \rightarrow (\texttt{a} \rightarrow \texttt{M b}) \rightarrow \texttt{N b}$$

Comparing to the monadic `bind`:

$$\texttt{bind} :: \texttt{M a} \rightarrow (\texttt{a} \rightarrow \texttt{M b}) \rightarrow \texttt{M b}$$

Polymonadic bind allows to compose computations with three different types instead of one. Similar to monads, polymonads must satisfy a number of laws. Polymonads allow to express different type and effects systems and information flow tracking, as an example.

An interesting example of what polymonads can model is *contextual effects*. Hicks et al. describe them as effects "which augment traditional effects with the notion of *prior* and *future* effects of an expression within a broader context". An example they give is a system where memory is partitioned into regions and read and write operations have a region as part of their effect. Then a sequence of read operations, for example, can capture which regions were read before and after every operation.

3.3.3 Eff programming language

The next example of programming with effects is another programming language, called Eff [Bauer and Pretnar, 2012]. In Eff effects are viewed as algebraic operations and they together with their handlers have first-class support. This is an approach known as *algebraic effect handlers* introduced by Plotkin and Power [Plotkin and Power, 2003]. The language is statically typed and

supports parametric polymorphism and type inference. In addition to the usual types, Eff also has effect types and handler types. An effect type denotes a collection of related operations, a handler type describes that handlers work on computations of one type and produce computations of some other type. There are several specific language constructs in Eff, namely *instantiation* of an effect instance (`new ref` or `new channel`, for example), *operation*, which can be applied when bundled with an effect instance, and *handler*, which is somewhat reminiscent to the pattern matching constructs found in functional languages. It defines which computation to perform depending on the evaluation of another computation, which can result in a value or an effect operation. This is the place where the semantics of different effects is defined. A handler can be applied to a computation using the `with − handle` construct, which reminds the `try − catch` construct in many languages. The last of these specific constructs is a *resource*, which allows to create an effect instance that is associated with the resource. In this case resource describes how to handle different operations for this effect instance by default as well as defining an initial state. The language constructs, its type system and denotation semantics is layed out in the paper. The authors also provide many interesting examples, ranging from stateful computations to transactions, backtracking and cooperative multithreading. Unfortunately, the Eff's type system does not capture effects in the types and it is highlighted in the paper that the language would benefit from a system that provides a static analysis of effects.

3.3.4 Algebraic effects and dependent types

Another example of programming with effects based on algebraic effects and inspired by the Eff programming language is described in [Brady, 2013b], which introduces `Effects` – a library for the Idris programming language [Brady, 2013a]. In the general case, there is a type `EffM` that describes a program using some *computation context* (which can be a monad, for example, but it does not have to be), lists of input and output effects as well as the program's return type. Programming using `Effects` from a user point of view is quite alike monadic programming in Haskell, since, for example, monadic do-notation is used. Each effect is associated with a *resource*, which can denote storage for stateful computations, for example. To run an effectful computation one must specify an initial value of the resource. To solve the problem similar to the one solved by lifting in monad transformers, `Effects` uses *labelled* effects to resolve the ambiguity. An effect is usually implemented as an algebraic data type (ADT) as well as an implementation of the handler for this effect. Handlers can be implemented for specific contexts as well as for the general case. The author highlights that monads and monad transformers can express more concepts, but `Effects` capture many useful use cases. Implementation of common effects and examples of their usage as well as the library implementation in Idris are described in details in the paper.

3.3.5 Extensible effects

The last in this chapter is an example of a library-based approach. Extensible effects is a library for the Haskell programming language [Kiselyov et al., 2013]. It is positioned as an alternative to monad transformers. The central concept of this library is a monad `Eff r`, where `r` is a type parameter that represents an *open union* of individual effects, that are combined (`r` stands for "requests").

One of the main ideas behind the library is that effects come from communication between a client and an effect handler (authority), which is quite similar to the algebraic effect handlers approach. The implementation is described in detail in the paper together with the examples of implementation of different effects as a user code (meaning, it can be done outside of the library). The authors provide a detailed analysis of monad transformers and problems with their expressiveness. One of the highlights of differences from monad transformers is that effects of the same type can be combined and as opposed to monad transformers, no explicit lifting is needed, appropriate operations are found by types. Another difference is that the order of effects is chosen only when running a computation, not when defining the computation and in addition to that handled effects are subtracted from the type. The authors claim that their framework subsumes the Monad Transformers Library (MTL) in Haskell and allows to express computations, that are not possible with MTL.

It is worth mentioning that algebraic effect handlers while having a good story of modularity suffer from performance problems (compared to monad transformers). As Wu and Schrijvers point out in [Wu and Schrijvers, 2015], modularity comes from the fact that syntax and semantics of effects are separated and so different handlers implementing different semantics can be provided for the same syntax tree. At the same time, "the construction and traversal of intermediate trees is rather costly". The work described in [Wu and Schrijvers, 2015] attempts to address this problem by trying to fuse several handlers into a single one and thus improving the performance.

There is also a follow up work on "Extensible effects" aimed at improving the algorithmic efficiency of the library and its simplification [Kiselyov and Ishii, 2015].

Chapter 4

Monadic Intermediate Language (MIL)

In this chapter we present the main part of this thesis – Monadic Intermediate Language. First, the language overview and some examples are given. Then we specify its grammar, the type system and describe representation of effects in detail. After this, some parts of the Haskell implementation are described. We conclude with a discussion and some comparison to the related work.

4.1 Overview

As was described in the Introduction, the main goal of this thesis is to design a compiler IL powerful enough to be used in compilers for modern programming languages and support reasoning about programs with effects and their transformation for the purpose of optimisations. Monadic Intermediate Language (MIL) is the result of this effort.

MIL is a rather small strict functional language. Its type system is based on System F (or polymorphic lambda calculus) [Girard, 1986]. Effects are modelled with monads, and monad transformers are used to combine the monads. MIL also has a number of additional features allowing to more easily express many features found in modern programming languages.

4.2 MIL by example

In this section we will look at some examples to get the feel for MIL syntax. Some common terminology will also be established.

4.2.1 Data types

MIL supports simple algebraic data types (ADTs), found in many statically typed functional programming languages. Here are the examples of the usual definitions of `Bool` and `List` data types:

```
type Bool = True | False;

type List A = Nil | Cons A (List A);
```

Here, `Bool` and `List` are also called *type constructors*. `List` type is *parameterised*, it takes one *type parameter* `A`, which can be instantiated with any type. We say that type `Bool` has *kind* `*` (star) and type `List` has kind `* => *` meaning that it takes a type and returns a type (hence the name type constructor). Kinds can be thought of as *types of types*. Type `Bool` has two *data constructors*: `True` and `False`. `List` also has two data constructors: `Nil` for empty list and `Cons` which carries a value of type `A` and a list. This means that `List` data type is *recursive*.

Every data constructor can be used as a function. For example, `True` has just type `Bool` (because it does not have any other data associated with it, so it is just a value of type `Bool`), `Cons` has type `forall A . A -> List A -> List A`, meaning that for any type `A` (`A` has kind `*`) it can be applied to a value of type `A` and a value of type `List A` producing `List A`. Data constructors can also be partially applied. There is some simplicity to the fact, that data constructors are treated pretty much in the same way as variables and functions, both for MIL users and for the MIL implementation.

4.2.2 Functions and expressions

Simple functions in MIL can be definied as follows:

```
intId : Int -> Int = \(i : Int) -> i;
```

MIL is *explicitly typed* (there is no type inference like in Haskell or ML), so the types of functions and variable binders must be specified. In this example, `intId` is the identity function for integers, it has the type `Int -> Int` (it takes an integer and returns an integer). The body of the function is a lambda expression.

Polymorphic functions look like this:

```
id : forall A . A -> A = /\A . \(x : A) -> x;
```

This is a polymorphic version of the identity function, which works on any type. It has a *universally-quantified* type (using `forall` keyword). The body of the function is the so-called "big lambda" or "type lambda", which introduces a *type variable* `A` (and `forall A` in the type) and then a lambda expression, where the variable binder for `x` uses the type variable `A`.

Functions are applied using juxtaposition (placing arguments next to functions using a whitespace). Square brackets are used for *type application* (instantiating a type variable in a `forall` type). For example, applying `id` to `True`:

```
true : Bool = id [Bool] True;
```

MIL has built-in tuples to be able to group values of different types together. Tuples can be empty as well.

```
empty : {} = {};

tuple : {Int, Bool, Float, Char} = {1, True, 1.23, 'c'};
```

The example above also demonstrates integer, floating point and character literals.

4.2.3 Bind and Return

The bread and butter of MIL is monadic `bind` and `return`. The following example uses `bind` (`let ... in` expression) to bind the result of `read_char` built-in function to the variable `c` and then return it in the `IO` monad (`return` needs to be annotated with a monad):

```
let (c : Char) <- read_char
in return [IO] c
```

4.2.4 Lifting

Another crucial piece is the monad transformer `lift` operation. The following example demostrates how to lift a computation in the `IO` monad into a combination of `State` and `IO` monads:

```
lift [IO => State ::: IO] return [IO] unit
```

The `lift` construct expects any MIL expression after monads (which are provided inside the square brackets). The typing rules and details of monads combination will be covered in the "Type system" section.

4.2.5 Pattern matching

MIL supports simple *pattern matching* with `case` expression. An expression which we match on (*scrutinee*) is evaluated and checked against patterns in the order they are written. For the first successful match the right-hand side (after `=>`) of the corresponding *case alternative* is the result of the `case` expression.

```
case x of
  | True => 0
  | False => 1
end
```

Possible patterns are literals, variable binders, data constructors (with variable binders for their data elements), tuples (with variable binders for elements) and the *default pattern* (underscore), which matches anything. Patterns cannot be nested.

4.2.6 Recursive binding

Recursive bindings inside a function can be introduced using `let rec` expression. It can specify several bindings (which can be mutually recursive) at once. The result of `let rec` expression is a value of the *body expression* (after `in`).

```
let rec
  (isEven : Int -> Bool) <- \(i : Int) -> ... isOdd ...;
  (isOdd : Int -> Bool) <- \(i : Int) -> ... isEven ...
in isEven 4
```

Although `let rec` has a quite similar syntax to the `let` expression, they should not be confused. The `let` expression being a monadic `bind` can have only one variable binder and requires its binder and body expressions to have monadic types. The `let rec` expression should be used purely for introducing one recursive binding or several mutually recursive bindings at once. One use case for it will be described in one of the following chapters.

4.2.7 Built-in data types and functions

MIL has several built-in types: `Unit` (which has only one value, which is the `unit` literal), `Bool` (defined as in one of the data type examples earlier), `Int` (for arbitrary integers), `Float` (for double precision floating point numbers), `Char` (for characters) and `Ref` (for reference cells).

MIL also provides a number of built-in functions. Here is the list with most of them and their types:

```
read_char : IO Char
print_char : Char -> IO Unit
new_ref : forall A . A -> State (Ref A)
read_ref : forall A . Ref A -> State A
write_ref : forall A . Ref A -> A -> State Unit
throw_error : forall E . forall A . E -> Error E A
add_int : Int -> Int -> Int
add_float : Float -> Float -> Float
sub_int : Int -> Int -> Int
sub_float : Float -> Float -> Float
mul_int : Int -> Int -> Int
mul_float : Float -> Float -> Float
div_int : Int -> Int -> Error Unit Int
div_float : Float -> Float -> Error Unit Float
```

4.3 Grammar

In this section the full grammar of MIL is outlined. The following tokens are used: *upper* for an upper case letter, *lower* for a lower case letter, *alphanum* for a possibly empty sequence of alphanumeric characters and an underscore. The details of integer, floating point and character literals are omitted. They are more or less what is found in most modern programming languages. {*pat*} denotes zero or more repetitions of *pat*, {*pat*}+ is used for one or more repetitions and [*pat*] for an optional *pat*.

program	→	{*typedef*} {*fundef*}	top-level definitions
typedef	→	`type` *upperid* {*typevar*} `=` *condefs* `;`	type definition
fundef	→	*lowerid* `:` *srctype* `=` *expr* `;`	function definition
condefs	→	*condef* {`|` *condefs*}	data constructor definitions
condef	→	*upperid* {*atomsrctype*}	data constructor definition
srctype	→	*appsrctype*	application type
		appsrctype `->` *srctype*	function type
		`forall` *typevar* `.` *srctype*	universally-quantified type
		appsrctype `:::` *appsrctype*	monad cons
appsrctype	→	*atomsrctype*	atomic type
		appsrctype atomsrctype	type application
atomsrctype	→	*typecon*	type constructor
		`{` [*srctypes*] `}`	tuple type
		`(` *srctype* `)`	parenthesised type
srctypes	→	*srctype* {`,` *srctype*}	source types
expr	→	*appexpr*	application expression
		`\` *varbinder* `->` *expr*	lambda abstraction
		`/\` *typevar* `.` *expr*	type lambda abstraction
		`let` *letbinder* `in` *expr*	monadic bind
		`return` `[` *srctype* `]` *expr*	monadic return
		`lift` `[` *srctype* `=>` *srctype* `]` *expr*	lifting
		`let rec` *letbinders* `in` *expr*	recursive binding
		`case` *expr* `of` {*casealt*}+ `end`	case expression
appexpr	→	*atomexpr*	atomic expression
		appexpr atomexpr	application
		appexpr `[` *srctype* `]`	type application
atomexpr	→	*literal*	literal
		var	variable
		conname	data constructor name
		`{` [*tupleelems*] `}`	tuple
		`(` *expr* `)`	parenthesised expression

letbinders	→	*letbinder* {`;` *letbinder*}	let binders
letbinder	→	*varbinder* `<-` *expr*	let binder
casealt	→	`|` *pattern* `=>` *expr*	case alternative
literal	→	`unit`	unit literal
			integer literal
			floating point literal
			character literal
tupleelems	→	*expr* {`,` *expr*}	tuple elements
pattern	→	*literal*	literal pattern
			variable pattern
			data constructor pattern
			tuple pattern
			default pattern
tupelempats	→	*tupelempat* {`,` *tupelempat*}	tuple element patterns
tupelempat	→	*var* `:` *srctype*	tuple element pattern
varbinder	→	`(` *var* `:` *srctype* `)`	variable binder
typecon	→	*upperid*	type constructor
typevar	→	*upperid*	type variable
conname	→	*upperid*	data constructor name
var	→	*lowerid*	variable
upperid	→	*upper alphanum*	upper case identifier
lowerid	→	*lower alphanum*	lower case identifier
		`_` *alphanum*	

4.4 Type system

This section is devoted to the details of the MIL type system and the effects representation. We will not provide a formal definition of the full type system, but rather focus on the crucial parts. For example, data type and function definitions will be omitted, since they are pretty straight-forward and similar to the typing in many other functional languages.

We use a common way of presenting typing rules using two-dimensional *inference rules* [Pierce, 2002]. We use this term for both *axioms*, which do not have a *premise* and rules with premise(s) (which go above the line) and *conclusion* (below the line). There can be several premises and in such case they are assumed to be combined with the logical *conjunction* ("and"). Rules can be read as "If what is stated in the premise(s) holds, then we may derive the conclusion". We use Γ to denote a *type environment*, which contains variables/functions/data constructors and their types. New bindings are added to it using comma. $\Gamma \vdash \mathtt{e} : \mathtt{T}$ should be read as "expression `e` has type `T` in the type environment Γ".

Variables (as well as functions and data constructors) get their types from the type environment Γ:

$$\frac{\mathtt{x} : \mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \quad \text{(T-Var)}$$

Lambda abstraction is what is found in most functional languages, but note that it does not allow variable shadowing:

$$\frac{\mathtt{x} \notin \Gamma \qquad \Gamma, \mathtt{x} : \mathtt{T}_1 \vdash \mathtt{e} : \mathtt{T}_2}{\Gamma \vdash \lambda(\mathtt{x} : \mathtt{T}_1) \to \mathtt{e} : \mathtt{T}_1 \to \mathtt{T}_2} \quad \text{(T-Abs)}$$

Function application also has a classical shape, but the argument and parameter types do not have to be the same, rather they need to satisfy the `isCompatible` relation, which we will define after all the rules in the "Monads and relations" subsection, together with other relations that are used:

$$\frac{\Gamma \vdash \mathtt{e}_1 : \mathtt{T}_1 \to \mathtt{T}_2 \qquad \Gamma \vdash \mathtt{e}_2 : \mathtt{T}'_1 \qquad \mathtt{isCompatible}(\mathtt{T}'_1, \mathtt{T}_1)}{\Gamma \vdash \mathtt{e}_1\ \mathtt{e}_2 : \mathtt{T}_2} \quad \text{(T-App)}$$

Type abstraction and type application have typical System F rules, but again, type variable shadowing is not allowed:

$$\frac{\mathtt{X} \notin \Gamma \qquad \Gamma, \mathtt{X} \vdash \mathtt{e} : \mathtt{T}}{\Gamma \vdash \Lambda \mathtt{X}\ .\ \mathtt{e} : \mathtt{forall}\ \mathtt{X}\ .\ \mathtt{T}} \quad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash \mathtt{e}_1 : \mathtt{forall}\ \mathtt{X}\ .\ \mathtt{T}_1}{\Gamma \vdash \mathtt{e}_1\ [\mathtt{T}_2] : [\mathtt{X} \mapsto \mathtt{T}_2]\mathtt{T}_1} \quad \text{(T-TApp)}$$

$[\mathtt{X} \mapsto \mathtt{T}_2]\mathtt{T}_1$ above means that all occurences of the type variable $\mathtt{X}$ in $\mathtt{T}_1$ are substituted with $\mathtt{T}_2$. Throughout this text we assume that such substitutions are *capture-avoiding* [Pierce, 2002]. Unfortunately, the current MIL implementation is naive in this respect and cannot correctly handle all the cases of substitution.

The next four rules specify how data constructors get their types:

$$\frac{\Gamma \vdash \mathtt{C} \in \mathtt{T}}{\Gamma \vdash \mathtt{C} : \mathtt{T}} \quad \text{(T-ConstrNil)}$$

$$\frac{\Gamma \vdash \mathtt{C}\ \mathtt{T}_1 ... \mathtt{T}_n \in \mathtt{T}}{\Gamma \vdash \mathtt{C} : \mathtt{T}_1 \to ... \to \mathtt{T}_n \to \mathtt{T}} \quad \text{(T-Constr)}$$

$$\frac{\Gamma \vdash \mathtt{C} \in \mathtt{T}\ \mathtt{X}_1 ... \mathtt{X}_n}{\Gamma \vdash \mathtt{C} : \mathtt{forall}\ \mathtt{X}_1 .\ ...\ .\mathtt{forall}\ \mathtt{X}_n.\mathtt{T}\ \mathtt{X}_1 ... \mathtt{X}_n} \quad \text{(T-ConstrNilTypeVars)}$$

$$\frac{\Gamma \vdash \mathtt{C}\ \mathtt{T}_1 ... \mathtt{T}_n \in \mathtt{T}\ \mathtt{X}_1 ... \mathtt{X}_n}{\Gamma \vdash \mathtt{C} : \mathtt{forall}\ \mathtt{X}_1 .\ ...\ .\mathtt{forall}\ \mathtt{X}_n.\mathtt{T}_1 \to ... \to \mathtt{T}_n \to \mathtt{T}\ \mathtt{X}_1 ... \mathtt{X}_n} \quad \text{(T-ConstrTypeVars)}$$

In the rules above $\Gamma \vdash \mathtt{C} \in \mathtt{T}$ means that "a data constructor $\mathtt{C}$ has been defined in type $\mathtt{T}$".

Typing of tuples is specified with the following two rules:

$$\Gamma \vdash \{\} : \{\} \qquad \text{(T-EmptyTuple)}$$

$$\frac{\texttt{for each i} \qquad \Gamma \vdash \mathtt{e_i} : \mathtt{T_i}}{\Gamma \vdash \{\mathtt{e_{i=1..n}}\} : \{\mathtt{T_{i=1..n}}\}} \qquad \text{(T-Tuple)}$$

Probably the most important typing rule is the one for monadic `bind`:

$$\frac{\begin{array}{c}\mathtt{x} \notin \Gamma \qquad \Gamma \vdash \mathtt{e_1} : \mathtt{M_1\ T'_1} \qquad \Gamma, \mathtt{x} : \mathtt{T_1} \vdash \mathtt{e_2} : \mathtt{M_2\ T_2} \qquad \mathtt{T_1} \equiv_\alpha \mathtt{T'_1} \\ \mathtt{isMonad(M_1)} \qquad \mathtt{isMonad(M_2)} \qquad \mathtt{isCompatibleMonad(M_2, M_1)}\end{array}}{\Gamma \vdash \mathtt{let\ (x : T_1) \leftarrow e_1\ in\ e_2} : \mathtt{highestEffectMonad(M_1, M_2)\ T_2}} \qquad \text{(T-Let)}$$

The crucial parts are that both $\mathtt{e_1}$ and $\mathtt{e_2}$ should have monadic types. These two monads have to satisfy the `isCompatibleMonad` relation. The type specified in the variable binder ($\mathtt{T_1}$) and the result type of $\mathtt{e_1}$ ($\mathtt{T'_1}$) must be alpha-equivalent (equivalent modulo renaming of type variables). Alpha-equivalence is used here and in other similar situations because MIL has universally quantified types, which introduce type variables. This can result in the types which have the same meaning (and thus need to be considered equivalent), but happen to use different type variables. The $\mathtt{e_2}$ expression gets the bound variable in scope. The monad for the type of the whole bind expression is chosen using the `highestEffectMonad` function. The rule also specifies that `bind` does not allow variable shadowing.

Monadic `return` typing rule is quite minimal. Its type is the monadic type `return` is annotated with applied to the type of the expression that is being returned:

$$\frac{\mathtt{isMonad(M)} \qquad \Gamma \vdash \mathtt{e} : \mathtt{T}}{\Gamma \vdash \mathtt{return\ [M]\ e} : \mathtt{M\ T}} \qquad \text{(T-Return)}$$

The next rule specifies the typing of the `lift` operation. This operation is annotated with two monads: we lift a computation in monad $\mathtt{M_1}$ to monad $\mathtt{M_2}$. The monad of the expression `e` that we are lifting ($\mathtt{M'_1}$) and monad $\mathtt{M_1}$ should satisfy the non-commutative version of the `isCompatibleMonad` relation. $\mathtt{M_1}$ also has to be a *monad suffix* of $\mathtt{M_2}$.

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{e} : \mathtt{M'_1\ T} \qquad \mathtt{isMonad(M'_1)} \qquad \mathtt{isMonad(M_1)} \qquad \mathtt{isMonad(M_2)} \\ \mathtt{isCompatibleMonadNotCommut(M'_1, M_1)} \qquad \mathtt{isMonadSuffix(M_1, M_2)}\end{array}}{\Gamma \vdash \mathtt{lift\ [M_1 \Rightarrow M_2]\ e} : \mathtt{M_2\ T}} \qquad \text{(T-Lift)}$$

The last rule describes `let rec` expression:

$$\frac{\begin{array}{c}\texttt{for each i} \qquad \mathtt{x_i} \notin \Gamma \qquad \Gamma, (\mathtt{x_j} : \mathtt{T_j})_{\mathtt{j=1..n}} \vdash \mathtt{e_i} : \mathtt{T'_i} \qquad \Gamma, (\mathtt{x_j} : \mathtt{T_j})_{\mathtt{j=1..n}} \vdash \mathtt{e} : \mathtt{T} \\ \mathtt{T_i} \equiv_\alpha \mathtt{T'_i}\end{array}}{\Gamma \vdash \mathtt{let\ rec\ (x_i : T_i) \leftarrow e_{i;i=1..n}\ in\ e} : \mathtt{T}} \qquad \text{(T-LetRec)}$$

Similarly to other expressions which introduce variables, `let rec` does not allow variable shadowing. All binding expressions ($\mathtt{e_i}$) are checked with all the variable binders in scope, so that they

can be mutually recursive. The type of a binding expression should be alpha-equivalent to the type specified in the corresponding variable binder.

Typing of `case` expressions is quite involved when written using judgement rules, so we will omit it here. It can be informally described as the following: pattern types should match the type of a scrutinee, every case alternative is checked separately, variables bound in patterns are in scope for the corresponding alternative. The types of expressions in alternatives should satisfy the `isCompatible` relation. The effect of the case expression is chosen using the `highestEffectMonad` function among all the alternatives.

4.4.1 Monads and relations

Next, we will define what are the possible monads and how they can be combined in MIL as well as the relations and helper functions used in the typing rules above.

There are four built-in monads in MIL: `Id` (identity), `State`, `IO` (for input/output) and `Error`. They all satisfy the `isSingleMonad` relation. Note that the `Error` type has an additional type parameter for the type of error values.

$$\texttt{isSingleMonad}(\texttt{Id})$$

$$\texttt{isSingleMonad}(\texttt{State})$$

$$\texttt{isSingleMonad}(\texttt{IO})$$

$$\texttt{isSingleMonad}(\texttt{Error T})$$

`isMonad` unary predicate defines what is considered a monad in MIL. Single monad is one such case:

$$\frac{\texttt{isSingleMonad}(\texttt{M})}{\texttt{isMonad}(\texttt{M})}$$

There is also an infix type constructor ::: that combines two monads. We call ::: a *monad cons* operator, similarly to list cons cells. This is the way to combine effects in MIL. One can look at it as a type-level list. What it represents is a monad transformer stack. Note, that in MIL there is no distinction between the `State` monad and the `StateT` monad transformer. It is the context that determines the meaning. When a monad is to the left of monad cons, it is considered a transformer, when it is to the right or is used as a type constructor elsewhere, it is a monad. When talking about MIL these terms can be used interchangeably. In line with this we also may (rather informally) use "combining/composing monads" for the sake of brevity, however what we actually mean is "combining monad transformers".

The monad cons operator should be thought of as right-associative. The following rule defines that $\texttt{M}_1$ ::: $\texttt{M}_2$ is a monad, if $\texttt{M}_1$ is a single monad and $\texttt{M}_2$ is a monad (so it can be a monad cons as well):

$$\frac{\texttt{isSingleMonad}(\texttt{M}_1) \qquad \texttt{isMonad}(\texttt{M}_2)}{\texttt{isMonad}(\texttt{M}_1 \texttt{ ::: } \texttt{M}_2)}$$

The following example gives an intuition with relation to monad transformers in Haskell (`State` in MIL does not have a type of storage as opposed to `StateT` in Haskell, so it is substituted with `()`):

```
Error Int ::: (State ::: IO) ⇒ ErrorT Int (StateT () IO)
```

For the sake of the definitions below, we also define $\texttt{isMonadCons}(\texttt{M})$ to hold if $\texttt{M}$ is a combination of two monads with :::. We also use `monadConsLeft` and `monadConsRight` functions to get the left-hand side and the right-hand side of a monad cons respectively.

One of the most important high-level relations is the `isCompatible` relation, which is used in typing of function applications and function bodies, as an example. In general, we can view the `isCompatible` relation in MIL as a subtyping relation extended to monads and their combinations with monad cons. It tries to capture when a value of one type can be used as a value of another type without violating the type safety [Pierce, 2002] (meaning that a program will not have run-time type errors and that one cannot "hide" program effects in any way).

If both of the types are monadic, then a separate (non-commutative) relation for monads is used:

$$\frac{\texttt{isMonad}(\texttt{M}_1) \qquad \texttt{isMonad}(\texttt{M}_2) \qquad \texttt{isCompatibleMonadNotCommut}(\texttt{M}_1, \texttt{M}_2)}{\texttt{isCompatible}(\texttt{M}_1, \texttt{M}_2)}$$

For type applications involving monads, we use the `isCompatible` relation recursively for both parts of the type application:

$$\frac{\texttt{isMonad}(\texttt{M}_1) \qquad \texttt{isMonad}(\texttt{M}_2) \qquad \texttt{isCompatible}(\texttt{M}_1, \texttt{M}_2) \qquad \texttt{isCompatible}(\texttt{T}_1, \texttt{T}_2)}{\texttt{isCompatible}(\texttt{M}_1\ \texttt{T}_1, \texttt{M}_2\ \texttt{T}_2)}$$

For function types, using the terminology from subtyping, we can say that they are covariant in the result types and contravariant in the argument types:

$$\frac{\texttt{isCompatible}(\texttt{T}_{21}, \texttt{T}_{11}) \qquad \texttt{isCompatible}(\texttt{T}_{12}, \texttt{T}_{22})}{\texttt{isCompatible}(\texttt{T}_{11} \to \texttt{T}_{12}, \texttt{T}_{21} \to \texttt{T}_{22})}$$

For universally quantified types we recurse down the types under `forall`:

$$\frac{\texttt{isCompatible}(\texttt{T}_1, [\texttt{Y} \mapsto \texttt{X}]\texttt{T}_2)}{\texttt{isCompatible}(\texttt{forall X . T}_1, \texttt{forall Y . T}_2)}$$

Note that we need to make sure that the type variables `X` and `Y` are considered equivalent further down when checking the types $\texttt{T}_1$ and $\texttt{T}_2$ for compatibility. We achieve this by using the capture-avoiding substitution.

For all the other cases type compatibility is subtyping:

$$\frac{\mathtt{T}_1 <: \mathtt{T}_2}{\mathtt{isCompatible}(\mathtt{T}_1, \mathtt{T}_2)}$$

Subtyping in MIL is defined as just alpha-equivalence for all the types except the tuple types. Tuple types in MIL have *width* and *depth subtyping*. We will not present these rules here, they can be found, for example, in [Pierce, 2002]. Tuples with subtyping are introduced into MIL to provide a good support for implementing subtyping in source languages.

The core of determining whether two monads are compatible is the following non-commutative operation:

$$\frac{\mathtt{isSingleMonad}(\mathtt{M}_1) \quad \mathtt{isSingleMonad}(\mathtt{M}_2) \quad \mathtt{M}_1 \equiv_\alpha \mathtt{M}_2}{\mathtt{isCompatibleMonadNotCommut}(\mathtt{M}_1, \mathtt{M}_2)}$$

$$\frac{\begin{array}{c}\mathtt{isMonadCons}(\mathtt{M}_1) \quad \mathtt{isMonadCons}(\mathtt{M}_2) \quad \mathtt{monadConsLeft}(\mathtt{M}_1) \equiv_\alpha \mathtt{monadConsLeft}(\mathtt{M}_2) \\ \mathtt{isCompatibleMonadNotCommut}(\mathtt{monadConsRight}(\mathtt{M}_1), \mathtt{monadConsRight}(\mathtt{M}_2))\end{array}}{\mathtt{isCompatibleMonadNotCommut}(\mathtt{M}_1, \mathtt{M}_2)}$$

$$\frac{\mathtt{isSingleMonad}(\mathtt{M}_1) \quad \mathtt{isMonadCons}(\mathtt{M}_2) \quad \mathtt{M}_1 \equiv_\alpha \mathtt{monadConsLeft}(\mathtt{M}_2)}{\mathtt{isCompatibleMonadNotCommut}(\mathtt{M}_1, \mathtt{M}_2)}$$

It specifies that two single monads are compatible if they are alpha-equivalent. Alpha-equivalence for MIL monads is pretty straight-forward: every monad is alpha-equivalent to itself. In the case of two `Error T`, their type arguments denoting the error types also must be alpha-equivalent. Another case is for two monad conses: their left-hand sides must be alpha-equivalent and then the recursive cases on the right-hand sides must hold as well. Finally, a single monad is compatible with a monad cons if it is alpha-equivalent to the left-hand side of the monad cons. One can think about this relation as the one that checks whether the first argument is a proper *prefix* of the second one. As an example, `State ::: Error Unit` is compatible with `State ::: (Error Unit ::: IO)`. It is also possible to think of every monad cons sequence as having an implicit monad variable `M` at the end, similarly to monad transformer stacks parameterised over the underlying monad in Haskell. An example of incompatible monads is `State ::: Error Int` and `Error Int ::: State`, because the order of `State` and `Error` is different. Another example is `State ::: IO` is incompatible with just `State`, because the first one has more effects than the second one and thus cannot be passed as an argument instead of a just stateful computation and cannot be a body of a function that declares only `State` as its effect.

`isCompatibleMonad` is a commutative version of the previous relation:

$$\frac{\mathtt{isCompatibleMonadNotCommut}(\mathtt{M}_1, \mathtt{M}_2)}{\mathtt{isCompatibleMonad}(\mathtt{M}_1, \mathtt{M}_2)}$$

$$\frac{\mathtt{isCompatibleMonadNotCommut}(\mathtt{M}_2, \mathtt{M}_1)}{\mathtt{isCompatibleMonad}(\mathtt{M}_1, \mathtt{M}_2)}$$

Having defined the monad compatibility, it is worth looking back at the compatibility of function types. Intuitively, if $\texttt{isCompatible}(\texttt{T}_1, \texttt{T}_2)$, $\texttt{T}_1$ has at most the effects of $\texttt{T}_2$, potentially less, but not more. Since function types are "covariant" in the result types, we can pass as an argument a function which returns a computation with less effects than the specified parameter type. Also, since function types are "contravariant" in the argument types, we can pass a function, which has a parameter with a more effectful type. For example, a function of type $(\texttt{State} ::: \texttt{IO})\ \texttt{Int} \to \texttt{State Int}$ can be passed as an argument to a function which has a parameter of type $\texttt{State Int} \to (\texttt{State} ::: \texttt{IO})\ \texttt{Int}$.

In the typing rule for the `lift` operation above, the relation `isMonadSuffix` was used. The intuition behind it is that it specifies whether it is possible to properly put a combination of monads on top of another monad transformer stack. Single monad $\texttt{M}_1$ is a suffix of a single monad $\texttt{M}_2$ if they are alpha-equivalent. In this case lifting is a no-op and the monad of a computation is not changed:

$$\frac{\texttt{isSingleMonad}(\texttt{M}_1) \qquad \texttt{isSingleMonad}(\texttt{M}_2) \qquad \texttt{M}_1 \equiv_\alpha \texttt{M}_2}{\texttt{isMonadSuffix}(\texttt{M}_1, \texttt{M}_2)}$$

Next, if both arguments are monad conses, to satisfy `isMonadSuffix` they can be either alpha-equivalent or the first monad cons is a suffix of the right-hand side of the second monad cons:

$$\frac{\texttt{isMonadCons}(\texttt{M}_1) \qquad \texttt{isMonadCons}(\texttt{M}_2) \qquad \texttt{M}_1 \equiv_\alpha \texttt{M}_2}{\texttt{isMonadSuffix}(\texttt{M}_1, \texttt{M}_2)}$$

$$\frac{\texttt{isMonadCons}(\texttt{M}_1) \qquad \texttt{isMonadCons}(\texttt{M}_2) \qquad \texttt{isMonadSuffix}(\texttt{M}_1, \texttt{monadConsRight}(\texttt{M}_2))}{\texttt{isMonadSuffix}(\texttt{M}_1, \texttt{M}_2)}$$

Given the above it is possible to `lift` a $\texttt{State} ::: \texttt{IO}$ computation into a $\texttt{Error Unit} ::: (\texttt{State} ::: \texttt{IO})$ computation. It is basically putting `Error Unit` on the top of the stack.

The third case is when the first argument is a single monad and the second argument is a monad cons. In this case we just shift and check if the single monad is a suffix of the right-hand side of the monad cons:

$$\frac{\texttt{isSingleMonad}(\texttt{M}_1) \qquad \texttt{isMonadCons}(\texttt{M}_2) \qquad \texttt{isMonadSuffix}(\texttt{M}_1, \texttt{monadConsRight}(\texttt{M}_2))}{\texttt{isMonadSuffix}(\texttt{M}_1, \texttt{M}_2)}$$

An example here is lifting from `IO` to $\texttt{State} ::: \texttt{IO}$.

This was the last rule, which implies that monad cons is never a suffix of a single monad.

Finally, `highestEffectMonad` is a function that takes two monads and returns the one, which encodes more effects. An important internal assumption in MIL is that the `highestEffectMonad` is used only on compatible monads (see `isCompatibleMonad`). We define that monads combined with monad cons have a higher effect than a single monad:

$$\frac{\texttt{isMonadCons}(\texttt{M}_1) \qquad \texttt{isSingleMonad}(\texttt{M}_2)}{\texttt{highestEffectMonad}(\texttt{M}_1, \texttt{M}_2) = \texttt{M}_1}$$

$$\frac{\texttt{isSingleMonad}(\texttt{M}_1) \qquad \texttt{isMonadCons}(\texttt{M}_2)}{\texttt{highestEffectMonad}(\texttt{M}_1, \texttt{M}_2) = \texttt{M}_2}$$

For two monads, `highestEffectMonad` returns the first one, so we do not have any ordering between the MIL monads:

$$\frac{\texttt{isSingleMonad}(\texttt{M}_1) \qquad \texttt{isSingleMonad}(\texttt{M}_2)}{\texttt{highestEffectMonad}(\texttt{M}_1, \texttt{M}_2) = \texttt{M}_1}$$

The most interesting case is when both arguments are monad conses. In this case we recurse into the right hand-sides of monad conses, since that is where they might differ:

$$\frac{\begin{array}{c}\texttt{isMonadCons}(\texttt{M}_1) \qquad \texttt{isMonadCons}(\texttt{M}_2) \\ \texttt{highestEffectMonad}(\texttt{monadConsRight}(\texttt{M}_1), \texttt{monadConsRight}(\texttt{M}_2)) = \texttt{monadConsRight}(\texttt{M}_1)\end{array}}{\texttt{highestEffectMonad}(\texttt{M}_1, \texttt{M}_2) = \texttt{M}_1}$$

$$\frac{\begin{array}{c}\texttt{isMonadCons}(\texttt{M}_1) \qquad \texttt{isMonadCons}(\texttt{M}_2) \\ \texttt{highestEffectMonad}(\texttt{monadConsRight}(\texttt{M}_1), \texttt{monadConsRight}(\texttt{M}_2)) = \texttt{monadConsRight}(\texttt{M}_2)\end{array}}{\texttt{highestEffectMonad}(\texttt{M}_1, \texttt{M}_2) = \texttt{M}_2}$$

One can say that the intuition behind `highestEffectMonad` is that the longer chain of monad conses has a higher effect.

Looking back at the T-Let typing rule again: it does not matter if it is a binding expression or a body which has a higher effect, they must be compatible (using the `isCompatibleMonad` relation) and the type with the highest effect is chosen as the type of the whole expression. The same applies to `case` expressions.

4.5 Haskell implementation

Since one of the aims of this thesis was to produce a programming framework for working with the designed monadic intermediate language, here we will outline some of the implementation details, namely, several core data types.

MIL is implemented in Haskell. There is a parser which given a program text produces a source version of the abstract-syntax tree (AST) and a type checker which given a source version of the AST produces a type annotated version of it. There is also a lint checker, which given an already type annotated AST verifies that it is well-typed. This is really useful as a tool to check that subsequent program transformations maintain its type correctness.

As it was mentioned above, we draw a distinction between the source AST and the type annotated AST. The source version looks more like what a user entered (or rather, a compiler generated) as a program. It is still explicitly typed, meaning, for example, that functions have type signatures and variable binders have types specified. The type annotated version contains more information: some syntax nodes are more refined and are annotated with types.

The biggest difference between the two is in representations of types. The source representation produced by the parser is captured in the `SrcType` data type:

```
data SrcType
  = SrcTyTypeCon TypeName
  | SrcTyArrow SrcType SrcType
  | SrcTyForAll TypeVar SrcType
  | SrcTyApp SrcType SrcType
  | SrcTyTuple [SrcType]
  | SrcTyMonadCons SrcType SrcType
```

It has data constructors for type constructors represented just as their names (`TypeName` is a wrapper for `String`), function types (`SrcTyArrow`), universally-quantified types, type application, tuple types and monad cons, which has `SrcTypes` as its operands. It is important to note that type variables and monads do not have separate data constructors, but are captured by `SrcTyTypeCon` together with type constructors. This is due to the fact that the parser cannot really distinguish them from each other (they are all just identifiers starting with an upper-case letter).

The internal representation of types is expressed as the `Type` ADT, which is slightly more involved. Types are converted from the source representation during the type checking phase. At the same time a number of checks are performed, for example, that all the types are properly kinded, that they use types which are in scope, that there is no type variable shadowing etc.

```
data Type
  = TyTypeCon TypeName
  | TyVar TypeVar
  | TyArrow Type Type
  | TyForAll TypeVar Type
  | TyApp Type Type
  | TyTuple [Type]
  | TyMonad MonadType
```

It has separate constructors for type constructors and for type variables. Function types, universally-quantified types, type applications and tuple types are essentially the same. Monadic types are expressed with the `TyMonad` data constructor, which has one field of type `MonadType`.

Built-in monads are captured in the `MilMonad` data type:

```
data MilMonad
  = Id
  | State
  | Error
  | IO
```

A monadic type in MIL is either a single monad or a single monad combined with another monadic type using monad cons. This is captured in the `MonadType` ADT, mentioned above:

```
data MonadType
  = MTyMonad SingleMonad
  | MTyMonadCons SingleMonad MonadType
```

This data type is essentially a non-empty list of `SingleMonads`.

Single monads are represented using the `SingleMonad` data type:

```
data SingleMonad
  = SinMonad MilMonad
  | SinMonadApp SingleMonad Type
```

It has two data constructors: one for just a built-in monad and another one for application of a single monad to a type. This is done to capture monads that might have additional type parameters. In the current state of MIL one such example is the `Error` monad. To get a monad the `Error` type constructor needs to be applied to a type representing error values.

Expressions in MIL are represented using the parameterised data type `Expr v ct mt t`. It has four type parameters:

- `v` for variable occurences
- `ct` for data constructor types
- `mt` for monads
- `t` for general type occurences

As opposed to types, expressions have only one data type for both source and type annotated representations, hence all the type parameters.

We will provide the definition of the `Expr` data type below to get a general feeling on how different MIL expressions are represented, but we will skip describing it in full detail:

```
data Expr v ct mt t
  = LitE Literal
  | VarE v
  | LambdaE (VarBinder t) (Expr v ct mt t)
  | AppE (Expr v ct mt t) (Expr v ct mt t)
  | TypeLambdaE TypeVar (Expr v ct mt t)
  | TypeAppE (Expr v ct mt t) t
  | ConNameE ConName ct
  | LetE (VarBinder t) (Expr v ct mt t) (Expr v ct mt t)
  | ReturnE mt (Expr v ct mt t)
  | LiftE (Expr v ct mt t) mt mt
  | LetRecE [(VarBinder t, Expr v ct mt t)] (Expr v ct mt t)
  | CaseE (Expr v ct mt t) [CaseAlt v ct mt t]
  | TupleE [Expr v ct mt t]
```

There are two type synonyms: one for the source representation of expressions and one for the type annotated expressions. This is a general pattern used in the MIL implementation for many other data types, like type definitions, constructors definitions, function definitions etc.

```
type SrcExpr = Expr Var () SrcType SrcType
type TyExpr  = Expr TyVarBinder Type MonadType Type
```

In the `Src` case variable occurences are just their names as strings, but in the type annotated case, they are names together with the variable type. `TyVarBinder` is used for this. Data constructor occurences just have `()` as their type in the `SrcExpr`, but an actual type in `TyExpr`. Both of these annotations can be useful when working with an instance of the AST, since it allows to get the types without asking the type environment. For monads and general type occurences, source representation of types (`SrcType`) is replaced with `MonadType` and the internal type representation respectively.

Implementation details specific to optimisations are presented in a separate chapter.

4.6 Discussion

Currently, effects in MIL are quite coarse-grained. For example, compared to MIL-lite by Benton and Kennedy [Benton and Kennedy, 1999], there is only one big `State` and no distinction between reading/writing is made. Input and output are not separated either. Non-termination effect is not captured in MIL.

None of the monadic ILs described in "Related work" had the flexibility of combining effects that is achieved in MIL with the help of monad transformers (which is, of course, partly due to that it wasn't a goal for those ILs to be able to express different semantics). In [Tolmach, 1998] the hierarchy of monads was fixed. In MIL-lite, effects are combined in sets, but there is no way to choose an interpretation of State and Error combination, for example.

When it comes to programming with effects, as it was mentioned several times in "Related work", a neccessity to do lifting when using monad transformers to combine monads is widely considered to be one of their biggest problems. While we agree, that it is a problem when it comes to everyday programming using effects, we think it is not such a big issue for an intermediate language. It is very useful for an IL to be human-readable, but it is usually not the main goal, since it is not a user facing language. Moreover, as it will be seen in the next two chapters, we didn't see lifting as the biggest mental overhead when generating or reading MIL code for the source languages that were designed as part of this work.

Chapter 5

Functional programming language (FunLang)

This chapter introduces the first source programming language which has been designed during this work – FunLang. It starts from the language overview and example programs. Then the code generation to MIL is described. Finally, we make conclusions regarding the implementation of the FunLang compiler using MIL.

5.1 Overview

FunLang is a rather small functional programming language. Its design is mostly inspired by Haskell. It is statically and explicitly typed. The type system of FunLang is based on System F. In addition to the pure polymorphic lambda calculus FunLang incorporates `do`-notation similar to the one found in Haskell and has a couple of built-in monads. It also adds minimal exception handling.

One can say that FunLang is quite similar to MIL in many ways. The main motivation behind designing FunLang as one of the languages for MIL evaluation is to be able to explore compilation of a modern statically typed functional language. Despite the many similarities to MIL, as we will see, the MIL code produced for FunLang programs looks rather different from the programs' source code. In addition to the semantics similar to the one of MIL, FunLang offers cleaner syntax, since it is a user facing language, rather than an intermediate representation. One of the main semantic differences is implicit effects which are present in FunLang programs, as we will see in this chapter.

5.2 FunLang by example

Similarly to the previous chapter, we will provide several examples showcasing the main features of the language.

5.2.1 Data types

As most statically typed functional programming languages, FunLang supports ADTs which can be parameterised. One of the simplest data types is `Bool`, which represents true and false values:

```
type Bool = True | False
```

An example of a parameterised data type often found in functional programming languages is a pair of two values:

```
type Pair A B = MkPair A B
```

Another canonical example of an ADT is a recursive data type representing a binary tree:

```
type Tree A
  = Empty
  | Node A (Tree A) (Tree A)
```

A tree is either empty or it is a node carrying a value of type `A`, which can have two children. Leaf nodes having a value `x` are represented as `Node x Empty Empty`.

5.2.2 Functions and expressions

Functions in FunLang have syntax rather similar to Haskell. One needs to write a function name and its type and then a so-called *function equation* containing the function body:

```
constInt : Int -> Int -> Int
constInt = \(a : Int) (b : Int) -> a;;
```

The idea is that potentially FunLang can be extended to have the same rich syntax for function equations as in Haskell, meaning having multiple of them with parameters being expressed with patterns to the left of the equal sign. This is the reason for having two semicolons at the end of a function definition. Each equation would end with one semicolon and then the whole function would end with one more. Unfortunately, multiple function equations and pattern matching for parameters are not implemented as part of this project.

From the example above one can also see the syntax for lambda expressions in FunLang, which allows to have multiple parameters instead of explicitly nesting several lambdas. Note that variable shadowing is not allowed in FunLang.

Polymorphic functions are defined with the “big lambda”, which also allows to specify several type variables at once. Below are examples of the identity function and function composition:

```
id : forall A . A -> A
id = /\A . \(x : A) -> x;;

compose : forall A . forall B . forall C . (B -> C) -> (A -> B) -> A -> C
compose = /\A B C . \(f : B -> C) (g : A -> B) (x : A) -> f (g x);;
```

Function application is specified with juxtaposition. Square brackets are used for type application:

```
one : Int
one = id [Int] 1;;
```

FunLang has `Unit`, `Bool`, `Pair`, `Int`, `Float`, `Char` and `String` as built-in types. It supports the usual infix arithmetic operations: `+`, `-`, `*` and `/`.

5.2.3 Monads

FunLang provides built-in `IO` and `State` monads, which are similar to the corresponding monads in Haskell. The notation looks very similar to Haskell, except that more things are explicit, like variable bindings, for example, and every *statement* in a `do` block is terminated with a semicolon, while the whole block is terminated with the `end` keyword.

```
main : IO Unit
main = do
  printInt 1;
  i : Int <- readInt;
  printInt i;
  printInt (execState [Int] [Unit] stateFun 1);
  return (evalState [Int] [Unit] stateFun 0);
end;;

stateFun : State Int Unit
stateFun = do
  i : Int <- get [Int];
  put [Int] i;
  modify [Int] (\(s : Int) -> s);
end;;
```

As it can be seen from the previous example, FunLang provides a number of built-in functions for working with `IO` and `State`:

```
printString : String -> IO Unit
readString : IO String
printInt : Int -> IO Unit
```

```
readInt : IO Int
printFloat : Float -> IO Unit
readFloat : IO Float
evalState : forall S . forall A . State S A -> S -> A
execState : forall S . forall A . State S A -> S -> S
runState : forall S . forall A . State S A -> S -> Pair A S
get : forall S . State S S
put : forall S . S -> State S Unit
modify : forall S . (S -> S) -> State S Unit
```

The first six functions are for reading values of several built-in types from the standard input and printing them to the standard output. They work in the `IO` monad. The functions `evalState`, `execState` and `runState` are for running computations in the `State` monad. They all take one such computation and an initial state value. The difference is that `evalState` returns the value of the computation as the result, `execState` returns the value of the state itself and `runState` returns both the value and the state as a pair. The last three functions are for working with state inside a stateful computation. They allow to read the state value, overwrite it and modify it with a function.

It is worth highlighting, that unfortunately, FunLang does not support combining monads in any way. In spite of this property of FunLang, combining monads in MIL is used extensively, as we will describe in the "Code generation" section. Moreover, some features of FunLang, like the ability to run stateful computations with `evalState`, `execState` or `runState` inside of any computation influences the representation of effects in MIL code quite significantly.

5.2.4 Exceptions

FunLang provides rather naive support for exception handling. There are two language constructs for it: `throw` and `catch`. The first one is an expression that allows to raise an exception. Exceptions in FunLang do not carry additional information, so `throw` is used standalone, without any value specified, but in order to assign a type to it, it needs to be annotated with a type. The `catch` construct is a binary infix operator, that tries to evaluate an expression on the left-hand side and if that throws an exception, it returns an expression on the right-hand side. The following example will result in `1`, since the expression on the right is `throw` and the handler expression is `1`:

```
throw [Int] catch 1
```

5.3 Code generation

In this section we will outline how FunLang code is translated into MIL code.

In order to generate FunLang code we needed to decide in which monads different kinds of computations should be expressed. Looking at FunLang, there are three different kinds of code: "pure" code (outside of monads), code inside `IO` and code inside `State`. Pure code is not completely pure,

though. First, functions can be non-terminating. MIL does not have a monad for non-termination, so we do not worry about this particular effect. Second, FunLang exceptions have non-monadic types, but we need to express potential failure in MIL. This means that the `Error` monad should be part of all computations. For stateful computations we need to add `State` and for input/output – IO. The problem here is that FunLang allows to "escape" `State` by using `evalState`, `execState` or `runState` and therefore running stateful computations inside pure or `IO` computations. Escaping `State` is not possible in MIL, so we need to add `State` to all stacks. Given all this, we have two effect stacks: `State ::: Error Unit` for pure computations and computations in the `State` monad and `State ::: (Error Unit ::: IO)` for computations in the `IO` monad. If we do an exercise of expanding types as we did in one of the previous chapters for the `IO` monad stack, we will get `s -> IO (Either Unit (a, s))`. This can be read as "a function that given a state value can perform IO and either results in an error or produces a value and a new state". `Unit` is chosen as a type for error values, because exceptions in FunLang do not carry any value. We choose to have `State` on top of `Error`, but for FunLang different orderings of `Error` and `State` cannot be observed, since `catch` can at most have a stateful computation which has been run with `evalState`, `execState` or `runState` and therefore state cannot escape outside of that computation. One could extend FunLang with the `Error` monad that can be combined with `State` somehow in order to be able to interrupt stateful computations with errors.

Note, that there is an important relation between the pure and the `IO` monad stacks above. They satisfy the `isCompatible` relation in MIL (the former is a prefix of the latter). This allows us to use computations in the pure monad stack inside computations in the `IO` stack.

5.3.1 General scheme and type conversions

In order to generate code for function definitions, a type of every function needs to be converted to the corresponding MIL representation. There were two main considertions here: all effects of FunLang that are possible to express in MIL have to be captured in types and code generation scheme should be as uniform as possible. These considerations together result in the fact that all FunLang code, even expressions not inside monads, becomes monadic code in MIL. In such a scheme every sub-expression has a monadic type and needs to be bound to a variable with `bind` before it can be used. For example, literal occurences result in `returns` and in order to be passed into a function their values need to be extracted from a monad (with `bind`). One can see some resemblance to the ANF in a sense that all expressions are broken down into intermediate simple expressions, like variables and function applications.

Next, we will provide some examples to demonstrate the code generation scheme outlined above.

A FunLang function with the simplest type `Int` can potentially throw an exception instead of returning an integer. This means that its MIL type should be `(State ::: Error Unit) Int`. The same way, when there is a universally-quantified type or a function type, instead of having lambda binders in the function body, it can just have `throw` with an appropriate type annotation. Therefore, there must be a monad in front of every `forall` type and every function parameter type. Below is an example of the identity function in FunLang and then the MIL code generated for it:

```
id : forall A . A -> A
```

```
id = /\A . \(x : A) -> x;;

id : (State ::: Error Unit) (forall A .
       (State ::: Error Unit) (A ->
         (State ::: Error Unit) A)) =
  return [State ::: Error Unit] /\A .
    return [State ::: Error Unit] \(x : A) ->
      return [State ::: Error Unit] x;
```

`IO` type gets translated to the corresponding monad stack described at the beginning of this section. We will consider type conversions for `State` in a separate section.

The next example demonstrates how function and type applications look in the generated code:

```
one : Int
one = id [Int] 1;;

one : (State ::: Error Unit) Int =
  let (var_1 : Int -> (State ::: Error Unit) Int) <-
    let (var_0 : forall A . (State ::: Error Unit)
                   (A -> (State ::: Error Unit) A)) <- id
    in var_0 [Int]
  in let (var_2 : Int) <- return [State ::: Error Unit] 1
  in var_1 var_2;
```

Every sub-expression gets bound to a variable and the code generator produces fresh variable names for all of the intermediate sub-expressions. One can relate nested binds above to the fact that applications are left-associative.

5.3.2 Data types and data constructors

Given the similarity between FunLang and MIL, there is basically almost no translation of data type definitions. There exists one problem with data constructors and their application as functions, though. Every MIL data constructor is introduced as a function in the global scope and therefore, it can be used as a function. Application of data constructors in MIL does not have any effect, which makes them different from all the other function definitions generated from FunLang, which have a monad attached to every argument position. In order to avoid having this special case which can complicate the code generation slightly, we generate a wrapper function for every data constructor. This wrapper function is then used instead of the data constructor occurrence itself.

The following example contains a definition of the `Pair` data type and the corresponding wrapper function for its only data constructor:

```
type Pair A B = MkPair A B;

con_MkPair :
  (State ::: Error Unit) (forall A .
    (State ::: Error Unit) (forall B .
      (State ::: Error Unit) (A ->
        (State ::: Error Unit) (B ->
          (State ::: Error Unit) (Pair A B))))) =
  return [State ::: Error Unit] /\A .
    return [State ::: Error Unit] /\B .
      return [State ::: Error Unit] \(var_1 : A) ->
        return [State ::: Error Unit] \(var_2 : B) ->
          return [State ::: Error Unit]
            MkPair [A] [B] var_1 var_2;
```

What the code generator did in this case was generating an MIL expression from the type of the data constructor. The problem of generating an expression which has a particular type can be really hard for arbitrary types, but in the case of data constructors the shape of possible types is quite restricted.

5.3.3 Built-in types and functions

Most of the FunLang built-in data types map one-to-one to the MIL data types, except for the `String` type, since MIL provides only `Char`. A solution to this is to predefine a `String` data type as the following:

```
type String
  = Empty_Str
  | Cons_Str Char String;
```

This is basically a list of characters. String is either empty or it is a cons cell of a character and another string.

When it comes to built-in functions related to reading and printing built-in data types, MIL provides only `read_char` and `print_char`, therefore FunLang built-in functions like `printString`, `readString` and so on need to be expressed in terms of `read_char` and `print_char`. All these functions have to work inside the `IO` stack. It is very helpful that the `IO` stack contains `Error`, since one needs to throw an exception in case when `readInt` is called and the first character in the input stream is a letter, for example. When using `read_char` and `print_char` lifting needs to be used in order for types to match, since these MIL functions have just `IO` in their types. The following code snippet is the implementation of the `printString` function:

```
printString : (State ::: Error Unit)
                (String -> (State ::: (Error Unit ::: IO)) Unit) =
```

```
return [State ::: Error Unit]
  \(s_ : String) ->
  case s_ of
  | Empty_Str =>
      return [State ::: (Error Unit ::: IO)] unit
  | Cons_Str (c_ : Char) (cs_ : String) =>
      let (unit_0 : Unit) <-
        lift [IO => State ::: (Error Unit ::: IO)] print_char c_
      in let (printString_ : String -> (State ::: (Error Unit ::: IO)) Unit) <-
           printString
         in printString_ cs_
  end;
```

In the actual implementation of FunLang most of the built-in functions, except for `printString` and `readString` are actually just stubs in order to demonstrate the concept without fully-implementing this non-crucial part.

Arithmetic operators map directly to built-in MIL functions:

```
plus : Int
plus = 1 + 2;;

plus : (State ::: Error Unit) Int =
  let (var_0 : Int) <- return [State ::: Error Unit] 1
  in let (var_1 : Int) <- return [State ::: Error Unit] 2
  in return [State ::: Error Unit] add_int var_0 var_1;
```

This is one of the cases when not every sub-expression is bound to a variable, since the code generator knows the type of the `add_int` function when generating code for addition.

A more interesting case is division. Since it is rather easy for division to fail, namely, when using 0 as a divisor, `div_int` and `div_float` functions in MIL have `Error` in their types. We need to use the `lift` operation in this case to get `State` on top:

```
division : Int
division = 1 / 2;;

division : (State ::: Error Unit) Int =
  let (var_0 : Int) <- return [State ::: Error Unit] 1
  in let (var_1 : Int) <- return [State ::: Error Unit] 2
  in lift [Error Unit => State ::: Error Unit] div_int var_0 var_1;
```

5.3.4 State

Code generation for FunLang computations inside the `State` monad makes use of MIL references. The main problem was to provide `State` functions with references to work with. It was solved by

adding an extra parameter of MIL `Ref` type to every `State` function. The example below shows a type of a stateful computation in FunLang and the corresponding MIL type:

```
stateFun : State Int Unit

stateFun : (State ::: Error Unit) (Ref Int -> (State ::: Error Unit) Unit)
```

This implies that the code generator needs to keep track of whether a sub-expression type has a reference as a parameter and pass the reference in scope as an argument. A special name for such reference is used: `state_`. This variable must be introduced by a lambda for every `State` function. The following example shows a simple `State` function that just reads the state value. One can see how the code generator introduced the `state_` reference and then supplied it to the `get` function:

```
stateFun : State Int Unit
stateFun = do
  i : Int <- get [Int];
  return unit;
end;;

stateFun : (State ::: Error Unit) (Ref Int -> (State ::: Error Unit) Unit) =
  return [State ::: Error Unit] \(state_ : Ref Int) ->
    let (i : Int) <-
      let (var_0 : forall S_ . (State ::: Error Unit)
                      (Ref S_ -> (State ::: Error Unit) S_)) <-
        get
      in let (var_1 : Ref Int -> (State ::: Error Unit) Int) <-
           var_0 [Int]
      in var_1 state_
    in let (var_2 : Unit) <- return [State ::: Error Unit] unit
    in return [State ::: Error Unit] var_2;
```

Built-in `State` functions such as `evalState`, `execState`, `runState`, `get`, `put` and `modify` are implemented in MIL and this code is emitted at the beginning of every program together with other built-in functions and data types. The function `evalState` creates a reference with the initial state value that it is given and runs a given `State` computation with this reference as an argument. The same happens in `execState`, but in addition it reads the reference to get the final state value out and returns it. The `runState` function does everything `execState` does, but then also constructs a pair with both the result and the final state value and returns this pair. The `get` and `put` functions are wrappers around `read_ref` and `write_ref` respectively. The `modify` operation is a more high-level combination of these two with a given state transformation function applied in between.

5.3.5 Exceptions

When it comes to code generation for FunLang exception handling, the `throw` expression maps almost directly to the `throw_error` function in MIL. Its first type parameter always gets `Unit`,

since no other values can be thrown in FunLang. Its second type parameter is the type annotation on `throw` converted to the corresponding MIL type. And finally, the error value itself is `unit`. The `lift` operation to put `State` on top also needs to be generated, since `throw_error` only has `Error` in its type.

Dealing with `catch` turned out to be a much bigger problem. When presenting MIL built-in functions in the previous chapter, we omitted the `catch_error` function, which one would definitely expect to see there. One possible type of `catch_error` could be `forall E . forall A . Error E A -> (E -> Error E A) -> Error E A`, where the first parameter (after the two type parameters) corresponds to the left-hand side of FunLang `catch` and the second parameter corresponds to the right-hand side wrapped in a lambda, which takes a `Unit` placeholder. The problem with this type and the design of FunLang is that in FunLang exceptions can be thrown and handled both inside pure computations and inside `IO` computations, which corresponds to two different monad stacks. So there are two possible types of computations that we need to pass to `catch_error` and none of those match the type above. As was presented in the previous chapter, a type of the argument to a function must satisfy the `isCompatible` relation with the corresponding parameter type. If we say that instead of `Error E A` the first parameter of `catch_error` could be `State ::: (Error E ::: IO) A` (as well as all the other occurences of `Error E A`), then we could pass a pure/stateful FunLang computation of type `(State ::: Error E) A` as an argument (it has less effects than `catch_error` expects). But if we were to apply `catch_error` with IO in its type inside a function of type `(State ::: Error E) A`, the MIL type/lint checker would give an error, since the result would have had more effects than specified by the function type. If we instead say that `catch_error` should have `(State ::: Error E) A` instead of `Error E A` in its type, then we cannot pass an IO computation as the first argument, since its type is not compatible (it has more effects than `catch_error` expects).

For this reason, MIL provides two functions: `catch_error_1` and `catch_error_2` as a workaround. The types of these functions are not fully specified in MIL itself. There are placeholders for monads, which a source language compiler needs to fill in, when using the MIL type/lint checker. FunLang puts the pure monad stack and the `IO` stack respectively and then uses the corresponding versions of `catch_error` depending on the context, in which `catch` expression is used. This solution is, of course, not sustainable and can support up to two possible monad stacks in a similar case, when exceptions can be thrown in all the contexts in the source language.

What one would really want to capture in the type of `catch_error` here is the fact that it expects a computation that *has* `Error` as one of its effects. But what `Error E A` means in MIL is that a computation has *just* the `Error` effect, not less and not more. Unfortunately, MIL does not offer an ability to express the fact that a computation "has at least this effect, but maybe more".

5.4 Conclusions

In general, we can say that code generation from FunLang to MIL is rather straight-forward, because FunLang and MIL are quite close to each other. The main difficulties were to get the type conversions right as well as working with references for `State`.

The code generator also used one small practical trick in order to avoid potential collisions with different names (function, variable and type names) in FunLang code when generating fresh names. FunLang's lexer does not allow underscores in those names, while MIL does allow them, so all the generated names contained underscores in them, which eliminates the possibility of collisions with names in source programs.

When looking at the MIL code generated for pure FunLang code, which is actually pure and not only specifies that in the type (meaning when it does not throw exceptions), one could see that the size of the generated code is significantly bigger than it could have been if it was non-monadic. A number of code transformations that have been implemented as part of this work allow to simplify generated programs and remove a lot of extraneous code. This will be described in detail in the chapter on optimisations. To go further and eliminate all the unnecessary returns and binds, remove monads from the types as well as introduce opportunities for more optimisations, one would need to implement some kind of *effect inference*. Such a process could analyse the code and infer that it does not have the specified effects and then rewrite the code, simplifying it.

One could quite easily extend FunLang with more powerful features like `case` expressions or combining monads with transformers, for example, because MIL is a rather expressive language and has those features available out of the box.

We think that the problem described in the section on code generation for exceptions might be the biggest current MIL limitation.

Chapter 6

Object-oriented programming language (OOLang)

In this chapter we will look at another source programming language – OOLang. Language overview, design principles and example programs are presented. Then we describe interesting parts of the MIL code generation. At the end, conclusions regarding implementation of the OOLang compiler using MIL are drawn.

6.1 Overview

OOLang is an object-oriented programming language. The design of OOLang is mainly inspired by three programming languages: C#, Haskell and Ruby. Most of the semantics is rather similar to the one of C#. Syntax is inspired by both Haskell and Ruby. Some crucial language design ideas and decisions as well as some general influence comes from Haskell.

OOLang offers a separation between pure and impure functions on the type level, encourages immutability and tries to eliminate one of the most common problem of the modern OO-languages: null references.

6.2 OOLang by example

This section will introduce most of the OOLang features using example programs.

6.2.1 Functions, statements and expressions

OOLang allows to define functions consisting of several statements. Functions can be pure or impure, which is captured in the return type. By default all functions are impure. Impurity in OOLang comes from input/output and assignments to references. Function definitions can contain parameters in curly braces with a function arrow (->) separating several parameters (and a

return type), which means that functions are curried. Note that in OOLang it is not possible to define a function, which would take several arguments at once (one way of supporting this could be to have tuples in the language and pass several arguments as a tuple). Function application is expressed using juxtaposition. Unlike many other programming languages with statements, there is no `return` statement in OOLang. Instead, every statement has a value (and a type) and so the value of the last statement is what is returned from a function.

OOLang has `Unit` (the type of the `unit` literal), `Bool` (which has literals `true` and `false` as possible values), `Int`, `Float`, `Char` and `String` as built-in types. It supports the usual infix arithmetic operations: `+`, `-`, `*` and `/`.

The following example demonstrates some of the features described above:

```
def main : Unit
  fun 1;
  unit;
end

def fun : {a : Int} -> {b : Float} -> Pure Int
  a;
end
```

OOLang supports higher-order functions:

```
def main : Unit
  applyInt idInt 1;
  unit;
end

def applyInt : {f : Int -> Pure Int} -> {x : Int} -> Pure Int
  f x;
end

def idInt : {x : Int} -> Pure Int
  x;
end
```

At the moment, there are no lambda expressions in OOLang, but given that the target for the language is MIL, it should be rather straight-forward to add them.

OOLang provides a conditional `when-otherwise` statement, which corresponds to `if-else` in most other languages. Both branches have to have the same type:

```
when b do
  1;
```

```
otherwise
  2;
end;
```

Exception handling is performed using `try-catch-finally`. Exceptions can be thrown with the `throw` statement, which is annotated with a type, since as it was mentioned earlier, every statement in OOLang must have a type. The types of `try` and `catch` block must be the same (or rather satisfy the subtyping relation, which we will present later in this chapter). The type of `finally` does not matter, since its value is not returned.

```
try
  throw [Int];
  1;
catch
  2;
finally
end;
```

OOLang provides a number of built-in functions for reading and printing values of built-in data types:

```
printString : String -> Unit
readString : String
printBool : Bool -> Unit
readBool : Bool
printInt : Int -> Unit
readInt : Int
printFloat : Float -> Unit
readFloat : Float
```

6.2.2 Mutability

By default all variables in OOLang are immutable. To declare a mutable variable a special type constructor `Mutable` is used. The next example demonstrates a declaration of an immutable and a `Mutable` variable and the `Mutable` assignment operator:

```
def assignments : Pure Unit
  x : Int = 0;
  y : Mutable Int <- 0;
  y <- 1;
end
```

When variables are declared in OOLang, they must be initialised. There is an exception to this rule involving the `Maybe` type, which will be described later in this section.

`Mutable` variables never escape the scope of the function they are declared in, therefore assignments to `Mutable` variables are considered pure, since they cannot change any global state outside of the function. Given this, `Mutable` variables are still quite useful, especially if OOLang is extended with loop constructs. It is worth highlighting that OOLang `Mutable` variables are somewhat similar to the `mutable` variables in F#.

6.2.3 References

OOLang also supports references, which are quite similar to ordinary variables in C# (for reference, not value, types) or Java. But there is a slight difference, which makes OOLang references remind of C pointers, namely that one can put a new value into the same reference cell (which is not possible in Java or C#, where one can just mutate the value or make a reference variable point to another value). There is a `ref` operator for creating new references, `:=` for assigning values to references and `!` for reading reference values. The following example demonstrates these operators and a declaration of a `Ref` variable (note that `=` is used for initialisation):

```
def references : Unit
  x : Ref Int = ref 0;
  x := 1;
  y : Int = !x;
end
```

`Ref` variables (unlike `Mutable` variables) can be passed around, so working with references is considered impure.

6.2.4 Maybe

Variables in OOLang (unlike C# or Java) cannot be `null`. Instead, a built-in `Maybe` type found in many functional programming languages is used. There are two literal values for the `Maybe` type: `just` which is used together with some value and `nothing` which has to be annotated with a type:

```
x : Maybe Int = just 1;
y : Maybe Int;
z : Maybe Int = nothing [Maybe Int];
```

There is a binary operator `??` (double question mark), which we call “nothing coalesce” (similar to the “null coalesce” operator in C#). It allows in one expression to check whether the left operand is `nothing` and if it is, the value of the right operand is returned, otherwise the value under `just` in the left operand itself is returned:

```
nothing [Maybe Int] ?? 0  # Evaluates to 0
```

As was mentioned above, variables of type `Maybe` can be left uninitialised. In this case they will get the value `nothing` (which will be possible to change later on, but only for `Mutable` and `Ref` variables):

```
def assignmentsMaybe : Pure Unit
  x : Mutable (Maybe Int);
  x <- just 1;
end

def referencesMaybe : Pure Unit
  x : Ref (Maybe Int);
end
```

`Ref` declaration without initialisation is the only case in OOLang, when working with `Ref` variables is considered to be pure.

6.2.5 Classes

OOLang supports classes with inheritance similar to C# and Java. By default all methods are virtual, meaning that they can be overriden. Overloading of methods is not supported. In addition to methods, classes can also contain data fields. Class methods are basically OOLang functions, but inside a class definition and with access to class data fields.

The following example contains a simple class hierarchy consisting of two classes with fields and a method, which is overriden in the subclass:

```
class Parent
  parentField : Int = 1;

  def method : {x : Int} -> Pure Int
    self.parentField;
  end
end

class Child < Parent
  childField : Bool = true;

  def method : {x : Int} -> Pure Int
    when self.childField do
      x;
    otherwise
      super.method x;
```

```
    end;
  end

  def getParent : Pure Parent
    super;
  end
end
```

All class members have to be accessed through a special variable `self` inside the class. Class fields cannot be accessed from the outside of the class. Super class members can be accessed through the `super` variable.

Unfortunately, OOLang does not support `Mutable` class fields in the code generator. We will describe this issue in the "Code generation" section.

The next example demonstrates constructing objects and working with them:

```
parentObj : Parent = Parent.new;
parentObj.method 1;

mChild : Maybe Child = just Child.new;
mParent : Maybe Parent = mChild ? getParent;
```

Object are created using a special construct `ClassName.new`, which does not have any parameters in the current state of OOLang, meaning that OOLang classes do not support custom constructors. Default constructor which initialises the fields is generated. Methods are applied to arguments in the same way as functions. There is a special syntax (`?`) for calling methods on objects wrapped in `Maybe`. If the object variable has `just` value, then the method is called on the underlying object and the result is wrapped in `just`, otherwise, `nothing` is returned. This operator is inspired by the `?.` operator in C# 6.0.

Since OOLang supports inheritance, one of the most interesting parts of its type system is the subtyping relation. It is also used for checking the usage of other types than classes. Note that it is not used for purity checking.

Subtyping in OOLang is reflexive (any type is a subtype of itself):

$$\mathtt{T} <: \mathtt{T}$$

If class `A` inherits from class `B` then `A` is a subtype of `B`:

$$\frac{\texttt{class}\ \mathtt{A} < \mathtt{B}\ldots}{\mathtt{A} <: \mathtt{B}}$$

Subtyping relation is also transitive:

$$\frac{\texttt{A} <: \texttt{B} \qquad \texttt{B} <: \texttt{C}}{\texttt{A} <: \texttt{C}}$$

For `Pure` types the following holds:

$$\frac{\texttt{A} <: \texttt{B}}{\texttt{Pure A} <: \texttt{B}}$$

$$\frac{\texttt{A} <: \texttt{B}}{\texttt{A} <: \texttt{Pure B}}$$

$$\frac{\texttt{A} <: \texttt{B}}{\texttt{Pure A} <: \texttt{Pure B}}$$

So, basically `Pure` is erased and subtyping is checked for the underlying types, meaning that `Pure` types are interchangeable with other types. This is why the subtyping relation is not used for purity checking (a separate mechanism is used).

Mutables are copied, so they can be used in the same places as ordinary types, except for special declaration and assignment operators usage. And this gives us rules similar to the ones for `Pure`:

$$\frac{\texttt{A} <: \texttt{B}}{\texttt{Mutable A} <: \texttt{B}}$$

$$\frac{\texttt{A} <: \texttt{B}}{\texttt{A} <: \texttt{Mutable B}}$$

$$\frac{\texttt{A} <: \texttt{B}}{\texttt{Mutable A} <: \texttt{Mutable B}}$$

`Ref` types are invariant. There is a known problem with mutable references being covariant. An example of this is covariant arrays in several popular programming languages, like Java and C#, which is explained, for example, in [Birka and Ernst, 2004].

`Maybe` type is covariant:

$$\frac{\texttt{A} <: \texttt{B}}{\texttt{Maybe A} <: \texttt{Maybe B}}$$

6.3 Code generation

In this section we will describe the code generation from OOLang to MIL.

The same way as with FunLang, we needed to decide on the monad stacks for different kinds of OOLang computations. As was described above, OOLang has pure and impure computations.

Pure computations do not perform input/output and do not manipulate state (with `Refs`), but they can throw exceptions. Therefore, the pure stack for OOLang is `Error Unit`. Similarly to FunLang, OOLang exceptions do not carry values. Impure computations add `State` and `IO`. We decided to order `Error` and `State` differently from FunLang to *not* get the state rollback semantics, so we have `Error Unit ::: (State ::: IO)` for impure OOLang computations. Again, the pure stack is a prefix of the impure stack (they satisfy the `isCompatible` relation) to be able to easily run pure computations inside impure ones.

6.3.1 General scheme and type conversions

The general approach to code generation and type conversions for OOLang is rather similar to what was described in the FunLang chapter. Every function type needs to be converted to the corresponding MIL type. There is one significant difference in OOLang, though. We know that no effects can happen between partial applications of a function up until the return type, therefore a monad is put only on top of the return type of the function. This is because in FunLang every parameter needs a lambda binder, which could potentially be substituted with `throw`, but in OOLang all parameter binders are already there (using MIL lambda expressions without any effects) and it is only the return type which is determined by the function body and therefore needs to capture potential effects. For return types, `Pure A` results in `Error Unit A` and `A` results in `(Error Unit ::: (State ::: IO)) A`.

Similarly to FunLang, sub-expressions are bound to variables with the monadic `bind`. OOLang statement sequences result in sequences of monadic `bind`s. Since every statement has a value, that value is what is bound to a variable. For some statements it is just a `Unit` variable.

OOLang `when` statements are expressed using the `case` expression to pattern match on the `Bool` condition. The return value of the statement is bound to a variable. The following example tries to capture most of what has been described about functions and statements so far:

```
def fun : {a : Int} -> {b : Bool} -> Unit
  when false do
    1;
    2;
  otherwise
    3;
  end;
  unit;
end

fun : Int -> Bool -> (Error Unit ::: (State ::: IO)) Unit =
  \(a : Int) -> \(b : Bool) ->
    let (var_2 : Int) <-
      let (var_0 : Bool) <-
        return [Error Unit ::: (State ::: IO)] False
      in case var_0 of
```

```
          | True =>
              let (var_1 : Int) <-
                return [Error Unit ::: (State ::: IO)] 1
              in return [Error Unit ::: (State ::: IO)] 2
          | False =>
              return [Error Unit ::: (State ::: IO)] 3
      end
  in return [Error Unit ::: (State ::: IO)] unit;
```

Some of the more specific statements (declarations, assignments, exception handling etc.) as well as class definitions will be covered in separate sub-sections.

One of the challenges in the OOLang code generator was to correctly generate code for variable and function occurences. In order to do this correctly, one needs to realise that there are four different kinds of variable/function occurences in OOLang:

- local variables of value types (non-function, primitive types)
- local variables of function types
- global functions with parameters
- global functions without parameters

For the first three cases, there is no monad at the front of the type. This means that the code generator needs to emit `return` in order to make an expression monadic. In the fourth case, either the pure stack or the impure stack is in the type, so no `return` is needed.

6.3.2 Built-in types and functions

Most of what was described for FunLang built-in types and functions holds for OOLang as well, so we will omit repeating those points and will concentrate on the things specific to OOLang.

Since OOLang provides the built-in `Maybe` type, the following definition for `Maybe` is generated for every OOLang program:

```
type Maybe A
  = Nothing
  | Just A;
```

A difference from FunLang is that `Bool` is a built-in data type with literals, rather than an ADT, so there are built-in functions for printing and reading boolean values. The `Bool` data type itself maps to the `Bool` ADT in MIL. The following code snippet is an implementation of the `printBool` function:

```
printBool : Bool -> (Error Unit ::: (State ::: IO)) Unit =
  \(b_ : Bool) ->
    case b_ of
      | True => printString
          (Cons_Str 't'
          (Cons_Str 'r'
          (Cons_Str 'u'
          (Cons_Str 'e' Empty_Str))))
      | False => printString
          (Cons_Str 'f'
          (Cons_Str 'a'
          (Cons_Str 'l'
          (Cons_Str 's'
          (Cons_Str 'e' Empty_Str)))))
    end;
```

It basically just pattern matches on the argument and outputs a corresponding `String`.

`readBool` is quite a long function, which basically reads characters with `read_char` one by one and pattern matches on them in order to find `true` or `false` as a sequence of characters. In case of failure it uses `throw_error`.

6.3.3 Mutability and references

OOLang has three kinds of variables: default immutable variables, `Mutables` and `Refs`. Declaration statements work similarly for all of them. The code generator emits a monadic `bind` that introduces the variable being declared in scope and the rest of the statements become a body of the `bind` expression and therefore they can access the declared variable.

`Mutable` variables are basically erased during the code generation. The MIL code produced for `Mutables` resembles the SSA/ANF-style code. Every assignment introduces a new variable in scope. This variable has an increasing counter as part of its name. The code generator internally keeps track of what is the current variable name to correctly emit the variable access.

OOLang `Ref` variables are compiled down to the MIL references. There is a direct mapping between the operations on references: `ref` corresponds to `new_ref`, `:=` corresponds to `write_ref` and `!` corresponds to `read_ref`.

The following example demonstrates both `Mutable` and `Ref` declaration, assignment and access (declarations of immutable variables and their occurences look exactly as for `Mutables`, so we omit them):

```
def main : Unit
  a : Mutable Int <- 0;
  a <- 1;
  a;
```

```
  a <- 2;
  a;
  r : Ref Int = ref 0;
  r := 1;
  !r;
  unit;
end

main : (Error Unit ::: (State ::: IO)) Unit =
  let (a : Int) <-
    return [Error Unit ::: (State ::: IO)] 0
  in let (a_1 : Int) <-
    return [Error Unit ::: (State ::: IO)] 1
  in let (var_0 : Int) <-
    return [Error Unit ::: (State ::: IO)] a_1
  in let (a_2 : Int) <-
    return [Error Unit ::: (State ::: IO)] 2
  in let (var_1 : Int) <-
    return [Error Unit ::: (State ::: IO)] a_2
  in let (r : Ref Int) <-
       let (var_6 : Int) <-
         return [Error Unit ::: (State ::: IO)] 0
       in lift [State => Error Unit ::: State] new_ref [Int] var_6
  in let (var_5 : Unit) <-
       let (var_4 : Int) <-
         return [Error Unit ::: (State ::: IO)] 1
       in lift [State => Error Unit ::: State] write_ref [Int] r var_4
  in let (var_3 : Int) <-
       let (var_2 : Ref Int) <-
         return [Error Unit ::: (State ::: IO)] r
       in lift [State => Error Unit ::: State] read_ref [Int] var_2
  in return [Error Unit ::: (State ::: IO)] unit;
```

6.3.4 Exceptions

When it comes to the code generation for exception handling, `try` block corresponds to the first parameter of the `catch_error` function, while `catch` block (with a lambda binder for the `Unit` error value on top) corresponds to the second one. Since all OOLang statements and statement blocks result in MIL expressions, the corresponding expressions are just passed as arguments to `catch_error`.

One interesting consideration here is that we need to make sure that `finally` block is executed after `try` and `catch`. For this reason, we bind the result of `catch_error` to a variable, then bind the result of `finally` and then return the `catch_error` result as the result of the whole statement.

The next example shows a translation for `try-catch-finally` statement:

```
def fun : Pure Unit
  try
    throw [Unit];
  catch
    unit;
  finally
    1;
  end;
end

fun : Error Unit Unit =
  let (var_0 : Unit) <-
    catch_error_1 [Unit] [Unit]
      (throw_error [Unit] [Unit] unit)
      (\(error_ : Unit) -> return [Error Unit] unit)
  in let (var_1 : Int) <- return [Error Unit] 1
  in return [Error Unit] var_0;
```

There is exactly the same problem with `catch_error` type as in the FunLang case, therefore the type checker specifies the appropriate monad stacks for `catch_error_1` and `catch_error_2`.

6.3.5 Classes

In order to represent OOLang classes, MIL tuples are used. Every class maps to a type, which is a tuple with two elements (which are tuples themselves) representing data fields and methods. Since MIL tuples have both width and depth subtyping, new data fields and/or methods can be added in subclasses and it will still be possible to work with a tuple representing a subclass object as if it were a tuple for a superclass object, for example, pass as function arguments, pattern match with the `case` expressions etc. Note that tuples for subclasses contain all of the superclass fields and methods before their own, if any. The inspiration for the representation of OOLang classes comes from [Pierce, 2002].

For every class definition three MIL functions are generated:

- `new_ClassName_Data` function, which we call a *class data constructor function*. This is a function that creates the data part of an object tuple. It contains generated code for field initialisation. The following example contains two class definitions in OOLang and two class data constructor function definitions in MIL:

  ```
  class Parent
    parentField : Int = 1;
    parentField2 : Bool = true;
  ```

```
end

class Child < Parent
  childField : Float = 0.01;
end

new_Parent_Data : Error Unit {Int, Bool} =
  let (self_parentField : Int) <- return [Error Unit] 1
  in let (self_parentField2 : Bool) <- return [Error Unit] True
  in return [Error Unit] {self_parentField, self_parentField2};

new_Child_Data : Error Unit {Int, Bool, Float} =
  let (self_parentField : Int) <- return [Error Unit] 1
  in let (self_parentField2 : Bool) <- return [Error Unit] True
  in let (self_childField : Float) <- return [Error Unit] 1.0e-2
  in return [Error Unit] { self_parentField
                         , self_parentField2
                         , self_childField};
```

Note that the subclass just duplicates the superclass field initialisations. It is also worth highlighting that right now the code generator does not support referencing superclass fields in field initialisers. We think that adding this should not cause any problems.

- `class_ClassName` function, which we call a *class definition function*. This function is where the code for class methods is contained. It takes a class data tuple as a parameter and returns the full class type. Method types are translated as all the other function types with one addition. Method types get an additional parameter of type `Unit`. The reason is that we do not want to evaluate methods at the point of their definition and such a parameter helps to introduce laziness, that is needed in this case. This also means that the code generator needs to supply an additional argument when applying methods. Method definitions need to be able to access `self` and if a class has a superclass, then `super` has to be available as well. The `super` variable is just bound to a constructed instance of the super class. The `self` case is a bit more interesting, since when we are defining methods, we are in the process of defining `self`. This is where the `let rec` expression is used to have a recursive binding for `self`.

  The following example contains a simple class definition function:

```
class Parent
  def method : {x : Int} -> Pure Int
    0;
  end
end

class_Parent : {} -> Error Unit {{}, {Unit -> Int -> Error Unit Int}} =
  \(self_data : {}) ->
```

```
    let rec (self : {{}, {Unit -> Int -> Error Unit Int}}) <-
              { self_data
              , {\(lazy_ : Unit) ->
                   \(x : Int) -> return [Error Unit] 0}}
    in return [Error Unit] self;
```

The next example demonstrates how a class definition for a subclass looks:

```
class Child < Parent
  def method : {x : Int} -> Pure Int
    x;
  end

  def childMethod : Pure Bool
    true;
  end
end

class_Child : {} -> Error Unit
  {{}, { Unit -> Int -> Error Unit Int
       , Unit -> Error Unit Bool}} =
  \(self_data : {}) ->
    let (super : {{}, {Unit -> Int -> Error Unit Int}}) <- new_Parent
    in let rec (self : {{}, { Unit -> Int -> Error Unit Int
                            , Unit -> Error Unit Bool}}) <-
       case super of
       | { super_data : {}
         , super_methods : {Unit -> Int -> Error Unit Int}} =>
         case super_methods of
         | {super_method : Unit -> Int -> Error Unit Int} =>
             { self_data
             , {\(lazy_ : Unit) ->
                  \(x : Int) ->
                    return [Error Unit] x,
                \(lazy_ : Unit) ->
                    return [Error Unit] True}}
         end
       end
    in return [Error Unit] self;
```

Note that in the example above the `method` from superclass is overriden in the subclass, so a new definition is put in its place. In the case, when a subclass does not override a method, a reference to the corresponding superclass method will be used as a method tuple element. The superclass instance is decomposed with a sequence of `case` expressions and all its data fields and methods are available.

One of the most interesting observations to make here is that the subclassing polymorphism is taken care of by storing methods' lambda expressions in a tuple. This tuple is essentially a classical *dispatch table* [Grune et al., 2012]. The ability to capture references to the `self` and `super` objects is powered by *closures* (data structures for storing a function together with its environment, which are heavily used in implementaion of functional programming languages).

One of the main differences of the approach to class and object representation which was taken here compared to the more classical one described, for example, in [Grune et al., 2012], is the access to the `self` object. Usually one would have a special parameter to every method, which represents `self` and would be passed as an argument to every method call. This would allow to avoid a recursive definition and a lazy parameter to every method. Given that some representation of methods' types is part of a class type (most likely via the dispatch table), this approach works well in languages with a type system not as strict as the one in MIL, namely, when having very general pointer or address types to represent method types and being able to cast them. In MIL, having `self` as a parameter in every method would require its type to be in every method type, which would make the class type recursive. One could think solving this with having an ADT instead of a tuple for every class type, but that would require some kind of subtyping for ADTs.

- `new_ClassName` function, which corresponds to the `ClassName.new` construct in OOLang. It basically puts together `new_ClassName_Data` and `class_ClassName`:

```
new_Parent : Error Unit
    { {Int, Bool}
    , {Unit -> Int -> Error Unit Int}} =
  let (self_data : {Int, Bool}) <- new_Parent_Data
  in class_Parent self_data;
```

An interesting caveat of the code generation for class definitions is that it should process classes in order from superclasses down to subclasses (for example, it generates field initialisers for a superclass which then will be used when generating a class data constructor function for a subclass). Therefore, the code generator performs sorting of classes by their depth in the class hierarchy.

Class member access is performed using pattern matching in MIL. The first `case` expression "separates" data fields from methods. The second one introduces data fields into the scope (as variable binders in a tuple pattern) and the third one does the same for methods:

```
def fun : {obj : Parent} -> Pure Int
  obj.method 1;
end

fun : {{}, {Unit -> Int -> Error Unit Int}} -> Error Unit Int =
  \(obj : {{}, {Unit -> Int -> Error Unit Int}}) ->
    let (var_3 : Int -> Error Unit Int) <-
      let (var_0 : {{}, {Unit -> Int -> Error Unit Int}}) <-
```

```
      return [Error Unit] obj
    in case var_0 of
         | {var_1 : {}, var_2 : {Unit -> Int -> Error Unit Int}} =>
             case var_1 of
               | {} =>
                   case var_2 of
                     | {method : Unit -> Int -> Error Unit Int} =>
                         return [Error Unit] method unit
                   end
             end
       end
  in let (var_4 : Int) <- return [Error Unit] 1
  in var_3 var_4;
```

Member access for objects wrapped in `Maybe` has one additional level of pattern matching on the value of the `Maybe` data type.

As we mentioned above, OOLang classes cannot have `Mutable` fields. They would be just pure immutable values in an object tuple, which class methods could not mutate. One could think about returning a new tuple for a mutated object, but this does not really work, since it would not be possible for other methods to see this change and get the new state of the object.

6.4 Conclusions

Modern object-oriented programming languages incorporate many of the ideas introduced and used for quite a long time in functional programming. OOLang tries to follow the same path, which makes MIL a good fit to be a target language for OOLang.

Despite the mismatch in the major programming paradigms supported by OOLang and MIL, the current state of MIL allowed to encode most of the OOLang features. OOLang classes and inheritance were a motivation to introduce tuples with subtyping to MIL.

Unfortunately, there is no good way in MIL to express OOLang `Mutable` class fields at the moment.

Many of the observations made in the corresponding section in the previous chapter apply here as well, for example, problems with capturing a desirable type for `catch_error`.

Chapter 7

Optimisations

The majority of modern compilers implement sophisticated optimisations to produce as good code as possible. In this chapter we will present several code transformations implemented in MIL. We conclude with a discussion around classical optimisation techniques and their relations to MIL.

As was mentioned in the "Introduction" chapter one of the goals of designing a monadic intermediate language is to be able to express and use optimising code transformations in the presence of effects. MIL provides a dozen of implemented transformations available for using from a source language compiler which targets MIL. Most of the transformations presented in this chapter are either direct implementations or inspired by the transformations presented in [Tolmach, 1998] and [Benton and Kennedy, 1999].

Unfortunately, since there is no intepreter or code generator implemented for MIL at the moment, we cannot execute MIL code and therefore cannot measure its runtime characteristics like running time or memory consumption. We can just estimate the impact of transformations by the code size. Depending on how MIL is implemented, an intuition about the usefulness of certain optimisations might be incorrect. Another way to get an idea about the runtime behaviour of some of the MIL constructs is to make connections to the semantics of similar operations as presented in [Tolmach, 1998] and [Benton and Kennedy, 1999].

We will provide some code snippets with real implementation of transformations, but to be able to do this in a reasonable amount of space we will focus at the expression level of granularity, since this is where the transformations happen. More high level cases (e.g. function definition) are merely recursing down to get to expressions. The MIL AST has support for Uniplate [Mitchell and Runciman, 2007] – a Haskell library for generic traversals of algebraic data types and so code transformations are implemented using these traversals. We want to highlight that many of the transformation implementations are relatively naive and serve the purpose of demonstrating the concept.

7.1 Monad laws

As was stated in Chapter 2 all monads must satisfy three monad laws: left identity, right identity and associativity. These three laws can be used not only to verify that a certain structure is indeed a monad, but also to optimise monadic code.

7.1.1 Left identity

The first law that we will consider is left identity. The following is its definition from Chapter 2:

$$\texttt{bind}\ (\texttt{return}\ \texttt{x})\ \texttt{f} = \texttt{f}\ \texttt{x}$$

The left identity monad law allows us to avoid a redundant `bind` and use `x` directly.

The following code snippet is a Haskell implementation of the left identity transformation for MIL:

```
leftIdentityExpr :: TyExpr -> TyExpr
leftIdentityExpr = descendBi f
  where
    f (LetE varBinder e1 e2) =
      case e1 of
        ReturnE _tm e ->
          (VarE varBinder, leftIdentityExpr e) `replaceExprIn`
            (leftIdentityExpr e2)
        _ -> LetE varBinder (leftIdentityExpr e1) (leftIdentityExpr e2)
    f expr = descend f expr
```

This and all the other transformations are implemented using the same pattern to perform a top-down traversal of the AST, namely using Uniplate's `descendBi` on the top level with a function that has a case where the optimisation can be applied and a general case, which uses `descend` with the function itself to continue the recursion. Using `descendBi` allows to apply the optimisation on the top level instead of skipping the top level and recursing down directly. We direct the reader to the Uniplate documentation for some related details.

In the implementation above we look for a `bind` expression which has `return` as a binder expression and substitute all of the occurences of the bound variable in the `bind` body with the expression under `return` (applying the transformation recursively to the subexpressions) using a helper function `replaceExprIn`.

The following is an MIL example containing the original code and then the resulting code after applying the transformation:

```
let (x : Unit) <- return [Id] unit
in return [Id] x;
```

```
==>

return [Id] unit;
```

7.1.2 Right identity

The second law allows us to eliminate redundant `bind` and a subsequent `return`:

$$\texttt{bind m return} = \texttt{m}$$

An implementation of this transformation looks like follows:

```
rightIdentityExpr :: TyExpr -> TyExpr
rightIdentityExpr = descendBi f
  where
    f (LetE varBinder e1 e2) =
      case e2 of
        ReturnE _tm (VarE vb) | vb == varBinder -> rightIdentityExpr e1
        _ -> LetE varBinder (rightIdentityExpr e1) (rightIdentityExpr e2)
    f expr = descend f expr
```

Here we look for a `bind` containing `return` of the bound variable as a body. If such an expression is found, it is replaced with the binder expression (with the transformation applied on top).

An example for this transformations looks pretty much the same as for the left identity, but what is really happening is that it is only the whole binder expression which is left (instead of subtituting `x` with `1` in the body):

```
let (x : Int) <- return [Id] 1
in return [Id] x;

==>

return [Id] 1;
```

7.1.3 Associativity

The last of the three monad laws is associativity. This law was presented as the following:

$$\texttt{bind}\ (\texttt{bind m f})\ \texttt{g} = \texttt{bind m}\ (\lambda \texttt{x} \rightarrow \texttt{bind}\ (\texttt{f x})\ \texttt{g})$$

The associativity transformation is implemented as in the snippet below:

```
associativityExpr :: TyExpr -> TyExpr
associativityExpr = descendBi f
  where
    f (LetE varBinder e1 e2) =
      case e1 of
        LetE varBinder' e1' e2' ->
          LetE varBinder' (associativityExpr e1')
            (LetE varBinder (associativityExpr e2') (associativityExpr e2))
        _ -> LetE varBinder (associativityExpr e1) (associativityExpr e2)
    f expr = descend f expr
```

The following example shows a simple MIL code snippet and its transformed version:

```
let (y : Unit) <-
  let (x : Unit) <- return [Id] unit
  in return [Id] x
in return [Id] y;

==>

let (x : Unit) <- return [Id] unit
in let (y : Unit) <- return [Id] x
in return [Id] y;
```

We can categorise the associativity transformation as one that *enables other transformations* (using the taxonomy of machine-independent transformations from [Torczon and Cooper, 2011]). This transformation allows to restructure the code and make it more linear, thus enabling other transformations to potentially improve the code further. For example, identity laws usually can be applied more times after the associativity has been applied.

7.2 Lift transformations

The next couple of transformations are related to the monad transformer `lift` operation. Both of them are trying to eliminate redundant `lift` operations. Whether these transformations can be considered optimisations really depends on the code that is generated for `lift`. On one hand, `lift` clearly has implementations for different monad transformers in Haskell, but on the other, one can think that `lift` can be erased after the MIL stage (for example, when generating LLVM or machine code). We could accept that a program in LLVM or machine code simply allows all the effects all the time, and MIL's type system ensured that everything is well typed.

7.2.1 Identity

The first one, which we call identity, can remove a `lift` where the source and the target monads are the same:

$$\texttt{lift}\,[\texttt{M}_1 \Rightarrow \texttt{M}_2]\,\texttt{e} = \texttt{e}, \texttt{where}\ \texttt{M}_1 \equiv_\alpha \texttt{M}_2$$

The next example is probably the simplest case of applying this transformation:

```
lift [Id => Id] return [Id] unit;

==>

return [Id] unit;
```

7.2.2 Composition

The second `lift` transformation that is implemented for MIL is trying to replace a sequence of two `lift` operations with one. In such a sequence of `lift`s the target of the inner (second) one is compatible with the source of the outer (first) one, which is guaranteed by the type/lint checking:

$$\texttt{lift}\,[\texttt{M}_2 \Rightarrow \texttt{M}_3]\,(\texttt{lift}\,[\texttt{M}_1 \Rightarrow \texttt{M}_2]\,\texttt{e}) = \texttt{lift}\,[\texttt{M}_1 \Rightarrow \texttt{M}_3]\,\texttt{e}$$

The next example demonstrates a composition of two `lift` operations:

```
lift [IO ::: State => Error Unit ::: (IO ::: State)]
  lift [State => IO ::: State] return [State] unit;

==>

lift [State => Error Unit ::: (IO ::: State)] return [State] unit;
```

7.3 Effect-dependent transformations

While, for example, monad laws and lift transformations above are applicable to all monads and monad transformer stacks, some code optimisations and transformations are effect-dependent. One of the strengths of intermediate languages based on monads and MIL in particular is the ability to express such transformations and correctly apply them only when it is possible.

7.3.1 Id

We will start with a transformation that is, in general, only applicable to pure computations. Computations inside the `Id` monad are one example of such computations.

The first transformation allows to reorder two computations. Such transformation is not obviously beneficial, but it can be used to enable other transformations or when it comes to low level considerations, may improve memory locality or parallelisation:

$$\texttt{let}\ (\texttt{x} : \texttt{T}_1) \leftarrow \texttt{e}_1\ \texttt{in}\ \texttt{let}\ (\texttt{y} : \texttt{T}_2) \leftarrow \texttt{e}_2\ \texttt{in}\ \texttt{e}$$

$$=$$

$$\texttt{let}\ (\texttt{y} : \texttt{T}_2) \leftarrow \texttt{e}_2\ \texttt{in}\ \texttt{let}\ (\texttt{x} : \texttt{T}_1) \leftarrow \texttt{e}_1\ \texttt{in}\ \texttt{e},$$

$$\texttt{where x is not used in e}_2\ \texttt{and Monad is Id}$$

Here is an implementation of this transformation:

```
exchangeExpr :: TyExpr -> TyExpr
exchangeExpr = descendBi f
  where
    f expr@(LetE varBinder e1 e2) =
      case e2 of
        LetE varBinder' e1' e2' | getBinderVar varBinder `isNotUsedIn` e1' ->
          case getTypeOf expr of
            TyApp (TyMonad (MTyMonad (SinMonad Id))) _ ->
              LetE varBinder' (exchangeExpr e1')
                (LetE varBinder (exchangeExpr e1) (exchangeExpr e2'))
            _ -> LetE varBinder (exchangeExpr e1) (exchangeExpr e2)
        _ -> LetE varBinder (exchangeExpr e1) (exchangeExpr e2)
    f expr = descend f expr
```

Another transformation, which can be used inside the `Id` monad, but not, for example, inside the `IO` monad in the general case is the elimination of redundant code:

$$\texttt{let}\ (\texttt{x} : \texttt{T}) \leftarrow \texttt{e}\ \texttt{in}\ \texttt{e}' = \texttt{e}',\ \texttt{where x is not used in e}'\ \texttt{and Monad is Id}$$

We skiped giving examples in these cases, since they would not be particularly interesting.

7.3.2 State

Next, we will describe four transformations applicable to computations inside the `State` monad. One thing to note when looking at these transformations is that there are no explicit checks for a monad being `State`. This is redundant, since the type system of MIL ensures that the state operations are used only inside the monad with `State`.

The first one is a special case of the reordering transformation presented in the previous section. We refer to it as "exchange new". It allows to reorder creation of two references:

$$\texttt{let}\ (\texttt{x} : \texttt{Ref}\ \texttt{T}_1) \leftarrow \texttt{new\_ref}\ [\texttt{T}_1]\ \texttt{e}_1\ \texttt{in}\ \texttt{let}\ (\texttt{y} : \texttt{Ref}\ \texttt{T}_2) \leftarrow \texttt{new\_ref}\ [\texttt{T}_2]\ \texttt{e}_2\ \texttt{in}\ \texttt{e}$$

$$=$$

$$\texttt{let}\ (\texttt{y} : \texttt{Ref}\ \texttt{T}_2) \leftarrow \texttt{new\_ref}\ [\texttt{T}_2]\ \texttt{e}_2\ \texttt{in}\ \texttt{let}\ (\texttt{x} : \texttt{Ref}\ \texttt{T}_1) \leftarrow \texttt{new\_ref}\ [\texttt{T}_1]\ \texttt{e}_1\ \texttt{in}\ \texttt{e},$$
$$\texttt{where}\ \texttt{x}\ \texttt{is not used in}\ \texttt{e}_2$$

In this transformation we are looking for a sequence of `bind` operations with `new_ref` as their binder expressions. It does a check for usage of the variable that holds the reference created with the first `new_ref`. This is done to avoid applying this transformation in the case, when the first reference is used to create the second reference.

Another special case of the reordering transformation is "exchange read", which can reorder reading of two references:

$$\texttt{let}\ (\texttt{x} : \texttt{T}_1) \leftarrow \texttt{read\_ref}\ [\texttt{T}_1]\ \texttt{e}_1\ \texttt{in}\ \texttt{let}\ (\texttt{y} : \texttt{T}_2) \leftarrow \texttt{read\_ref}\ [\texttt{T}_2]\ \texttt{e}_2\ \texttt{in}\ \texttt{e}$$
$$=$$
$$\texttt{let}\ (\texttt{y} : \texttt{T}_2) \leftarrow \texttt{read\_ref}\ [\texttt{T}_2]\ \texttt{e}_2\ \texttt{in}\ \texttt{let}\ (\texttt{x} : \texttt{T}_1) \leftarrow \texttt{read\_ref}\ [\texttt{T}_1]\ \texttt{e}_1\ \texttt{in}\ \texttt{e},$$
$$\texttt{where}\ \texttt{x}\ \texttt{is not used in}\ \texttt{e}_2$$

It is very similar to the "exchange new" transformation. It also does a check for usage of the variable that holds the value read from the first reference. This is done to avoid applying this transformation in the case, when the first reference contains another reference that is read in the second `read_ref`.

The third transformation for `State` computations allows to eliminate a reference reading in the case when this reference has already been read and bound to a variable:

$$\texttt{let}\ (\texttt{x} : \texttt{T}) \leftarrow \texttt{read\_ref}\ [\texttt{T}]\ \texttt{r}\ \texttt{in}\ \texttt{let}\ (\texttt{y} : \texttt{T}) \leftarrow \texttt{read\_ref}\ [\texttt{T}]\ \texttt{r}\ \texttt{in}\ \texttt{e}$$
$$=$$
$$\texttt{let}\ (\texttt{x} : \texttt{T}) \leftarrow \texttt{read\_ref}\ [\texttt{T}]\ \texttt{r}\ \texttt{in}\ \texttt{let}\ (\texttt{y} : \texttt{T}) \leftarrow \texttt{return}\ [\texttt{State}]\ \texttt{x}\ \texttt{in}\ \texttt{e}$$

Again, we look for a sequence of two `read_ref`s as in the previous transformation, but here we also check that the same reference is read (it works only when the reference is bound to a variable). In this case, `read_ref` is replaced with `return` of the variable that is bound to the result of the first `read_ref`.

Below is a small example that instead of reading the reference `x` again, reuses the value of the variable `a`, which already contains the value of `x`:

```
let (x : Ref Int) <- new_ref [Int] 1
in let (a : Int) <- read_ref [Int] x
in let (b : Int) <- read_ref [Int] x
in return [State] unit;

==>
```

```
let (x : Ref Int) <- new_ref [Int] 1
in let (a : Int) <- read_ref [Int] x
in let (b : Int) <- return [State] a
in return [State] unit;
```

The next `State` transformation also eliminates a reference reading, but in this case the information from a reference writing operation is used:

$$\texttt{let}\ (\texttt{u} : \texttt{Unit}) \leftarrow \texttt{write\_ref}\ [\texttt{T}]\ \texttt{r}\ \texttt{c}\ \texttt{in}\ \texttt{let}\ (\texttt{x} : \texttt{T}) \leftarrow \texttt{read\_ref}\ [\texttt{T}]\ \texttt{r}\ \texttt{in}\ \texttt{e}$$

$$=$$

$$\texttt{let}\ (\texttt{u} : \texttt{Unit}) \leftarrow \texttt{write\_ref}\ [\texttt{T}]\ \texttt{r}\ \texttt{c}\ \texttt{in}\ \texttt{let}\ (\texttt{x} : \texttt{T}) \leftarrow \texttt{return}\ [\texttt{State}]\ \texttt{c}\ \texttt{in}\ \texttt{e}$$

This transformation is very similar to the previous one, except for that the first operation must be `write_ref` in this case.

The following is an example of reusing an expression that was written to a reference instead of reading the reference:

```
let (x : Ref Int) <- new_ref [Int] 1
in let (z : Unit) <- write_ref [Int] x 2
in let (a : Int) <- read_ref [Int] x
in return [State] unit;

==>

let (x : Ref Int) <- new_ref [Int] 1
in let (z : Unit) <- write_ref [Int] x 2
in let (a : Int) <- return [State] 2
in return [State] unit;
```

The last `State` transformation eliminates an unnecessary reference writing, which directly follows a creation of the reference:

$$\texttt{let}\ (\texttt{x} : \texttt{Ref T}) \leftarrow \texttt{new\_ref}\ [\texttt{T}]\ \texttt{c}_1\ \texttt{in}\ \texttt{let}\ (\texttt{u} : \texttt{Unit}) \leftarrow \texttt{write\_ref}\ [\texttt{T}]\ \texttt{x}\ \texttt{c}_2\ \texttt{in}\ \texttt{e}$$

$$=$$

$$\texttt{let}\ (\texttt{x} : \texttt{Ref T}) \leftarrow \texttt{new\_ref}\ [\texttt{T}]\ \texttt{c}_2\ \texttt{in}\ \texttt{e}$$

This transformation is again quite similar to the ones above. An example follows:

```
let (x : Ref Int) <- new_ref [Int] 1
in let (z : Unit) <- write_ref [Int] x 2
in return [State] unit;
```

```
==>

let (x : Ref Int) <- new_ref [Int] 2
in return [State] unit;
```

7.3.3 Error

The last couple of the effect-specific transformations implemented for MIL is a pair of (rather naive) transformations which try to eliminate unnecessary `catch_error` calls.

The first one looks for a specific case, when the first argument of `catch_error` is `throw_error`, which basically means that the handler part will definitely be executed:

$$\texttt{catchName}\ [\texttt{Unit}]\ [\texttt{T}]\ (\texttt{throw\_error}\ [\texttt{Unit}]\ [\texttt{T}]\ \texttt{unit})\ (\lambda(\texttt{err} : \texttt{Unit}) \to \texttt{e})$$

$$=$$

$$\texttt{e},$$

$$\texttt{where catchName} == \texttt{catch\_error\_1 or catchName} == \texttt{catch\_error\_2}$$

It tries to find an application of `catch_error_1` or `catch_error_2` with `throw_error` as the first non-type argument. This transformation is simplified by the fact that it looks only for the case when the type of error values is `Unit`.

This optimisation is shown in the following code snippet:

```
catch_error_1 [Unit] [Int]
  (throw_error [Unit] [Int] unit)
  (\(e : Unit) -> return [Error Unit] 1);

==>

return [Error Unit] 1;
```

The second one looks for another specific case, when the first argument of `catch_error` is `return`, which means that no exception can be raised, making the `catch_error` call unnecessary:

$$\texttt{catchName}\ [\texttt{T}_1]\ [\texttt{T}_2]\ (\texttt{return}\ [\texttt{Error}\ \texttt{T}_1]\ \texttt{e})\ \texttt{h}$$

$$=$$

$$\texttt{return}\ [\texttt{Error}\ \texttt{T}_1]\ \texttt{e},$$

$$\texttt{where catchName} == \texttt{catch\_error\_1 or catchName} == \texttt{catch\_error\_2}$$

An example of this transformation is shown below:

```
catch_error_1 [Unit] [Int]
  (return [Error Unit] 1)
  (\(e : Unit) -> return [Error Unit] 2);

==>

return [Error Unit] 1;
```

7.4 Case expression transformations

In this section we present two transformations around `case` expressions which is an important part of implementing conditional and pattern matching constructs in source languages.

7.4.1 Constant case elimination

Constant case elimination transformation allows to remove a `case` expression, which has a known outcome (because of literal patterns and a literal scrutinee):

$$\texttt{case } s_i \texttt{ of } | \; s_j \Rightarrow e_j \texttt{ end}$$

$$=$$

$$e_i,$$

$$\texttt{where s is literal or data constructor}$$

It tries to find literal or data constructor scrutinee expressions, which are, basically, constant values and then to find a case alternative with a pattern corresponding to such a value.

An example of applying this transformation is presented below:

```
fun : State Int =
  case 1 of
    | 0 => return [State] 0
    | 1 => return [State] 1
    | _ => return [State] 2
  end;

==>

fun : State Int = return [State] 1;
```

This transformation can be applied to eliminate unnecessary `when` statements in OOLang, when it is known that a condition evaluates to `true` or `false`.

7.4.2 Common bind extraction

The common bind extraction is a transformation that can *hoist* a `bind` out of `case` alternatives. There are two variations of it: the first one deals with a common *body* expression, while the second one is about a common *binder* expression. The implementation of these transformations in MIL is very naive and uses simple equality both for variables and for body/binder expressions.

The first variant is presented below:

$$\texttt{case s of} \mid \texttt{p}_\texttt{i} \Rightarrow \texttt{let}\ (\texttt{v}_\texttt{i} : \texttt{T}) \leftarrow \texttt{e}_\texttt{i}\ \texttt{in e end}$$

$$=$$

$$\texttt{let}\ (\texttt{v} : \texttt{T}) \leftarrow \texttt{case s of} \mid \texttt{p}_\texttt{i} \Rightarrow \texttt{e}_\texttt{i}\ \texttt{end in e},$$

$$\texttt{where all occurrences of}\ \texttt{v}_\texttt{i}\ \texttt{in e are substituted with v}$$

The following is an example of applying it:

```
case 1 of
  | 0 => let (x : Int) <- return [IO] 0 in return [IO] unit
  | 1 => let (y : Int) <- return [IO] 1 in return [IO] unit
end;

==>

let (x : Int) <-
  case 1 of
    | 0 => return [IO] 0
    | 1 => return [IO] 1
  end
in return [IO] unit;
```

The second variation looks as the following:

$$\texttt{case s of} \mid \texttt{p}_\texttt{i} \Rightarrow \texttt{let}\ (\texttt{v}_\texttt{i} : \texttt{T}) \leftarrow \texttt{e in e}_\texttt{i}\ \texttt{end}$$

$$=$$

$$\texttt{let}\ (\texttt{v} : \texttt{T}) \leftarrow \texttt{e in case s of} \mid \texttt{p}_\texttt{i} \Rightarrow \texttt{e}_\texttt{i}\ \texttt{end},$$

$$\texttt{where all occurrences of}\ \texttt{v}_\texttt{i}\ \texttt{in}\ \texttt{e}_\texttt{i}\ \texttt{are substituted with v}$$

Below is a corresponding example:

```
case 1 of
  | 0 => let (x : Unit) <- return [IO] unit in return [IO] 0
  | 1 => let (y : Unit) <- return [IO] unit in return [IO] 1
end;

==>

let (x : Unit) <- return [IO] unit
in case 1 of
    | 0 => return [IO] 0
    | 1 => return [IO] 1
  end;
```

7.5 Constant folding

Constant folding is one of the very essential and popular optimisations in compilers which deals with finding constant expressions and evaluating them at compile time instead of doing unnecessary computations at runtime.

In MIL constant folding is implemented for all built-in arithmetic operations, except for division. It is not implemented for division, since division is a potentially failing operation (for example, when dividing by 0), so it would require a more detailed analysis. This transformation is using the following identities between the MIL built-in functions and mathematical operations ($\mathtt{l}_i$ are literals):

$$\mathtt{add\_int}\ \mathtt{l}_1\ \mathtt{l}_2 = \mathtt{l}_1 + \mathtt{l}_2$$

$$\mathtt{add\_float}\ \mathtt{l}_1\ \mathtt{l}_2 = \mathtt{l}_1 + \mathtt{l}_2$$

$$\mathtt{sub\_int}\ \mathtt{l}_1\ \mathtt{l}_2 = \mathtt{l}_1 - \mathtt{l}_2$$

$$\mathtt{sub\_float}\ \mathtt{l}_1\ \mathtt{l}_2 = \mathtt{l}_1 - \mathtt{l}_2$$

$$\mathtt{mul\_int}\ \mathtt{l}_1\ \mathtt{l}_2 = \mathtt{l}_1 * \mathtt{l}_2$$

$$\mathtt{mul\_float}\ \mathtt{l}_1\ \mathtt{l}_2 = \mathtt{l}_1 * \mathtt{l}_2$$

This time we will not have an example with only one transformation applied to a small piece of MIL code, but rather an extended example with a piece of OOLang code and a number of different transformations applied (the whole pipeline is presented in the next section). The example contains simple OOLang function performing arithmetic operations and printing their results. What follows is a fragment of non-optimised MIL code (we needed to cut it to take a reasonable amount of space) and finally the optimised version of it:

```
def main : Unit
  a : Int = 2 + 3;
```

```
  b : Int = 3 - 2;
  printInt b;  # Make sure it is not optimized away completely
  c : Int = 2 * 2;
  d : Int = a + c;
  printInt d;  # Make sure it is not optimized away completely
  e : Float = 2.0 + 3.0;
  f : Float = 3.0 - 2.0;
  printFloat f;  # Make sure it is not optimized away completely
  g : Float = 2.0 * 2.0;
  h : Float = e + g;
  printFloat h;  # Make sure it is not optimized away completely
  i : Int = 6 / 2;
  j : Float = 6.0 / 2.0;
end

==>

main : (Error Unit ::: (State ::: IO)) Unit =
  let (a : Int) <-
    let (var_30 : Int) <- return [Error Unit ::: (State ::: IO)] 2
    in let (var_31 : Int) <- return [Error Unit ::: (State ::: IO)] 3
    in return [Error Unit ::: (State ::: IO)] add_int var_30 var_31
  in let (b : Int) <-
       let (var_28 : Int) <- return [Error Unit ::: (State ::: IO)] 3
       in let (var_29 : Int) <- return [Error Unit ::: (State ::: IO)] 2
       in return [Error Unit ::: (State ::: IO)] sub_int var_28 var_29
  in let (var_2 : Unit) <-
       let (var_0 : Int -> (Error Unit ::: (State ::: IO)) Unit) <-
          return [Error Unit ::: (State ::: IO)] printInt
       in let (var_1 : Int) <-
            return [Error Unit ::: (State ::: IO)] b
       in var_0 var_1
  ...

==>

main : (Error Unit ::: (State ::: IO)) Unit =
  let (var_2 : Unit) <- printInt 1
  in let (var_5 : Unit) <- printInt 9
  in let (var_8 : Unit) <- printFloat 1.0
  in let (var_11 : Unit) <- printFloat 9.0
  in let (i : Int) <- div_int 6 2
  in let (j : Float) <- div_float 6.0 2.0
  in return [Error Unit ::: (State ::: IO)] unit;
```

7.6 Transformations and source languages

As we saw above, every transformation is implemented as a pure function which takes a typed MIL program and returns a typed MIL program. This allows to easily compose different transformations in a pipeline.

To chain different transformations in the code, we use a *postfix application operator* (also known as a very popular *pipeline operator* in the F# programming language). We define it in Haskell as follows:

```
(|>) :: a -> (a -> b) -> b
a |> f = f a
```

Here is how the chain of transformations for FunLang looks like:

```
optimiseMil :: MIL.TyProgram -> MIL.TyProgram
optimiseMil milProgram =
  milProgram
  |> MILTrans.associativity
  |> MILTrans.leftIdentity
  |> MILTrans.rightIdentity
  |> MILTrans.associativity
  |> MILTrans.associativity
  |> MILTrans.foldConstants
  |> MILTrans.leftIdentity
  |> MILTrans.foldConstants
```

The one for OOLang looks like the following:

```
optimiseMil :: MIL.TyProgram -> MIL.TyProgram
optimiseMil milProgram =
  milProgram
  |> MILTrans.associativity
  |> MILTrans.leftIdentity
  |> MILTrans.rightIdentity
  |> MILTrans.foldConstants
  |> MILTrans.leftIdentity
  |> MILTrans.foldConstants
  |> MILTrans.eliminateConstantCase
```

These particular pipelines are targeting some specific use cases used for testing. In order to achieve better results and apply as many optimisations as possible one could make use of *fixed point optimisations* (which are also available in Uniplate). Such a process would apply optimisation(s) until an optimisation pass does not change the program. There are different approaches of varying complexity to solving the problem of finding the best sequence of transformations, which we did not focus on in this thesis.

Of course, a compiler writer can implement additional MIL code transformations to optimise use cases specific to the source language being compiled.

Unfortunately, some of the transformations are not really applicable to the source languages presented in the previous chapters because of the monad stacks they have. For example, `State` transformations cannot be directly applied since they work only on computations that have just `State` as their effect, but monad stacks in FunLang and OOLang have, for example, `IO` in their impure stacks in addition to `State`.

7.7 Discussion

Correctness is often considered "the single most important criterion that a compiler must meet" [Torczon and Cooper, 2011]. This, of course, includes optimisations. Safety is extremely important when designing and applying code transformations. We can see that MIL type and effect system is of great help here. It plays a role in implementing transformations and being able to quite easily find out effect information or make assumptions based on types about a piece of code instead of performing additional analysis to derive these facts, which are not present in the IL (which is usually the case for most ILs). Moreover, MIL allows to run a lint checking potentially after every transformation to make sure that none of the transformations made a program ill-typed. While many classical ILs are imperative and statement based, meaning that most of the operations are assigning values to variables, MIL is a functional language and therefore it is *expression-based*, which in general makes it more easy to reason about the code, which implies that it should be easier to see whether a certain transformation is safe or not.

In this thesis we tried to work with quite simple-looking algebraic identities for expressing code transformations. Such identities are usually very minimal and very general at the same time, e.g. monad laws. This makes the implementation of optimisations based on them only a dozen lines of code in size. We also want to highlight that most industrial compilers are *not* implemented in a functional language, especially not in such an expressive and concise one as Haskell, while MIL, being a Haskell library, certainly benefits from this. Classical optimisations often use additional data structures (for example, sets) to maintain certain information about the program. It is certainly possible to implement more powerful and sophisticated transformations for MIL using a more algorithmic approach to the problem.

One example of this is recognising equivalent `case` expressions and hoisting them to perform the matching only once. An interesting observation to make here is that this kind of optimisation is performed on a functional language construct, but it can make consequent method calls on the same object in OOLang more efficient, given the way how those are compiled. Optimising method calls is one of the most important targets for OO-language compilers [Grune et al., 2012].

Some of the optimisations presented here try to meet the same goals as the *value numbering* method [Torczon and Cooper, 2011], [Muchnick, 1997], namely eliminating some redundancy in computations. MIL transformations like "use read", "use write" are more specialised and simple, while value numbering is a more general and more complex technique. MIL certainly could benefit from using value numbering to eliminate more redundancies (see examples at the end of this section).

One of the major well-known approaches to compiler optimisations nowadays is *data-flow analysis*. Examples of data-flow optimisations are *copy-propagation*, *constant-propagation* and *partial-redundancy* [Muchnick, 1997], [Torczon and Cooper, 2011]. Data-flow analyses mainly focus on how values flow through the program and not on what effects can happen (which is one of the strongest properties of MIL), although this still needs to be handled somehow. Data-flow techniques should be possible to apply to MIL as well. One possible way to go would be to incorporate Hoopl – a Haskell library for data-flow analysis and transformations [Ramsey et al., 2010]. However, an important consideration and a potential obstacle with this is that very many common optimisations are using the notions of *control-flow graph (CFG)* and *basic blocks* [Muchnick, 1997]. MIL, as was mentioned above, is a functional, expression-based language, but even more importantly, it is a higher order language (meaning, that it supports higher order functions). This introduces more complexity to the analysis (compared to simpler and more low-level ILs), which many common techniques are not designed to handle. It could be considered to transform MIL further to something more low-level and imperative to benefit from more classical optimisations.

We will conclude with two examples that show some of the MIL strengths and weaknesses and its relation to the classical methods mentioned above. The examples are mainly based on the value numbering example from Chapter 8 in [Torczon and Cooper, 2011].

First, we will demonstrate a redundancy which MIL cannot eliminate, but value numbering can. Below is a fragment of code that reads several integer values and then does some arithmetic manipulations and `Mutable` assignments in OOLang. One can notice that there is a common subexpression, namely `a - d`, which could be computed once.

```
b : Mutable Int <- readInt;
c : Mutable Int <- readInt;
d : Mutable Int <- readInt;
a : Mutable Int <- b + c;
b <- a - d;
printInt b;
c <- b + c;
d <- a - d;
printInt d;

==>

let (b : Int) <- readInt
in let (c : Int) <- readInt
in let (d : Int) <- readInt
in let (var_21 : Unit) <- printInt (sub_int (add_int b c) d)
in printInt (sub_int (add_int b c) d);
```

Looking at the generated (and optimised) MIL code, one can see that `b + c - d` is present twice (note that the expression for `a` was propagated to both its use sites). This is something that value numbering would be capable of eliminating, effectively rewriting `d <- a - d;` to `d <- b;`.

The second example is very similar to the first one, but there is an important difference: the integers are not read, but just have constant values. We introduced print statements to make sure that the code is not optimised away completely. Another important consideration is that we disabled all optimisations, except for monad laws for this example. It is especially important that constant folding is disabled, because otherwise almost everything would have been computed at compile time.

```
b : Mutable Int <- 1;
c : Mutable Int <- 2;
d : Mutable Int <- 3;
a : Mutable Int <- b + c;
printInt a;
b <- a - d;
printInt b;
c <- b + c;
printInt c;
d <- a - d;
printInt d;

==>

let (var_2 : Unit) <- printInt (add_int 1 2)
in let (var_5 : Unit) <- printInt (sub_int (add_int 1 2) 3)
in let (var_8 : Unit) <- printInt (add_int (sub_int (add_int 1 2) 3) 2)
in printInt (sub_int (add_int 1 2) 3);
```

Looking at the generated (and optimised) MIL code one can see that monad laws in this case did pretty much what constant and copy propagation techniques mentioned above are designed to do.

7.8 Conclusions

This chapter presented implementations of many transformations of MIL code ranging from monad laws and some other effect-independent transformations to transformations for specific effects like `Id`, `State` and `Error`.

Implementing transformations shown in this chapter does not require a lot of work. The structure of MIL AST and its support for Uniplate makes it possible to write clear and concise Uniplate-based transformation code.

Generality and composability of most transformations allow to easily reuse them and build modular code optimisation pipelines in source language compilers.

Applicability of some transformations to the code generated for source languages can be somewhat limited, but we believe that it can largely be solved by implementing some kind of effect inference/elimination process mentioned in Chapter 5.

Chapter 8

Conclusions

Finally, we will look at the results of this work and outline some ideas for future work.

8.1 Results

In this thesis we tried to focus on the design and implementation of a compiler IL that would allow to capture program effects such as input/output, exceptions, state manipulation in programs themselves and provide a way to express useful optimising code transformations (in the presence of effects) that can be composed and reused. In addition to this, the ambition was to explore an ability to provide a way of combining different effects to express different semantics of source languages.

As the result of this work, we introduced a monadic IL for modern programming languages – MIL, which fulfills the goals that were set for this work (the limitations will be mentioned in the next section). It uses monads to represent computational effects and monad transformers to combine these effects and allow to capture different semantics in the ordering of effects. It is possible to use MIL as a target for languages that belong to two major programming paradigms: object-oriented and functional. An example of this are two source programming languages (FunLang and OOLang) that were designed and implemented as part of this project.

MIL provides an expressive framework for implementing effect-aware code transformations. A number of code transformations has been implemented and are provided with MIL to demonstrate this and to be used in building optimisation pipelines in source language compilers, which target MIL. By providing support for the Uniplate library in the MIL AST, the implementations of different transformations are made clear and concise.

All of the source code for this project as well as this report are freely available at https://bitbucket.org/dmytrolypai/mil-masters-thesis-chalmers and https://github.com/dmytrolypai/mil-masters-thesis-chalmers. It includes rather comprehensive test suites for MIL, FunLang and OOLang as well as interactive environments for exploring and experimenting with the source languages.

8.2 Future work

Finally, we will outline some of the ideas for future work that came up during the work on this thesis.

Probably, one of the most interesting and important directions in continuing the work on MIL is the code generation from MIL. We think that the most appealing choice is to generate code for LLVM, which is a mature and powerful compiler framework. By generating code for LLVM we could reuse its low-level generic optimisations and native code generation. Other possibilities for lower level code generation include generating C code or native code.

Something that we did not address in this project and that is important to do is to evaluate MIL from the efficiency point of view, namely perform benchmarks on executable programs. This can be quite easily done given a code generator, as suggested above. Another way could be to write an interpreter for MIL in a way that it would be able to record some metrics during the execution, for example, a number of function applications, `bind` operations etc.

Another very important direction of MIL development is to make the MIL representation of effects and their combination more expressive. Right now MIL stacks are rather fixed and also order-dependent. This, of course, gives us benefits of expressing different semantics, but at the same time it cuts off the flexibility that might be available with a more set-based representation of effect combinations. As was mentioned in several previous chapters, MIL does not allow to express the desirable type of the `catch_error` function. Maybe, some sort of polymorphism in MIL stacks and an ability to have *monad variables* like, e.g. `Error Unit ::: (m ::: State)` would help in solving this problem.

To be able to use more transformations on a practical program MIL needs some kind of effect inference/elimination procedure to infer the real set of effects of a program, which might be smaller than what is declared. Implementing such a process for MIL would allow to optimise programs in source languages targeting MIL more heavily. A somewhat related area is trying to have more fine-grained effects, similar to [Benton and Kennedy, 1999], for example, splitting `IO` into `Input` and `Output` and `State` into `Allocation`, `Reading` and `Writing`.

MIL has a number of built-in monads and it does not provide a way to define new custom monads and therefore effects. Although the latter could be a subject for further exploration, we believe that it might be an unnecessary complication for MIL. The absence of user-defined effects is not a big limitation, given that MIL is an intermediate language and its goal is to have a reasonable set of built-in monads that allow to express most of the useful effects in potential source languages. But what we think might be beneficial is to explore which other monads can be added to MIL as built-in to be able to more easily cover a wider range of use cases. As was mentioned in Chapter 2 there exists a lot of interesting monads. What comes to mind as the most relevant to MIL are Backtracking [Kiselyov et al., 2005] and Par monad [Marlow et al., 2011]. The first one could be used to express, for example, logic programming languages and the second one could provide a way to support parallelism in the IL.

Another related track can be to try bringing laziness into MIL. One of the ways of doing it might be to incorporate ideas from [Peyton Jones et al., 1998], e.g. introducing language constructs for evaluation suspension and forcing. This paper was explained in more detail in the "Related work"

chapter.

Finally, one could look into a broader area of compiler modularity, in addition to what MIL already provides in terms of combining effects and pure composable transformations and their reuse. What we did not focus in this work was trying to have a modular representation of the AST (similar to the "a la carte" approach) and using monad transformers in the compilers themselves to compile different features/effects separately. Addressing this by following the work described in [Liang et al., 1995], [Swierstra, 2008], [Harrison and Kamin, 1998], [Day and Hutton, 2012] could be a way to cover this area of compiler modularity.

Bibliography

[Aho et al., 2006] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley.

[Alpern et al., 1988] Alpern, B., Wegman, M. N., and Zadeck, F. K. (1988). Detecting Equality of Variables in Programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 1–11, New York, NY, USA. ACM.

[Appel, 1998] Appel, A. W. (1998). SSA is Functional Programming. *SIGPLAN Not.*, 33(4):17–20.

[Appel, 2007] Appel, A. W. (2007). *Compiling with Continuations*. Cambridge University Press, New York, NY, USA.

[Bauer and Pretnar, 2012] Bauer, A. and Pretnar, M. (2012). Programming with Algebraic Effects and Handlers. Technical Report arXiv:1203.1539.

[Benton and Kennedy, 1999] Benton, N. and Kennedy, A. (1999). Monads, effects and transformations. *ELECTRONIC NOTES IN THEORETICAL COMPUTER SCIENCE*, 26:1–18.

[Birka and Ernst, 2004] Birka, A. and Ernst, M. D. (2004). A Practical Type System and Language for Reference Immutability. *SIGPLAN Not.*, 39(10):35–49.

[Boquist and Johnsson, 1997] Boquist, U. and Johnsson, T. (1997). The GRIN Project: A Highly Optimising Back End for Lazy Functional Languages. In *Selected Papers from the 8th International Workshop on Implementation of Functional Languages*, IFL '96, pages 58–84, London, UK, UK. Springer-Verlag.

[Brady, 2013a] Brady, E. (2013a). Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593.

[Brady, 2013b] Brady, E. (2013b). Programming and Reasoning with Algebraic Effects and Dependent Types. *SIGPLAN Not.*, 48(9):133–144.

[Cytron et al., 1991] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1991). Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490.

[Davidson and Fraser, 1980] Davidson, J. W. and Fraser, C. W. (1980). The Design and Application of a Retargetable Peephole Optimizer. *ACM Trans. Program. Lang. Syst.*, 2(2):191–202.

[Day and Hutton, 2012] Day, L. E. and Hutton, G. (2012). Towards Modular Compilers for Effects. In *Proceedings of the 12th International Conference on Trends in Functional Programming*, TFP'11, pages 49–64, Berlin, Heidelberg. Springer-Verlag.

[Eisenberg, 2015] Eisenberg, R. (2015). System FC, as implemented in GHC. https://github.com/ghc/ghc/blob/master/docs/core-spec/core-spec.pdf.

[Flanagan et al., 1993] Flanagan, C., Sabry, A., Duba, B. F., and Felleisen, M. (1993). The Essence of Compiling with Continuations. *SIGPLAN Not.*, 28(6):237–247.

[Girard, 1986] Girard, J.-Y. (1986). The System F of Variable Types, Fifteen Years Later. *Theor. Comput. Sci.*, 45(2):159–192.

[Grune et al., 2012] Grune, D., van Reeuwijk, K., Bal, H., Jacobs, C., and Langendoen, K. (2012). *Modern Compiler Design (2nd edition)*. Springer.

[Harris et al., 2005] Harris, T., Marlow, S., Peyton-Jones, S., and Herlihy, M. (2005). Composable Memory Transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 48–60, New York, NY, USA. ACM.

[Harrison and Kamin, 1998] Harrison, W. and Kamin, S. N. (1998). Modular Compilers Based on Monad Transformers. *IN PROCEEDINGS OF THE IEEE INTERNATIONAL CONFERENCE ON COMPUTER LANGUAGES*, pages 122–131.

[Harrison, 2001] Harrison, W. L. (2001). *Modular Compilers and Their Correctness Proofs*. PhD thesis, Champaign, IL, USA. AAI3017092.

[Hicks et al., 2014] Hicks, M., Bierman, G., Guts, N., Leijen, D., and Swamy, N. (2014). Polymonadic Programming.

[Hutton and Meijer, 1998] Hutton, G. and Meijer, E. (1998). Monadic Parsing in Haskell. *J. Funct. Program.*, 8(4):437–444.

[Jaskelioff, 2009] Jaskelioff, M. (2009). *Modular Monad Transformers*, volume 5502, pages 64–79. Springer Berlin Heidelberg.

[Jones, 1995] Jones, M. P. (1995). Functional Programming with Overloading and Higher-Order Polymorphism. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 97–136, London, UK, UK. Springer-Verlag.

[Kelsey, 1995] Kelsey, R. A. (1995). A Correspondence Between Continuation Passing Style and Static Single Assignment Form. *SIGPLAN Not.*, 30(3):13–22.

[Kiselyov and Ishii, 2015] Kiselyov, O. and Ishii, H. (2015). Freer Monads, More Extensible Effects. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell*, Haskell 2015, pages 94–105, New York, NY, USA. ACM.

[Kiselyov et al., 2013] Kiselyov, O., Sabry, A., and Swords, C. (2013). Extensible Effects: An Alternative to Monad Transformers. *SIGPLAN Not.*, 48(12):59–70.

[Kiselyov et al., 2005] Kiselyov, O., Shan, C.-c., Friedman, D. P., and Sabry, A. (2005). Backtracking, Interleaving, and Terminating Monad Transformers: (Functional Pearl). *SIGPLAN Not.*, 40(9):192–203.

[Lattner and Adve, 2004] Lattner, C. and Adve, V. (2004). LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA. IEEE Computer Society.

[Leijen, 2013] Leijen, D. (2013). Koka: Programming with Row-Polymorphic Effect Types. Technical Report MSR-TR-2013-79.

[Liang and Hudak, 1996] Liang, S. and Hudak, P. (1996). Modular Denotational Semantics for Compiler Construction. In *Proceedings of the 6th European Symposium on Programming Languages and Systems*, ESOP '96, pages 219–234, London, UK, UK. Springer-Verlag.

[Liang et al., 1995] Liang, S., Hudak, P., and Jones, M. (1995). Monad Transformers and Modular Interpreters. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 333–343, New York, NY, USA. ACM.

[Marlow, 2010] Marlow, S., editor (2010). *Haskell 2010 Language Report*.

[Marlow et al., 2011] Marlow, S., Newton, R., and Peyton Jones, S. (2011). A Monad for Deterministic Parallelism. *SIGPLAN Not.*, 46(12):71–82.

[Milner et al., 1997] Milner, R., Tofte, M., and Macqueen, D. (1997). *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA.

[Mitchell and Runciman, 2007] Mitchell, N. and Runciman, C. (2007). Uniform Boilerplate and List Processing. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, Haskell '07, pages 49–60, New York, NY, USA. ACM.

[Moggi, 1990] Moggi, E. (1990). *An abstract view of programming languages*. University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science.

[Moggi, 1991] Moggi, E. (1991). Notions of Computation and Monads. *Inf. Comput.*, 93(1):55–92.

[Muchnick, 1997] Muchnick, S. S. (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Peyton Jones, 1992] Peyton Jones, S. (1992). Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine - Version 2.5. *JOURNAL OF FUNCTIONAL PROGRAMMING*, 2:127–202.

[Peyton Jones et al., 1998] Peyton Jones, S., Shields, M., Launchbury, J., and Tolmach, A. (1998). Bridging the Gulf: A Common Intermediate Language for ML and Haskell. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 49–61, New York, NY, USA. ACM.

[Peyton Jones et al., 1999] Peyton Jones, S. L., Ramsey, N., and Reig, F. (1999). C–: A Portable Assembly Language That Supports Garbage Collection. In *Proceedings of the International Conference PPDP'99 on Principles and Practice of Declarative Programming*, PPDP '99, pages 1–28, London, UK, UK. Springer-Verlag.

[Peyton Jones and Wadler, 1993] Peyton Jones, S. L. and Wadler, P. (1993). Imperative Functional Programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 71–84, New York, NY, USA. ACM.

[Pierce, 2002] Pierce, B. C. (2002). *Types and programming languages*. MIT Press.

[Plotkin and Power, 2003] Plotkin, G. and Power, J. (2003). Algebraic Operations and Generic Effects. *Applied Categorical Structures*, 11(1):69–94.

[Ramsey et al., 2010] Ramsey, N., Dias, J. a., and Peyton Jones, S. (2010). Hoopl: A Modular, Reusable Library for Dataflow Analysis and Transformation. *SIGPLAN Not.*, 45(11):121–134.

[Sabry and Felleisen, 1992] Sabry, A. and Felleisen, M. (1992). Reasoning About Programs in Continuation-passing Style. *SIGPLAN Lisp Pointers*, V(1):288–298.

[Swierstra, 2008] Swierstra, W. (2008). Data Types à La Carte. *J. Funct. Program.*, 18(4):423–436.

[Tolmach, 1998] Tolmach, A. P. (1998). Optimizing ML Using a Hierarchy of Monadic Types. In *Proceedings of the Second International Workshop on Types in Compilation*, TIC '98, pages 97–115, London, UK, UK. Springer-Verlag.

[Torczon and Cooper, 2011] Torczon, L. and Cooper, K. (2011). *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition.

[Wadler, 1990] Wadler, P. (1990). Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 61–78, New York, NY, USA. ACM.

[Wadler, 1995] Wadler, P. (1995). Monads for Functional Programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, UK. Springer-Verlag.

[Wu and Schrijvers, 2015] Wu, N. and Schrijvers, T. (2015). *Fusion for Free*, volume 9129, pages 302–322. Springer International Publishing.