Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Odd Kristian Kvarmestøl

# Database Access through a Functional Programming Language

Master's thesis in Informatics
Supervisor: Svein Erik Bratsberg
May 2020

NTNU
Kunnskap for en bedre verden

Odd Kristian Kvarmestøl

# Database Access through a Functional Programming Language

Master's thesis in Informatics
Supervisor: Svein Erik Bratsberg
May 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

**Abstract**

*Stage* is a new functional programming language. It was designed to be statically typed and without function side effects. These principles were chosen with the intent of making the language easy to use and easy to debug. In this project an SQL interface was developed for Stage. This interface provides developers with an ergonomic system for writing SQL queries, that take parameters from the program. The SQL interface aims to be type safe and to automate tedious tasks. Such tasks include: keeping track of query parameters, and converting the returned tuples into native data types.

This thesis has two main goals: (1) to explore the implementation of an SQL interface into Stage, and (2) to develop Stage into a working language able to express the SQL interface. At the outset of the project, the idea for the language and a foundation for the compiler had been implemented. During this project the compiler and language was further developed into a functioning prototype. The design of the SQL interface was conceived and developed along with the compiler, and a simplified version was implemented. This version serves as a proof-of-concept for the fully designed interface.

This thesis describes the design of Stage and the SQL-interface. The language's compiler, and the systems it is comprised of, will also be discussed.

**Sammendrag**

*Stage* er et nytt funksjonelt programmeringsspråk. Språket har statisk typing og tillater ikke funksjoner med sideeffekter. Dette er for å gjøre språket lett å bruke samt lett å feilsøke. Et SQL-grensesnitt har blitt utviklet for Stage som en del av dette prosjektet. Dette grensesnittet tilbyr utviklere et ergonomisk system for å skrive SQL-spørringer som bruker parametere fra applikasjonen sin. SQL-grensesnittet er typesikkert og har som mål å automatisere bort omstendelige oppgaver. Dette er oppgaver som å holde styr på parameterne til en spørring eller konvertere radene gitt av databasen til datatyper i programmet.

Oppgaven har to hovedmål: (1) å utforske design av et SQL-grensesnitt for Stage, og (2) å utvikle Stage til å bli et fungerende språk som kan uttrykke SQL-grensesnittet. I begynnelsen av prosjektet var ideen for språket samt et grunnlag for kompilatoren implementert. I løpet av prosjektet har språket og kompilatoren blitt utviklet til en fungerende prototype. Designet av SQL-grensesnittet ble utviklet sammen med kompilatoren, og en forenklet versjon av SQL-grensesnittet ble implementert. Denne versjonen fungerer som en proof-of-concept for det fullstendige grensesnittet.

Denne masteroppgaven beskriver designet til Stage og SQL-grensesnittet. Kompilatorens design, samt systemene den er satt sammen av, blir også diskutert.

# Contents

# Chapter 1

# Introduction

Stage is a functional programming language that is currently being designed and developed by the author of this thesis. The goal of the language is to make specifying real time data flow systems easy. The language is being designed with strong type safety in mind to make development as easy as possible while ensuring the system will run as expected when in use.

However, so far no effort has been made to integrate database access into this language. Having such an ability is important to allow persistent and reliable storage of data for both short and long term.

The mutable nature of a DBMS is inherently conflicting with the pure and type safe nature of pure functional languages. Resolving this disparity is an important part to implementing such a library.

This master project aims to integrate DBMS access into the programming language. This will be done as a literature review of existing database interfaces, and will result in a prototype implementation of a library that is capable of interacting with a DBMS in a functional manner. In this report we present the design and a prototype implementation for both the new pure functional programming language and an SQL interface for this language.

The project source code is hosted on GitHub[1], and the version at the time of project delivery is available at the tag `v0.1.0`[2].

The goals for this master project is:

- Design database access as an integrated part of a new functional programming language.
- Design an API that is a natural part of the language.
- Find a database solution that may easily be supported as a part of the functional language.
- Evaluate if a server based database (e.g. PostgreSQL) or an embedded database (e.g. SQLite) is the best solution.
- Prototype a database solution for this programming language.

[1] `https://github.com/oddkk/stage`

[2] `https://github.com/oddkk/stage/tree/v0.1.0`

Chapter 2 will lay out the foundation for the language and the database access interface. Chapter 3 will introduce the syntax and semantics of the language, and chapter 4 will detail the design of the SQL interface. Chapter 5 will describe the implementation of the prototype compiler and chapter 6 will review the design and implementation of the language, compiler, and SQL interface.

# Chapter 2

# Background

This chapter introduces the fundamentals of *functional programming languages* and in particular their *type systems* and common patterns. We will also examine some existing database interfaces for other languages to look at common patterns and approaches. These systems provide the foundation upon which the language, SQL-interface, and compiler discussed in later chapters are built.

## 2.1 Functional Programming

Functional programming is a paradigm that is based on using functions and their compositions as the primary building block of larger programs. This kind of language extend the pure $\lambda$ calculus from category theory with additional constructs[1].

Functional programming languages come in two versions: pure and impure[1][2]. Impure functional languages, such as Standard ML, have facilities for state and for evaluating functions with side effects. Examples of side effects are I/O, such as writing to a file or reading from a database, and assigning to a mutable variable. In pure functional languages, such as Haskell, the program is unable to modify state or directly invoke side effects.

The pure languages can further be classified as *call-by-value* or *call-by-name*[1], often called lazy. Call-by-value indicates that values are eagerly being calculated when they are used as function arguments. For example in an expression such as $f(a+2)$, the argument $a+2$ will be evaluated before being passed to the function $f$. Call-by-name would instead substitute the argument into the function using capture-avoiding substitution, thereby postponing evaluation until the argument is actually needed. The same effect can be achieved in call-by-value languages by having the function take a function of no parameters instead of the value itself as parameter. In the previous example for example this would be $f(() => a+2)$.

### 2.1.1 Type System

Languages such as Python, JavaScript, and Lisp employ dynamic type checking. These languages store types together with values, and defer the checking of types to when a value is used at run-time[3][4]. This style of language allow developers to make functions which can be applied over values of a wide variety of types. A drawback of such languages is that it is difficult to find bugs caused by passing a value of an unexpected type to a function[5].

On the other hand, languages like C and Java requires all types to be known at compile time, and the developer is required to explicitly state the type of all variables in the program text. This allows the compiler to check if the program contains "silly mistakes"[3], such as passing a string to a parameter expecting an integer.

Hindley[6] and Milner[5] independently developed a static, polymorphic type system. This typing system aimed to introduce some of the simplicity of dynamic languages into statically typed languages. This is achieved in particular through the use of parametric polymorphism and type inference. This kind of type system has two primary tasks: type checking and type inference. The type inference system attempts to fill out the types not explicitly specified in the program text. Type checking aims to verify that the typing schema of the program is correct.

In the Hindley-Milner system there are two kinds of types: *mono types* and *poly types*. Mono types represent exactly one kind of values. `int` is an example of a mono type because it only represents only integer numbers. Poly types are composed of other type variables. These variables can themselves be either mono- or poly-types. A function such as $\forall\alpha.\alpha \rightarrow \alpha$ is an example of a poly type. The type variable $\alpha$ can be substituted by any mono- or poly-type. Poly types are useful for describing polymorphic constructs, such as parametric types, functions, and type classes. Poly types can be turned into mono types by substituting all type variables with mono types.

### 2.1.2 Monads

The monad design pattern was borrowed by Haskell from category theory[7]. Monads provide a way for programmers to specify a composable and serial list of operations that are evaluated inside the context of the monad. Monads allow for sequencing actions in a way that emulates side effects. This gives purely functional programming languages a way of specifying stateful programs.

Monads were first considered as a tool for analyzing lambda calculus with state in category theory by Moggi in 1989[8]. They were later introduced as a mechanism to provide stateful operations in pure functional languages in 1990[7]. Monads have since been introduced into the Haskell programming language, both in the language itself and as a core mechanism driving the Glasgow Haskell Compiler[9].

Each monad `M` consist of three functions[9]:

- A **type constructor**, `M`: $[T : \textbf{Type}] \rightarrow M[T]$, that creates a monadic type of type $T$.
- A **type converter**, `return`: $(\$T) \rightarrow M[T]$.
- A **combinator**, `bind`: $(M[\$T], (M[T]) \rightarrow M[\$U]) \rightarrow M[U]$

While the programming language itself is functional, it still needs to have side effects. Certain monads can be used to express "actions" that, when dispatched by the underlying runtime system, can perform operations with side effects. The result of the side effect can then be returned to the program[2][10]. In both Scala Cats Effect[11] and Haskell[12] this monad is named IO.

Using the combinator we can create a monad that will evaluate two monads in sequence. The combinator evaluates the first monad and passes the returned value to the function on the right. This function returns another monad that is then evaluated. For example, to read a line from `stdin` and echo that line back twice we can write the following in Haskell.

```
1 getLine >>= (\ln -> putStrLn(ln) >>= (\_ -> putStrLn(ln)))
```

This expression returns a new IO-monad. The line will not be read and printed back immediately when the monad is declared. The monad instead must be passed to some mechanism that will execute it. In Haskell this is done by returning the monad from main.

```
1 main :: IO ()
2 main = getLine >>= (\ln -> putStrLn(ln) >>= (\_ -> putStrLn(ln)))
```

Syntactic convenience have been introduced in Haskell to make monadic expressions easier to write. First, notice that the output from the first `putStrLn` is not used. We can introduce an operator >> which functions like bind but ignores the value returned from the left hand monad. In Haskell, this can be written as follows.

```
1 (>>) :: Monad m => m a -> m b -> m b
2 a (>>) b = a >>= (\_ -> b)
3
4 main = getLine >>= (\ln -> putStrLn(ln) >> putStrLn(ln))
```

To make the sequencing of operations easier, Haskell introduced do-notation[13]. This notation is reminiscent of imperative programs with statements that are evaluated in order.

```
1 main = do {
2     ln <- getLine;
3     putStrLn(ln);
4     putStrLn(ln);
5 }
```

The result of a monadic computation can be kept by prefixing the expression with a name and `<-`, as shown on line 2 of the previous listing. Do-notation is merely syntactic sugar that is translated into a series of binds. This do-expression example is translated into the same expression as above.

The type converter function, `return`, returns a monad that will return the given value. For example, the above expression could be modified to always print "test" twice as show in the following listing.

```
main = do {
    ln <- return("test");
    putStrLn(ln);
    putStrLn(ln);
}
```

While these examples all use monads to deal with IO, monads have other uses. One example is the State monad[7]. This monad executes a series of operations, and is able to `get` and `put` a value into its store. The following example in Haskell illustrates this monad.

```
runState do {
    v1 <- get;
    put v1 + 3;
    v2 <- get;
    put v1 + 10;
    return v2;
} 5
```

This expression returns `(8, 15)`. This tuple contains the value returned from the expression, v2, and the state at the end of the operations.

### 2.1.3 Type Classes

Languages such as C++ and C# provide mechanisms for overloading functions and operators. This way of providing different functionality based on the type signature of the function is called ad-hoc polymorphism[3]. This is opposed to parametric polymorphism where a function is defined over many types with the same behaviour for each. Type classes are a way of making "ad-hoc polymorphism less ad hoc"[14]. Type classes introduce the notion of a "class" of types that possess a common set of operations[13].

For Haskell, the type class for equality tests, `Eq`, can be declared over a type parameter `a` with the class statement listed below. This type class will contain the equality test function `(==)` which must be implemented for all instances of this type class. Afterwards, instances of the type class can be declared with the instance statement. Here, the character and integer types receive such implementations.

```
class Eq a where
    (==) :: a -> a -> bool

instance Eq Int where
    (==) = eqInt

instance Eq Char where
    (==) = eqChar
```

After this, the equality test function can be used, either for comparison of integers or characters, or any other type the type class is implemented for. Any usage of the function on types that are not an instance of the class results in a compilation error.

```
> 2 == 4
False
> 'a' == 'a'
True
> (\a -> a) == (\a -> a)
error:
    • No instance for (Eq (p0 -> p0)) arising from a use of '=='
        (maybe you haven't applied a function to enough arguments?)
    • In the expression: (\ a -> a) == (\ a -> a)
      In an equation for 'it': it = (\ a -> a) == (\ a -> a)
```

## 2.2 SQL Database Access

SQL database access libraries fundamentally take or construct SQL statements, passes those statements to the database server, and receives and presents the result to the application. Several paradigms have been developed on top of this core principle to make interfacing with the database easier. This section will discuss some of these approaches.

### Imperative Interfaces

The simplest library paradigm for interacting with databases is to provide a interface for querying using SQL as a string. Databases such as PostgreSQL and MySQL provide such libraries for C which serves as the foundation for other libraries that provide abstractions on top. Libraries such as ODBC and JDBC provide standard APIs for accessing a database of any of the supported kinds.

The following is an example of querying a PostgreSQL database using libpq[15]. The text provided as parameters is escaped before being incorporated into the query, allowing user data to be passed directly without risk of vulnerabilities. Failing to properly escaping user-provided data can enable SQL-injection attacks.

The parameters can be passed as types other than text. For such parameters the binary value is interpreted on the back end according to what type it is expected to be. The values returned from the query are also passed as text, as requested by the `resultFormat` parameter being 0. Both the returned values and the parameters can be passed as their binary representation. This does require the client to have knowledge about the back end's representation of these values. For parameters, the types can then either be inferred from the query, be explicitly stated in the query by suffixing parameter usage with '::' followed by the type name, or by specifying OIDs through the `paramTypes` parameter to PQexecParams. The returned fields' types can be controlled by suffixing the fields to return with '::' followed by the type name, or be checked by inspecting the field's type OID with the `PQftype` routine.

```c
#include <stdio.h>
#include <libpq-fe.h>

int main(int argc, char *argv[])
{
```

```
    PGconn *connection;
    connection = PQconnectdb("dbname=test");

    const char *const args[] = { "1" };

    PGresult *query;
    query = PQexecParams(
        connection,
        "select * from account where account=$1",
        1,    // nParams
        NULL, // paramTypes
        args, // paramValues
        NULL, // paramLengths
        NULL, // paramFormats
        0     // resultFormat, 0 = all data returned as text.
    );

    if (PQresultStatus(query) == PGRES_TUPLES_OK) {
        int num_rows, num_columns;
        num_rows    = PQntuples(query);
        num_columns = PQnfields(query);

        for (int row = 0; row < num_rows; row++) {
            for (int column = 0; column < num_columns; column++) {
                printf("%s ", PQgetvalue(query, row, column));
            }

            printf("\n");
        }
    } else {
        printf("Query failed:\n%s", PQresultErrorMessage(query));
        return -1;
    }

    return 0;
}
```

### Functional Interfaces

We will investigate a library named Doobie for Scala as an example of a functional query interface. Scala is a multi-paradigm programming language built on top of the Java Virtual Machine[16]. It is geared towards both object oriented and functional programming.

Doobie is a purely functional JDBC API[17]. It is built on Cats, which is a library that provides abstractions for functional programming in Scala, and aims to provide the foundation for other pure and type full libraries[18].

As an example, the following Scala code demonstrates the same query as the C example above.

```
import doobie._;
import doobie.implicits._;
import cats.implicits._;
import cats.effect.IO;
import scala.concurrent.ExecutionContext;

object Main extends App {
```

```
8   implicit val cs = IO.contextShift(ExecutionContext.global);
9
10  val xa = Transactor.fromDriverManager[IO](
11    "org.postgresql.Driver",
12    "jdbc:postgresql://localhost/test",
13    "test", "test"
14  );
15
16  case class Account(account: Long, name: String);
17
18  def getByID(id: Long) =
19    sql"select account, name from account where account=${id}".query[Account];
20
21  getByID(5)       // :: Query0[Account]
22    .to[List]      // :: ConnectionIO[List[Account]]
23    .transact(xa)  // :: IO[List[Account]
24    .unsafeRunSync // :: List[Account]
25    .foreach(println);
26 }
```

The getByID function constructs an object of type **`Query0[Account]`**. This object represents a query to the database which is closed over its input arguments. The library utilize the string interpolation inside of `getByID` to capture the query arguments. When evaluated, the query is expected to return objects of type **`Account`**. Next, using `to[List]` indicates we expect to receive a list of **`Account`**s from the database. This function returns an instance of the **`ConnectionIO`** monad. This monad will, when evaluated, query the database within the context of the database connection. This context is created using `transact(xa)` with a connection object `xa`. This monad executes the actions described by the **`ConnectionIO`** monad within the context of the **`IO`** monad. Finally, since Scala allows routines with side effects, the query is evaluated with `unsafeRunSync`.

### Language Integrated Queries

Language Integrated Query (LINQ) is an API developed by Microsoft, which is integrated into their language C# for performing operations over lists of objects[19] [20]. While not itself an ORM, together with LINQ to SQL, LINQ can act as one. LINQ also allows querying other collections, such as XML-files and web-based resources. The goal with LINQ is to abstract away the actual query interface and provide a uniform way of working with collections from different sources. It also provides type safety and a means for the programmer's IDE (Visual Studio and IntelliSense in the case of C#) to reason about the queries' type. This enables the IDE to provide the user with auto completion of code. In addition to the IEnumerable interface's methods, LINQ provides a language extension that allows programmers to write queries with a syntax reminiscent of SQL.

LINQ is based around the IEnumerable and IEnumerable<T> interfaces which provides basic methods for manipulating a list of elements. These provide a monadic style of list comprehensions inspired by Haskell[13].

Another option is to base the language integrated queries on the list comprehension system[21]. Such a system could utilize the monadic list comprehensions.

This also does not require a completely separate embedded domain specific language (DSL), like the SQL-like language extension for C# LINQ.

# Chapter 3

# Language Design

The goal for this programming language is to allow users to easily describe real-time data streaming and signalling applications, similar to streaming databases. It is designed to be easy to use, concise, and safe. This is achieved through static typing, pure functions, and immutability.

## 3.1 Language Overview

This section will introduce the core syntax and semantics of the language and will discuss the design decisions behind it.

```
1 use mod base.init.printLn;
2
3 !printLn("Hello, World!");
```

To run this example, type the above program text into a file, `hello_world.stg`, and pass it to the compiler:

```
1 $ stage hello_world.stg
2 Hello, World!
```

The line `use mod base.init.printLn;` imports the print line function from the base module which contain standard functionality. The text is printed with the line `!printLn("Hello, World!");`. Notice the exclamation mark before `printLn(...)`. This tells the compiler that the action on the right of the exclamation should be executed when the program is initialized. Omitting the exclamation would result in nothing being printed. This mechanism is discussed in detail in section 3.1.8.

### 3.1.1 Modules

Modules are a way of organizing pieces of code and data into composable parts. A module is a directory in the file system under the system module directory, that contains zero or more `.stg` files in a hierarchy. The root name space is populated by a file named `mod.stg`, if present. All other files create their own name space inside the parent name space. Directories containing `.stg` files also create name

spaces, and as with the top-level directory, these will be populated by the `mod.stg` file, if present.

Modules can also provide a shared object, named `module.so`, in the top directory of the module, which can perform complex interactions with the compiler (see section 5.5.1).

If a module contains init-expressions (see section 3.1.8), they will be executed as part of the module initialization (see section 5.5).

Foreign modules can be accessed with the `mod <modname>` expression, where `<modname>` is the name of an available module. For example, to access the member `a` on the module `test`, one could write `mod test.a`.

### 3.1.2 Composite Data Types

Composite data types, structs, provides a way of grouping members of several types together in a single object.

There are two uses for composite data types: as data structures, `Structs`, and as modules. Each file inside a module compiles into a composite data type that represents its name space. These name spaces are combined in the module's top level composite data type.

Data structures, `structs` can be declared by enclosing a set of statements in `struct { stmt1; ... stmtn; }`. The statements available in modules are also available within structures.

When a composite data type is declared inside another composite data type, either as a name space in a module or as a data structure, all the members of the parent's scope can be accessed as closures. This means that the values will be stored on the child data type at the time of the parent's instantiation. This also means that values on a data type is available for all its descendants.

#### Member Declaration and Assignment

A composite data type is composed of zero or more members. Each member has a type that must be known at the time of compilation, and may have an assigned value. If no default value was given during declaration, a value must be provided during object instantiation.

A member *a* of type *int* can be declared with the following statement. This member will not be bound, so it must be bound by another statement, either at the composite declaration site, or during object instantiation.

```
1 a: int;
```

A member *a* can be bound to the value *2* with the following statement. The value expression must be be of the same type as *a* was declared with. In this case, a was declared as an integer, and 2 is an integer literal.

```
1 a = 2;
```

Instead of having to use two statements to declare and bind the above member, the statements can be combined into the following declare-and-bind statement.

```
a: int = 2;
```

In the case of the above declaration, the type of *a* is obvious from the value expression *2*. In such cases the type expression can be omitted.

```
a := 2;
```

Declarations and assignments inside a composite data type can appear in any order and can depend on any other member, as long as there are no cyclic dependencies.

```
# OK
a := b;
b := 2;

# Compilation error: Cyclic dependency between 'c' and 'd'.
c := d;
d := c;
```

All members must be bound exactly once before the containing data type can be instantiated. Binding a member multiple times or not binding a member at all results in a compilation error.

Composite data types can provide a default bind that can be overridden by another bind. An overridable bind is designated by prefixing the bind operator (=) with a tilde (~). In the following example, *a*, *b*, and *c* will all be bound to *5*.

```
a: int ~= 2;
a = 5;

b: int;
b ~= 2;
b = 5;

c :~= 2;
c = 5;
```

To avoid confusion about what overridable bind would be applied, each member can have at most one overridable bind. For instance, the following snippet results in an compilation error due to conflicting overridable binds.

```
a: int;
a ~= 2;
a ~= 5;
```

Based on this description we can impose three rules that a composite object need to exhibit before it can be instantiated.

1. Each member must be bound by either an overridable or a non-overridable bind.
2. Each member can be bound by at most one non-overridable bind.
3. Each member can be bound by at most one overridable bind.

#### Use Statements

Use statements allows the user to import names from other scopes into the current one. This is useful to reduce clutter from having to type out a fully qualified path each time. It is also useful for importing operator names into the current context.

The expression of the use expression can be a direct reference to a member, in which case that member will be imported. The `use`'s expression can also be suffixed with `.*` to import all names of that scope into the current scope.

The following example will import the names from the composite object `a` of type `Test` into the current scope.

```
1 Test := struct {
2     a: int = 2;
3     b: int;
4 };
5
6 test: Test;
7 use test.*;
8
9 b = 5 + a;
```

A special case of the `use` statement is the `mod` statement. `mod`, if used as a statement instead of as an expression, is translated into a use of that mod. For example, `mod test;` is equivalent to `use mod test`.

In the current compiler implementation, while the `use`d names are available in the data type's scope and all descendent scopes, they are not part of the object definition. They can therefore not be accessed as members of an instance of the data type.

#### Instantiation

After having created a composite data type `S`, an object of that type can be instantiated using the expression `S { ... }`. The body of the instantiation can be filled with bind statements that fill out the unbound or overridably bound members from the data type. As noted above, all members must be bound before the object can be instantiated. Failing to bind all members results in a compilation error.

### 3.1.3 Variants

A variant is a data type that can have one of a predefined set of values. Such a type is defined by enclosing a comma separated list of the variant's options in the `variant { ... }` expression. A simple variant is the boolean type which can have either the value `true` or the value `false`. This type is defined as follows.

```
1 Bool := variant { true, false };
2 use Bool.*;
```

Variant options can have a data field. In this case the option value is only present if the corresponding option is selected. An example of a variant with data type is the `Maybe` type. This definition of the variant holds either `some` integer value or `none`.

```
1 Maybe := variant { some int, none };
2
3 a := Maybe.some(2);
4 b := Maybe.none;
```

In reality we wish to define `Maybe` over any type, so it is defined as a polymorphic type.

```
Maybe := [$T] variant { some T, none };

a: Maybe[int]    = Maybe.some(2);
b: Maybe[String] = Maybe.some("Hello, World!");
```

### 3.1.4 Functions

Functions can be declared as follows.

```
(a: int, b: int) -> int => a + b
```

This expression consists of two parts: the function prototype and the body. The prototype, `(a: int, b: int) -> int`, is everything up to the lambda operator, `=>`. A function prototype consists of zero or more parameters separated by commas (`,`). Each parameter has a name, followed by colon (`:`) and its type. After the list of parameters, the type of the returned value is prefixed with the function type operator, `->`.

Often the return type of the function is obvious from the body. In those cases the return type specifier can be omitted.

```
(a: int, b: int) => a + b
```

Because anonymous, in-line functions are common, the parameter type can also be omitted for functions with a single parameter. Currently the type of the parameter must be obvious from their use in the body. In a future implementation of the language the functions should strive to be as generic as possible[5].

```
a => a + 2
```

When used in the context of a scope, functions can capture a closure of variables in that scope. If the closure value is not constant at compile time it is filled in at the time of the function's instantiation.

```
b := 2;
f := (a: int) -> int => a + b;

A := struct {
    x: int;
    f := (a: int) -> a + x;
};

s := A { x = 5; };
!printLn("${s.fn(2)}"); # 7

t := A { x = 10; };
!printLn("${t.fn(2)}"); # 12
```

Recursion is possible for named functions.

```
fac := (a: int) -> int
=> match a {
    0 => 1;
    _ => a * fac(a-1);
```

```
};

!printInt(fac(9));
```

Parametric functions can be declared by placing template parameters in the function prototype. A template parameter is designated by the name of the parameter prefixed with `$`. For instance, the identity function, `id` can be declared like this.

```
id := (a: $T) -> T => a;

a := id(2);               # a = 2
b := id("Hello, World!"); # b = "Hello, World!"
c := id(true);            # c = true
```

A template parameter can only be declared with the '`$`' prefix once. This declaration should happen at the point of the function prototype that is considered authoritative for that parameter's value. For all other uses the name of the parameter is enough. It does not matter from the type solver's perspective what parameter is tagged as authoritative, but this information is used when reporting typing errors.

The template parameter can appear inside of type constructors and function type constructors. For instance, the compose function, $f \circ g$, can be declared as follows.

```
compose := (f: ($T) -> $U, g: (U) -> $V) -> (T) -> V
    => (a: T) -> V => g(f(a));
```

A function type can be declared using the function type operator, `->`. For instance, `(int, int) -> String` is the type of a function that takes two integers as parameters and returns a string. The function type declaration does not include parameter names. The function parameter names are not part of the function type and are only used for naming the parameters in the function body scope.

### 3.1.5 Tuples

Tuples contain a fixed set of members of disparate types. They are similar to structs except their members are anonymous. A tuple can be created by surrounding a comma separated list of elements with parenthesis. For example, a three-tuple of a integer, a string, and a boolean can be created as follows. `(2, "test", true)`. The type of this object is declared by `Tuple(int, String, Boolean)`. `Tuple` is a varadic type constructor that takes N types as parameters, and returns the type of an N-tuple with the given types as members.

This feature is not yet available in the prototype compiler, but is discussed as it is necessary for the SQL-interface. We leave the exact implementation of this feature as future work.

### 3.1.6 Match Expressions

Match expressions allows conditionally evaluate one of several expressions based on patterns. The cases of the match expression are considered from top to bottom,

and the branch at the first matching case is executed. The syntax for the match expressions is inspired by the similar construct in *Rust*[22].

As a simple example, a match expression can select a branch based on matching a value.

```
a := 2;
!printLn(match a {
    1 => "one";
    2 => "two";
    3 => "three";
    _ => "something else";
});
# "two"
```

_ is the wild card. It matches anything and acts as the default case in case no other cases match. Notice that if we put the wild card at the top of the case list, that branch will always be executed.

```
a := 2;
!printLn(match a {
    _ => "something else";
    1 => "one";
    2 => "two";
    3 => "three";
});
# "something else"
```

Match expressions can also pattern match on variants and on instantiated objects.

```
a := Some(2);
!printLn(match a {
    Some(2) => "some two";
    Some(_) => "something else";
    None    => "nothing";
});
# some two

Test := struct { a: int; }

b := Test { a = 2; };
!printLn(match b {
    { a = 2; } => "test two";
    { a = _; } => "test something else";
});
# test two
```

In cases where a case want to match on anything and the branch uses that matched value, one can use a pattern/template parameter in place of the wild card.

```
a := Some(2);
!printLn(match a {
    Some(1)  => "some one";
    Some($v) => toString(v);
    None     => "nothing";
});
# 2
```

### 3.1.7 Classes

Classes provides a way of declaring different behaviour or values depending on a set of parameters. As opposed to parametric polymorphism where the behaviour is the same for all compatible parameters, the programmer can make different behaviour for each set of parameters they choose to implement for. The condition for members of a class is that their members must conform to a specified type schema. Classes provides a replacement for ad-hoc polymorphism, such as function and operator overloading.

Classes define a set of common fields that must be implemented for each object it is defined on. Type classes are comprised of two parts: the class declaration and the instance implementations. The class declaration details what members are required and their types. The instance implementations binds the fields from its class for a particular object.

While classes are not limited to being defined over types, that is the most common use case. This case is often referred to as a type class.

Classes are declared using a **`class`** expression.

```
Eq := class[$T, $U] {
    op==: (T, U) -> Bool;
    op!= :~= (lhs: T, rhs: U) -> Bool
        => not(op==(lhs, rhs));
};
```

The programmer must specify an implementation for each set of parameters the class should respond to. Referencing a type class with parameters that have no matching implementation is a compilation error.

```
impl Eq[int, int] {
    op== := (lhs: int, rhs: int) -> Bool
        @native("int_int_eq");
};
```

### 3.1.8 Side Effects

Because this language is a pure functional programming language all function calls must be pure. This means they can have no side effects and that the return value of the function is exclusively determined by its arguments. This has advantages for code cleanliness, composability, and correctness, but does however make the language in itself inert. To allow a program to perform actions that has side effects, such as reading a file or interfacing with a database, we represent the actions in terms of monads. Monads, as described in section 2.1.2, are used to describe a sequence of operations that are executed within the context of the monad.

#### IO Monad

Like Haskell and Scala with Cats, this language has a monad named IO that represents operations with potential side effects on the world. A simple example of

using the IO monad is to print a line to the screen. The base module provides the `printLn` function that does exactly that.

```
printLn := (val: String) -> IO[Unit] => (...);
```

Calling `printLn("Hello, World!")` alone does not result in a line being printed to the screen. What this function does is return an object that when executed within the context of `IO` will print the given string. An `IO` monad can be evaluated at module initialization by using the `withIO` init monad as an init expression.

```
!withIO(printLn("Hello, World!"));
```

### Init Monad

The `Init` monad represents computations that happen as part of the program initialization. This is intended to be actions such as setting up systems and loading data required for the program to function.

Init monad calls start from the module's object instantiation using init expressions.

### Init expressions

Init-expressions provides a mechanism, similar to do notation, for evaluating monadic expressions. The init expression monads are evaluated as part of the instantiation of objects. Init expressions are syntactic sugar which translates to a series of bind calls.

When a composite data type contains one or more init expressions its instantiations turns into init monads. If the module's top-level scope contains init-expressions the resulting init-monad will be executed during the module's initialization.

```
random := (min: int, max: int) -> Init[int] => ...;

A := struct !Init {
    b: int = !random(0, 10);
};

b: Init[A] = A {};
```

```
A := struct {
    b: int;
};

b: Init[A] = random(0, 10) >>= value => return(A { b = value; });
```

As opposed to do-notation, init expressions does not provide a guarantee that the operations will be bound in the order they appear in the program text. This is in line with the design decision that members of data types can be declared in any order. The compiler only guarantees that the init expressions will be evaluated in an order such that their dependencies are fulfilled.

### Do-Expressions

While useful, writing monadic expressions using the `bind` function directly is very verbose and hard to read and write.

```
printAccount := (a: Account) -> IO[Unit]
    => bind(printLn(f"id: ${a.id}"),
        b => bind(printLn(f"name: ${a.name}"),
            b => bind(printLn(f"register: ${a.register}"),
                b => printLn(f"last login: ${a.lastLogin")
            )
        )
    );
```

We can improve this by creating the bind operator `>>=`.

```
op>>= := bind;

printAccount := (a: Account) -> IO[Unit]
    =>  printLn(f"id: ${a.id}")
    >>= (b => printLn(f"name: ${a.name}"))
    >>= (b => printLn(f"register: ${a.register}"))
    >>= (b => printLn(f"last login: ${a.lastLogin"));
```

Furthermore, since we are not interested in the value returned from the previous expression, we can create another operator, `>>`, which ignores the returned value.

```
op>> := (a: IO[$T], b: IO[$U]) -> IO[U]
    => a >>= dc => b;

printAccount := (a: Account) -> IO[Unit]
    => printLn(f"id: ${a.id}")
    >> printLn(f"name: ${a.name}")
    >> printLn(f"register: ${a.register}")
    >> printLn(f"last login: ${a.lastLogin");
```

Finally, we can introduce syntactic sugar to make the chain of monads even easier to read and modify. This syntax is based on the do-notation from Haskell[13], as well as the for comprehension from Scala[16].

```
printAccount := (a: Account) -> IO[Unit]
    => do {
        printLn(f"id: ${a.id}");
        printLn(f"name: ${a.name}");
        printLn(f"register: ${a.register}");
        printLn(f"last login: ${a.lastLogin");
    };
```

This syntax makes the monads very reminiscent of a series of imperative statements.

If we rewrite the example of reading a line from stdin and echoing that twice from section 2.1.2 from Haskell to this language, we can see the resulting expressions are very similar. Note that in order to execute the monad we wrap it in `!withIO();`.

```
!withIO(getLine >>= (ln => printLn(ln) >>= (_ => printLn(ln))));
!withIO(getLine >>= (ln => printLn(ln) >> printLn(ln)));
!withIO(do {
```

```
    ln <- getLine;
    printLn(ln);
    printLn(ln);
});
```

### 3.1.9 String Interpolation

Strings can be prefixed with a modifier that enables string interpolation. For instance, the base module provides the `f` prefix witch transforms the string into a default string formatter. The string interpolation is syntactic sugar that is expanded into a list of string literals and expressions.

```
w := "world";
!printLn(f"Hello, ${w}!");

!printLn(StringInterpolator["f"].compose([
    StringInterpolator["f"].fromLit("Hello, "),
    StringInterpolator["f"].fromExpr(w),
    StringInterpolator["f"].fromLit("!")
]));
```

These string interpolators can be declared by implementing the type class `StringInterpolator` for the string of the string prefix. A string interpolator provides functions for transforming input literals and expressions into segments of type `Part`, `fromExpr` and `fromLit`, and a function that concatenates the list of these segments to the final object of type `Out`.

The string formatter, `f`, concatenates each string literal and each expression cast to a string to form the formatted string.

```
StringInterpolator := class [prefix: String] {
    Out: Type;
    Part: Type;

    fromLit: (String) -> Part;
    fromExpr: ($U) -> Part;
    compose: (List[Part]) -> Out;
};

impl StringInterpolator["f"] {
    Out = String;
    Part = String;

    fromLit = id;
    fromExpr = string.ToString.toString;
    compose = string.join;
};
```

## 3.2 Formal Language Definition

This section details the formal definition of the language, including parsing rules and the rules for type resolution.

### 3.2.1 Language Grammar

An abbreviated formal description of the language syntax.

⟨*module*⟩ ::= ⟨*stmt-list*⟩

⟨*stmt-list*⟩ ::= ⟨*stmt-list*⟩ ⟨*stmt*⟩
| ⟨*empty*⟩

⟨*stmt*⟩ ::= ⟨*use-stmt*⟩ ';'
| ⟨*impl-stmt*⟩ ';'
| ⟨*assign-stmt*⟩ ';'
| ⟨*expr*⟩ ';'
| ';'

⟨*expr*⟩ ::= ⟨*expr1*⟩
| ⟨*func-decl*⟩
| ⟨*object-decl*⟩
| ⟨*object-inst*⟩
| ⟨*type-class-decl*⟩
| ⟨*match-expr*⟩

⟨*expr1*⟩ ::= ⟨*ident*⟩
| '$' ⟨*ident*⟩
| '!' ⟨*ident*⟩
| '_'
| ⟨*number-lit*⟩
| ⟨*string-lit*⟩
| ⟨*array-lit*⟩
| ⟨*cons-call*⟩
| ⟨*func-call*⟩
| ⟨*func-proto*⟩
| ⟨*do-expr*⟩
| ⟨*special*⟩
| ⟨*expr1*⟩ '.' ⟨*ident*⟩
| ⟨*expr1*⟩ ⟨*binary-op*⟩ ⟨*expr1*⟩
| '(' ⟨*expr*⟩ ')'

⟨*use-stmt*⟩ ::= 'use' ⟨*expr*⟩
| 'use' ⟨*expr*⟩ '.*'

⟨*assign-stmt*⟩ ::= ⟨*ident*⟩ ':' ⟨*expr*⟩
| ⟨*ident*⟩ ':' ⟨*expr*⟩ '=' expr
| ⟨*ident*⟩ ':=' expr
| ⟨*expr*⟩ '=' expr
| ⟨*ident*⟩ ':' ⟨*expr*⟩ '~=' expr

| ⟨*ident*⟩ ':∼=' expr
| ⟨*expr*⟩ '∼=' expr

⟨*mod-expr*⟩ ::= 'mod' ⟨*ident*⟩

⟨*type-class-decl*⟩ ::= 'class' '[' ⟨*template-decl-args*⟩ ']' '{' ⟨*stmt-list*⟩ '}'

⟨*template-params*⟩ ::= ⟨*template-params1*⟩
| ⟨*template-params1*⟩ ','

⟨*template-params1*⟩ ::= ⟨*template-params1*⟩ ',' ⟨*template-param*⟩
| ⟨*template-param*⟩

⟨*template-param*⟩ ::= '$' ⟨*expr*⟩
| ⟨*ident*⟩ ':' ⟨*expr*⟩

⟨*impl-stmt*⟩ ::= 'impl' ⟨*expr*⟩ '[' ⟨*func-args*⟩ ']' '{' ⟨*stmt-list*⟩ '}'

⟨*special-args*⟩ ::= ⟨*special-args1*⟩ ','
| ⟨*special-args1*⟩

⟨*special-args1*⟩ ::= ⟨*special-args1*⟩ ',' ⟨*special-arg*⟩
| ⟨*special-arg*⟩

⟨*special-arg*⟩ ::= ⟨*number-lit*⟩
| ⟨*string-lit*⟩

⟨*func-decl*⟩ ::= '(' ⟨*func-decl-params*⟩ ')' '⇒' ⟨*expr*⟩
| '(' ⟨*func-decl-params*⟩ ')' '→' ⟨*expr1*⟩ '⇒' ⟨*expr*⟩
| ⟨*ident*⟩ '⇒' ⟨*expr*⟩
| '(' ⟨*func-decl-params*⟩ ')' '→' ⟨*expr1*⟩ '@native(' ⟨*string-lit*⟩ ')'

⟨*func-decl-params*⟩ ::= ⟨*func-decl-params1*⟩
| ⟨*func-decl-params1*⟩ ','
| ⟨*empty*⟩

⟨*func-decl-params1*⟩ ::= ⟨*func-decl-params1*⟩ ',' ⟨*func-decl-param*⟩
| ⟨*func-decl-param*⟩

⟨*func-decl-param*⟩ ::= ⟨*ident*⟩ ':' ⟨*expr*⟩

⟨*func-proto*⟩ ::= '(' ⟨*func-proto-params*⟩ ')' '→' ⟨*expr1*⟩

⟨*func-proto-params*⟩ ::= ⟨*func-proto-params1*⟩
| ⟨*func-proto-params1*⟩ ','
| ⟨*empty*⟩

⟨*func-proto-params1*⟩ ::= ⟨*func-proto-params1*⟩ ',' ⟨*func-proto-param*⟩
| ⟨*func-proto-param*⟩

⟨*func-proto-param*⟩ ::= ⟨*expr*⟩

⟨*func-call*⟩ ::= ⟨*expr1*⟩ '(' ⟨*func-args*⟩ ')'

⟨*cons-call*⟩ ::= ⟨*expr1*⟩ '[' ⟨*func-args*⟩ ']'

⟨*func-args*⟩ ::= ⟨*func-args1*⟩
| ⟨*func-args1*⟩ ','
| ⟨*empty*⟩

⟨*func-args1*⟩ ::= ⟨*func-args1*⟩ ',' ⟨*func-arg*⟩

⟨*func-arg*⟩ ::= expr

⟨*object-decl*⟩ ::= 'struct' '{' ⟨*stmt-list*⟩ '}'

⟨*object-inst*⟩ ::= ⟨*expr1*⟩ '{' ⟨*stmt-list*⟩ '}'

⟨*variant-decl*⟩ ::= 'variant' '{' ⟨*variant-items*⟩ '}'

⟨*variant-items*⟩ ::= ⟨*variant-items1*⟩ ','
| ⟨*variant-items1*⟩
| ⟨*empty*⟩

⟨*variant-items1*⟩ ::= ⟨*variant-items1*⟩ ',' ⟨*variant-item*⟩
| ⟨*variant-item*⟩

⟨*variant-item*⟩ ::= ⟨*ident*⟩
| ⟨*ident*⟩ ⟨*expr*⟩

⟨*match-expr*⟩ ::= 'match' ⟨*expr1*⟩ '{' ⟨*match-cases*⟩ '}'

⟨*match-cases*⟩ ::= ⟨*match-cases*⟩ ⟨*match-case*⟩
| ⟨*empty*⟩

⟨*match-case*⟩ ::= ⟨*expr1*⟩ '⇒' ⟨*expr*⟩ ';'

⟨*do-expr*⟩ ::= 'do' '{' ⟨*do-expr-stmts*⟩ '}'

⟨*do-expr-stmts*⟩ ::= ⟨*do-expr-stmts*⟩ ⟨*do-expr-stmt*⟩
| ⟨*empty*⟩

⟨*do-expr-stmt*⟩ ::= ⟨*pattern*⟩ '<-' ⟨*expr*⟩ ';'
| ⟨*expr*⟩ ';'

⟨*array-lit*⟩ ::= '[' ⟨*array-body*⟩ ']'
| '[' ⟨*array-body*⟩ ',' ']'
| '[' ⟨*array-body*⟩ ',' '...' ']'

⟨*array-body*⟩ ::= ⟨*array-body*⟩ ',' ⟨*array-item*⟩
| ⟨*array-item*⟩

⟨*array-item*⟩ ::= ⟨*expr*⟩

The ⟨*binary-op*⟩ non-terminal matches any binary operator. Currently +, -, *, /, ==, !=, <, >, <=, >=, &&, ||, &, |, >>, <<, and >>= are recognized as binary operators.

The ⟨*ident*⟩ non-terminal matches any user-provided word that is not the name of a terminal, as well as any binary operator prefixed with 'op', for instance 'op+'.

### 3.2.2 Core Language

The language is comprised of a set of core operations, which is what the compiler works on for the semantic analysis and code generation. The syntactic sugar is built on top of the core language.

The core language contains the following expressions.

- **Function declaration.**
  `(p1: T1, ..., pn: Tn) -> Tr => e`
  Declares a function with parameters `p1 ... pn` with respective types `T1 ... Tn`, return type `Tr`, and body `e`.
- **Native function declaration.**
  `(p1: T1, ..., pn: Tn) -> Tr @native("function name")`
  Declares a function with parameters `p1 ... pn` with respective types `T1 ... Tn` and return type `Tr` which calls the native function named "`function name`". See section 5.5.1.
- **Function type declaration.**
  `(T1, ..., Tn) -> Tr`
  Declares a function type with parameters of type `T1 ... Tn` and return type `Tr`.
- **Template Declaration.**
  `[p1: T1, ..., pn: Tn] e`
  Declares a parametric template over object `e` The parameters `p1 ... pn` of respective type `T1 ... Tn` are provided to the expression in `e`.
- **Function call.**
  `fn(a1, ..., an)`
  Calls the function `fn` with arguments `a1 ... an`.
- **Construct object.**
  `cons[a1, ..., an]`
  Packs the object constructor `cons` with arguments `a1 ... an`.

- **Instantiate object.**
  `inst { m1 = e1; ... mn = en }`
  Packs the object `inst` with extra binds of members `m1 ... mn` to expressions `e1 ... en`.
- **Access.**
  `target.name`
  Unpacks the member `name` from the composite object `target`.
- **Literal.**
  Any object.
- **Native literal.**
  `@native("literal name")`
  References a native object named "`literal name`" registered by the native module extension (section 5.5.1).
- **Lookup.**
  Any identifier.
- **Module.**
  `mod target`
  Returns the top level object for the module named `target`.
- **Match.**
  `match condition { pat1 => expr1; ... patn => exprn; }`
  Executes the first of `expr1 ... exprn` where the corresponding `pat1 ... patn` matches `condition`.
- **Wild card.**
  `_`
  Represents the pattern that matches anything.
- **Init expression.**
  `!exprid`
  References the init expression with id `exprid` of the parent composite.
- **Type class.**
  `class[p1: T1, ..., pn: Tn] { m1: M1; ... mk: Mk; }`
  Registers a type class with parameters `p1 ... pn` of respective type `T1 ... Tn` and with members `m1 ... mk` of type `M1 ... Mk`.
- **Composite.**
  `struct { statements }`
  Registers a composite data type with statements `statements`. These statements can be:
  - Member declaration (section 3.1.2).

```
1 m: T;
2 m: T = e;
3 m: T ~= e;
4 m := e;
5 m :~= e;
```

    Declares a member `m` on the composite data type with type `T` and value `e`. If `T` is not present the type is derived from the value. If the assign-

ment operator is prefixed with tilde (~=) the bound value is overridable.

- **Member assignment** (section 3.1.2).

```
m = e;
m ~= e;
```

Binds member `m` to value `e`. If the assignment operator is prefixed with tilde (~=) the bound value is overridable.

- **Use** (section 3.1.2).

```
use e;
use e as name;
use e.*;
```

(1) Put the value `e` with name `e` in the current scope. This is only available if `e` is a lookup, module, or access expression. (2) Put the value `e` in the current scope with name `name`. (3) Put all members of the expression `e` in the current name space. This is only available if the value returned by `e` is a composite.

- **Init expression**.

`!expr`

Evaluates the init monad provided by `expr` during module initialization. The result of this monad can be referenced using the init expression node.

- **Type class implementation**.

`impl target[a1, ..., an] { binds }`

Creates an implementation for the type class `target` with arguments `a1 ... an` where the type class members are bound by the `binds`.

- **Variant.**

`variant { opt1 T1, ... optn Tn }`

Registers a variant type with options `opt1 ... optn` with respective types `T1 ... Tn`. Each of the types are optional.

### 3.2.3 Syntactic Sugar

To make the core language easier to use, syntactic sugar is introduced to automate or provide more convenient syntax for commonly used patterns.

#### Infix Binary Operators

Infix binary operators are just function calls with the left and right hand side as its two arguments. This syntax sugar is resolved during parsing. Any binary operation, such as `2 + 3`, is converted into a function call. The target function of the call is a variable with the operator symbol prefixed with 'op' as name. For instance, the previous addition is translated into `op+(2, 3)`.

### Lists

Lists are a simple rewrite syntax sugar, and are handled at the transition from syntax tree to abstract syntax tree. A list is simply a linked list specified by the `cons` function. `cons` is a tuple of the current value, the head, and the rest of the list, the tail. A list literal, $[e_1, e_2]$ is translated into $\text{cons}(e_1, \text{cons}(e_2, \text{nil}))$. A list pattern can also match on only the first few elements. Such a pattern, $[e_1, e_2, \dots]$ is translated into $\text{cons}(e_1, \text{cons}(e_2, \_))$, where _ is the wild card.

### Do Notation

Do notation is a simple rewrite syntax sugar. It is exactly equivalent to calling bind (the >>= or >> operators) on its statements.

If we care about the returned value from a monad, we can write a statement like `a <- getLine;`. This will be translated to `getLine >>= (a => <next>)`. If we do not care about the returned value, we can write `getLine;`. This will be translated to `getLine >> <next>`.

### Object Instantiation with Init Expressions

Object instantiation with init expressions requires knowledge of the object to instantiate and the types of its members. It is therefore performed very late in the compile pipe line. In the current compiler implementation it is resolved as part of code generation.

When an object with init expressions is instantiated it does not result in the object itself, $T$, but rather an init-monad with itself, Init$[T]$. The instantiation is structured as a series of binds of the init expression monads, with the non-init expressions and bound members being passed as closures to the bottom-most bind, where the packed object is returned. The init-monad binds are structured in such a way that all their dependencies have been fulfilled before the monad itself is evaluated.

The following example illustrates how an object with init expressions is translated to the core syntax.

```
1  random := (min: int, max: int) -> Init[int] => ...;
2
3  Test := struct {
4      a := !random(b, c);
5      b := !random(x, x+5);
6      c := !random(b, y);
7
8      x: int;
9      y: int;
10 };
11
12 test: Init[Test] = Test { x = 2; y = 5};
13
14 test: Init[Test] =
15     do {
16         x <- return(2);
```

```
17          e2 <- random(x, x+5);
18          b <- return(e2);
19
20          y <- return(5);
21
22          e3 <- random(b, y);
23          c <- return(e3);
24
25          e1 <- random(b, c);
26          a <- return(e1);
27
28          return(Test { a = a; b = b; x = x; y = y; });
29      };
```

In the current compiler implementation, as this transformation happens at the code generation stage, such an instantiation does not bind return monads for the member binds and expression evaluations. In stead, they are expressed as imperative assignments and passed as closures to the next function. The resulting code, however, is functionally equivalent.

#### Object Constructor Decay

As parametric functions are represented as object constructors, the user must annotate references to parametric values with `[]`. This often happens in cases where the parameters of the constructor can be inferred, and the extra notation serves no purpose. Object constructor decay automatically transforms uses of object constructors into instances of the constructed object in cases where the object constructor itself is not expected.

This operation is applied during the type check phase. Any AST node that returns an object constructor is transformed into a construct node with the original node as its target function. For a discussion about the implementation of this feature, see section 5.2.4.

### 3.2.4 Composite Data Types

We present the algorithm used in the current compiler prototype implementation for verifying the binds of the data type and ordering the operations that constitute its instantiation. The resulting actions from the instantiation ordering algorithm typically is used to generate byte code.

The composite data type for type $t$ is a tuple $C_t = (M, E, B, \tilde{B})$. $M$ is a set of the members of the data type, $M = \{m_1 : t_{m_1}, \ldots, m_n : t_{m_n}\}$, with $m_i : t_{m_i}$ being the member named $m_i$ of type $t_{m_i}$. $E$ is a set of expressions associated with the data type, $E = \{e_1 : t_{e_1}, \ldots, e_k : t_{e_k}\}$, with $e_i : t_{e_i}$ being the expression $e_i$ of type $t_{e_i}$. $B$ is a set of binds that bind parts of expressions from $E$ to parts of members in $M$ (equation 3.3). $\tilde{B}$ is a set of binds that are overridable. $\tilde{B}$ has the same domain as $B$.

The set of descendants for a member $m : t$ can be determined by $desc(m : t)$ (equation 3.1). A descendant is a member of a composite member or its children,

or the top member itself. For instance in the following listing, $desc(val : A) = \{a : \textbf{int}, b : \textbf{int}, x : \textbf{B}, val : \textbf{A}\}$.

```
1 A := struct { a: int; x: B; };
2 B := struct { b: int; };
3
4 val := A { (...) };
```

$$desc(m : t) = \{m : t\} \cup \begin{cases} \bigcup_{d:t_d \in C_t.M} desc(d : d_t) & \text{if type } t \text{ is composite } C_t \\ \emptyset & \text{otherwise} \end{cases} \tag{3.1}$$

As a convenience, the descendants of a composite data type, $C_t$ can be determined with $desc(C_t)$ (equation 3.2).

$$desc(C_t) = \bigcup_{m:t_m \in C_t.M} \{m : t_m\} \cup desc(m : t_m) \tag{3.2}$$

$$B \subseteq \{(d : t_d, e : t_e) \mid (d : t_d, e : t_e) \in (desc(C_t) \times (\bigcup_{a:t_a \in C_t.E} desc(a : t_a))), t_d = t_e\} \tag{3.3}$$

The set of all bind targets for a set of binds, $B$, can be determined with the function $targets(B)$. This is useful for considering what descendants are touched by each bind, while keeping track of the duplicate binds for validation.

$$targets(B) = \bigcup_{(m:t_m, e:t_m) \in B} desc(m : t_m) \times \{e : t_m\}$$

At the site of instantiation the use can provide additional expressions and binds, $E_1$ and $B_1$ respectively. There are several requirements that must be fulfilled for a object instantiation to be valid.

1. All members must be bound, either overridably or not. (Equation 3.4)
2. No member can be bound non-overridably more than once. (Equation 3.5)
3. No member can be bound overridably more than once. (Equation 3.6)

For the validity test equations (3.4, 3.5, and 3.6) of composite $C_t = (M, E, B, \tilde{B})$ with extra binds, $B_1$, and extra expressions, $E_1$, we will use $B = C_t.B \cup B_1$ and $\tilde{B} = C_t.\tilde{B}$.

$$\emptyset \Leftrightarrow C_t.M - \{m : t \mid (m : t, e : t) \in targets(B \cup \tilde{B})\} \tag{3.4}$$

$$\begin{aligned} \emptyset \Leftrightarrow \{m : t \mid & \\ & (m : t, e : t) \in targets(B), \\ & m : t \in (targets(B) - \{(m : t, e : t)\})\} \end{aligned} \tag{3.5}$$

$$\begin{aligned}\emptyset \Leftrightarrow \{m : t \mid & \\ & (m : t, e : t) \in targets(\tilde{B}), \\ & m : t \in (targets(\tilde{B}) - \{(m : t, e : t)\})\}\end{aligned} \tag{3.6}$$

Provided all the requirements above are satisfied, a bind solution can be determined by *solve* (equation 3.7).

$$\begin{aligned}solve(C_t, B_1) = & \, targets(C_t.B \cup B_1) \\ & \cup \{(m : t, e : t) \mid \\ & \quad (m : t, e : t) \in targets(C_t.\tilde{B}), \\ & \quad m : t \notin \{m_1 : t_1 \mid (m_1 : t_1, e_1 : t_1) \in targets(C_t.B \cup B_1)\}\}\end{aligned} \tag{3.7}$$

Let $dep(e : t) : C_t.E \to \mathscr{P}(desc(C_t.M))$ be a function of the members that expression $e : t$ is dependent on.

The directed dependency graph that dictates the order of which the expressions should be evaluated is $G = (C_t.E, D(C_t))$, where $C_t.E$ is the set of vertices of the graph and $D(C_t)$ is the set of edges (dependencies).

$$D(C_t) = \bigcup_{e:t \in C_t.E} (dep(e : t) \times \{e : t\})$$

Given such a dependency graph, a final order of evaluation can be derived for the expressions using a topological sort. In addition to evaluating the expressions, the value of expressions must be unpacked onto the members who are bound to them, and the composite members must be packed with their now evaluated members.

There are 4 operations that are used to instantiate objects:

1. *EXPR*$(e : t)$: Evaluates the expression $e : t$, taking its dependencies, $dep(e : t)$, as arguments.
2. *INIT_EXPR*$(e : t)$: Evaluates the expression returning an init monad and binds the monad with a lambda function of the remaining actions.
3. *BIND*$(m : t)$: Assigns the part of member $m : t$'s bound expression to the member. After this point the member is available as a dependency to other expressions.
4. *PACK*$(m : t)$: Packs the composite member $m : t$ with its members. All the members $desc(m : t)$ are expected to be available, either having been bound or packed.

Given a composite data type, $C_t$, a set of extra binds, $B_1$, and a set of extra expressions, $E_1$, the list of operations required to instantiate the object is given by the following algorithm.

1. Let $S := solve(C_t, B_1)$ be the bind solution.

2. Let $G := (C_t.E, D(C_t))$ be the dependency graph.
3. Let $O_m := \emptyset$ be the set of members that have been packed.
4. Let $A := [\,]$ be the list of operations required to instantiate the object.
5. Topologically sort the graph $G$ into the list $L$.
6. For each expression $e : t \in L$:
    a. For each composite member

$$m : t_m \in \{m : t_m \mid m : t_m \in dep(e : t), \{m : t_m\} \subset desc(m : t_m)\}$$

        i. If $m : t_m \notin O_m$:
            A. Append $PACK(m : t_m)$ to $A$.
            B. Add $m : t_m$ to $O_m$.
    b. Append $EXPR(e : t)$ to $A$.
    c. For each terminal member

$$m : t_m \in \{m : t_m \mid m : t_m \in dep(e : t), \{m : t_m\} = desc(m : t_m)\}$$

        i. If $m : t_m \notin O_m$:
            A. Append $BIND(m : t_m)$ to $A$.
            B. Add $m : t_m$ to $O_m$.
7. For each composite member that is not yet visited

$$m : t_m \in \{m : t_m \mid (m : t_m, e : t_m) \in S, \{m : t_m\} \subset desc(m : t_m), m : t_m \notin O_m\}$$

    a. Append $PACK(m : t_m)$ to $A$.
    b. Add $m : t_m$ to $O_m$.
8. Return $A$.

### 3.2.5 Typing Rules

The type system and notation is based on "Types and Programming Languages"[4]. This system of typing rules describe both how the types of expressions relate to each other, as well as how the names are being propagated through the expressions.

- `LIT`: For a constant value $v : T$

$$\Gamma \vdash v : T$$

- `LOOKUP`:

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

- `ACCESS`:

$$\frac{\Gamma \vdash x : \text{struct}\{m : T; \ldots\}}{\Gamma \vdash x.m : T}$$

- `FUNC`: Abstraction

$$\frac{\Gamma, p_1 : P_1, \ldots, p_n : P_n \vdash e : R}{\Gamma \vdash (p_1 : P_1, \ldots, p_n : P_n) \to R \Rightarrow e : (P_1, \ldots, P_n) \to R}$$

- `FUNC_NATIVE`:

$$\frac{P_1 : \text{Type}, \ldots, P_n : \text{Type} \in \Gamma \qquad R : \text{Type} \in \Gamma}{\Gamma \vdash (p_1 : P_1, \ldots, p_n : P_n) \to R \ @\text{native}(name) : (P_1, \ldots, P_n) \to R}$$

  *name* is a literal string referring to a C-function registered to the system from the current module.
- `FUNC_TYPE`:

$$\frac{P_1 : \text{Type}, \ldots, P_n : \text{Type} \in \Gamma \qquad R : \text{Type} \in \Gamma}{\Gamma \vdash (P_1, \ldots, P_n) \to R : \text{Type}}$$

- `CALL`: Application

$$\frac{\Gamma \vdash f : (p_1, \ldots, p_n) \to r \qquad \Gamma \vdash a_1 : p_1, \ldots, a_n : p_n}{\Gamma \vdash f(a_1, \ldots, a_n) : r}$$

- `COMPOSITE`:

$$\frac{T_1 : \text{Type}, \ldots, T_n : \text{Type} \in \Gamma, m_1 : T_1, \ldots, m_n : T_n}{\Gamma \vdash \text{struct}\{m_1 : T_1; \ldots; m_n : T_n;\}}$$

- `INST`: Instantiate

$$\frac{\begin{array}{c} C : \text{struct}\{m_1 : T_1; \ldots; m_n : T_n\} \in \Gamma \\ \Gamma, m_1 : T_1, \ldots, m_n : T_n \vdash v_1 : T_1, \ldots, v_n : T_n \end{array}}{\Gamma \vdash C\{m_1 = v_1; \ldots; m_n = v_n;\} : T}$$

- `VARIANT`:

$$\frac{\{T_1 : \text{Type}, \ldots, T_n : \text{Type}\} \subseteq \Gamma}{\Gamma \vdash \text{variant}\{v_1 T_1; \ldots; v_n T_n;\} : Type}$$

- `TEMPL`: Template

$$\frac{\Gamma, p_1 : P_1, \ldots, p_n : P_n \vdash e : T}{\Gamma \vdash [p_1 : P_1, \ldots, p_n : P_n]\ e : [P_1, \ldots, P_n] \to T}$$

- `CONS`: Constructor

$$\frac{\Gamma \vdash c : [p_1 : P_1, \ldots, p_n : P_n] \to T}{\Gamma \vdash c[a_1, \ldots, a_n] : [p_1 \mapsto a_1, \ldots, p_n \mapsto a_n]T}$$

## 3.3 Standard Module

The `base` module provides standard functionality that is commonly required for all other modules.

### 3.3.1 `prelude`

The `prelude` name space is `used` by default by every module. This is to ensure all names required for the basic functionality of the compiler is available, as well as to provide easy access to commonly used features.

`Type`: `Type`
The type of types.

`int`: `Type`
The type of 64 bit signed integers.

`String`: `Type`
The type of UTF-8 encoded strings.

`Unit`: `Type`
The unit type. Contains only the value `unit`

`unit`: `Unit`
The unit value.

`Init`: `[T: Type] -> Init[T]`
Type constructor for the init monad.

`Bool`: `Type`
The boolean variant type which contains `true` and `false`.

`true`: `Bool`
The true boolean value.

`false`: `Bool`
The false boolean value.

### 3.3.2 `init`

The init module contains functionality related to the init monad.

`return`: `(val: $T) -> Init[T]`
Wrap the value val in the init monad.

`bind`: `(a: Init[$T], f: (T) -> Init[$U]) -> Init[U]`
Sequence the monad returned by f to be evaluated after a.

`withIO`: `(a: io.IO[$T]) -> Init[T]`
Execute an IO monad as part of the module initialization.

`printLn`: `(val: String) -> Init[Unit]`
A convenience for printing a line of text to the terminal within the context of the Init monad. Equivalent to `withIO(io.printLn(val))`

### 3.3.3 `io`

The IO module contains functionality related to the IO monad.

`IO`: `[T: Type] -> IO[T]`
Type constructor for the IO monad.

`return`: `(val: $T) -> IO[T]`
Wrap the value val in the init monad.

`bind`: `(a: IO[$T], f: (T) -> IO[$U]) -> IO[U]`
Sequence the monad returned by f to be evaluated after a.

`printLn`: `(val: String) -> IO[Unit]`
Writes the text val to stdout.

### 3.3.4 `list`

`List`: `[T: Type] -> List[T]`
Type constructor for the List type.

`nil`: `List[$T]`
Empty list.

`cons`: `(head: $T, tail: List[T]) -> List[T]`
A linked list with head and tail.

`head`: `(a: List[$T]) -> Maybe[T]`
Returns some head if the list is not nil, none otherwise.

`tail`: `(a: List[$T]) -> List[T]`
Returns the tail of the list. If the list is empty, it returns the empty list.

`map`: `(fn: (T) -> $U, a: List[$T]) -> List[U]`
Applies the function fn to each element of list a and returns the output as a new list.

`foldl`: `(fn: (T) -> $U, a: List[$T]) -> List[U]`
Applies the function fn to each element of list a and returns the output as a new list.

### 3.3.5 `composite`

This API is not implemented in the current compiler prototype.

`Composite`: `class [T: Type]`
Type class that is automatically implemented for all composite (struct) data types.

`Composite[T].tupleType`: `$U`
The type of a tuple that contains anonymous versions of the members of data type T in order of definition. For example:

```
Composite[struct {a: int, b: String}].tupleType = Tuple(int, String)
```

`Composite[T].fromTuple`: `(Composite[T].tupleType) -> T`
Instantiates the composite object from the tuple.

# Chapter 4

# SQL-Interface

This report will consider an SQL-interface that allows users to dispatch SQL queries and receive sets of results back. The goal of this interface is to make it easy for users to specify queries, pass parameters to said query, and receive rows from the query for further processing. By easy it is implied that the compiler should provide useful error messages and hints in cases where the user have made a mistake in the program. The interfaces should fit into the functional and type safe nature of the language.

The SQL examples are written for PostgreSQL.

## 4.1 Database Connection

Connecting to the database, like other operations with side effects, is encapsulated in an IO-monad.

All operations on the database requires the `Connection` handle, which is accessible in the context of an IO monad, using a specific connect function for the database they are connecting to. The connection object creates a uniform interface for querying any database.

```
1 connect := (conStr: String) -> Connection => ...;
```

Using this interface, the user can connect to, for instance, a PostgreSQL database using the following snippet.

```
1 mod sql;
2
3 db := !withIO(sql.postgresql.connect("dbname=test"));
```

## 4.2 Simple Query Interface

This first implementation of an SQL interface provides a `query` monad. It will, when executed, send the provided query string and parameter strings to the database back end and return a list of rows, each consisting of a list of fields represented as strings.

```
query := (db: Connection,
          q: String,
          params: List[String]) -> IO[List[List[String]]] => ...;
```

The user can now, for example, query the database as follows:

```
rows := !withIO(query(db, "select * from account where id=$1"), ["3"]);
```

While this query interface is easy to implement into the language and allows the user to make any query to the database, it does not fulfill all the requirements put forth in the introduction. Both the parameters and the output of the query are passed as strings, and the conversion to the correct data types, as well as unwrapping the tuple lists, must be done manually. This is both error prone and tedious.

## 4.3 Query Interface

We propose this query interface as a layer on top of the previous query monad, which is inspired by the Scala library Doobie[17]. This API simplifies the process of querying the database while providing greater type safety.

This system is not yet implemented in the prototype sql module due to certain compiler features that are not complete. See section 6.2.1 for a discussion about this.

The first part of executing a query is to input the SQL that should be evaluated. A simple and naïve approach would be to use the default string formatter to interpolate the variables directly into the SQL statement. In this implementation this can be acceived by the following expressiont.

```
id := 2;
q: Query = sql.markSafe(f"select * from account where id=${id}");

!withIO(sql.query(db, q));
```

This poses an issue when using untrusted parameters. A malicious user would be able to inject rogue SQL into the statement in an SQL-injection attack. Because of this we instead use the `sql`-formatter. The query is bundled with the query parameters in the `Query` data type, which is passed to the database back end for escaping and evaluation. This data structure stores the parameters next to the string, and inserts place holder variable references in the query string. When the query is executed, the parametrized query string along with the list of parameters is passed to the backend for escaping and evaluation.

```
id := 2;
q: Query = sql"select * from account where id=${id}";

!withIO(sql.query(db, q));
```

This query will be translated into the following SQL with `2` being passed as parameter 1.

```
select * from account where id=$1::integer;
```

Because we know what type the user passed as each parameter we can explicitly tag the usages in the generated SQL. This provides the database engine with more information about the user's intent and can provide error messages in case of type mismatch.

Sometimes the programmer wishes to compose a query of multiple parts. This can be accomplished safely with SQL fragments. Any such query string prefixed by sqlf is an SQL fragment, and when interpolated like a variable, the fragment is inserted literally, and the parameters are renamed to remain unique in the new query.

```
id := 2;
name := "test";

nameFilter := sqlf"name=${name}"
q := sql"select * from account where id=${id} and ${nameQuery}";

!withIO(sql.query(db, q));
```

This query translates into the following SQL statement with parameter 1 as `2` and parameter 2 as `"test"`.

```
select * from account where id=$1::integer and name=$2::text;
```

The safe version of the query is designed to be much simpler to use than the unsafe version presented first. Because the query-function only accepts Query objects as statements, one would have to manually tag the potentially compromised string as safe. The intention is to make specifying safe programs easier than specifying potentially dangerous ones. Of course, some times the programmer wants to escape the safety of the automatic escaper. In this case they can resort to the `markSafe` function. This makes it clear that this location has a higher potential for vulnerabilities and indicates extra scrutiny from the programmer is in order to ensure the program is still safe.

When a `Query` object has been created the next step is to pass the query to the database. This can be accomplished by one of three monads: `query`, `queryOne`, or `execute`. `query`, when evaluated, returns a list of rows. `queryOne` expects and returns only one row as a result, and raises an error if the query produces more than one row. `execute` is intended for queries that does not return any rows, such as updates, inserts, and deletes. This monad returns the number of rows affected by the command.

For the `query` and `queryOne` monads the final step is to transform the fields of the returned tuples into native data types.

After the database back end has processed the query the resulting tuples are returned to the application. At this point the tuples must be transformed back into data that is easily usable within the language. The simple query interface presented first solved this by returning a list the fields as strings for each row. While simple to implement, this puts the burden on the application developer to both unpack the list into a usable data structure, as well as converting the strings into their correct data types.

This improved interface automates this task by letting the user specify what data type the returned tuples should have, and automatically transforming the rows into that type. The simplest form of this mechanism can be seen with a tuple. The `query` and `queryOne` must be qualified with the type of the tuples they are expected to return.

```
!withIO(do {
    db <- postgresql.connect();
    rows <- query[(int, String)](db, sql"select id, name from account");
    mapM_(((id, name)) => printLn(f"${id}: ${name}"), rows);
});
```

The query is expected to return rows of fields with types matching the members of the tuples, in the same order. A run-time error is returned if there is a type mismatch.

This interface is useful for ad-hoc queries, but if the goal of the query is to fill a data structure with data the programmer still has to transform the data into the final data type for each query. In that case the programmer can implement the `SQLTuple` type class on the target data type.

```
SQLTuple := class [T: Type] {
    tupleType: Type;
    fromTuple:  (tupleType) -> T;
};

Account := struct {
    id:   int;
    name: String;
};

impl SQLTuple[Account] {
    fieldTypes = (int, String);
    fromTuple  = ((i, n): Tuple(int, String)) -> Account
        => Account { id = i; name = n; };
};

!withIO(do {
    db <- postgresql.connect();
    rows <- query[Account](db, sql"select id, name from account");
    mapM_(a => printLn(f"${a.id}: ${a.name}"), rows);
});
```

the data type transform can be automated further in cases such as this where the members of the target data type match both the type and the order of the returned tuples. By introducing a default implementation of SQLTuple that matches on any composite data type, the developer can automatically get back the final data type without having to implement a special transformation.

```
impl SQLTuple[$T] {
    fieldTypes = Composite[T].tupleType;
    fromTuple = Composite[T].fromTuple;
};

Account := struct {
    id:   int;
    name: String;
```

```
};

!withIO(do {
    db <- postgresql.connect();
    rows <- query[Account](db, sql"select id, name from account");
    mapM_(a => printLn(f"${a.id}: ${a.name}"), rows);
});
```

### 4.3.1 Type Mapping

Type mapping concerns how data stored in the database is translated into language-native types that can be reasoned about by the compiler.

The type class `SQLType[$T]` is responsible for performing this type mapping. Any type that can be pased to and from the SQL-backend is implemented for this type class.

```
SQLType := class[$T] {
    toSQL:      (T) -> String;
    fromSQL:    (String) -> T;
    typeSuffix: Maybe[String];
};

impl SQLType[String] {
    toSQL       = v => v;
    fromSQL     = v => v;
    typeSuffix  = None;
};

impl SQLType[String] {
    toSQL       = v => base.toString(v);
    fromSQL     = v => base.toInt(v);
    typeSuffix  = Some("integer");
};
```

The function `toSQL` is responsible for converting the native object into a string that can be passed to the database backend, and `fromSQL` takes a string returned from the database and converts it back into a native object. For providing additional type safety in the SQL query, types can optionally specify a suffix that is inserted into the query to indicate what type the user intended it to be. For PostgreSQL, such a parameter is translated into, for example `$1::integer` for parameter 1 of type int.

### 4.3.2 Query String Interpolation

As described above, we provide a string interpolator that returns a query object that is ready to be sent to the database. Here we will investigate how this mechanism functions.

The `sql` and `sqlf` prefixed strings are translated into lists of expressions returning parts of the expressed query, similarly to the string formatter prefix `f` (see section 3.1.9). The difference from the string format interpolator is that the parts are `QueryPart` objects. These object represent either a literal, sql-safe string or a parameter that will be passed to the underlying database for escaping.

```
QueryPart := variant {
    literal String,
    parameter struct { val: String; typeName: Maybe[String]; },
};
```

A query, such as `sql"select * from account where name=${aName} and id=${aId}"` is translated into an expression such as the following.

```
StringInterpolator["sql"].compose([
    StringInterpolator["sql"].fromLit("select * from account where name="),
    StringInterpolator["sql"].fromExpr(aName),
    StringInterpolator["sql"].fromLit(" and id="),
    StringInterpolator["sql"].fromExpr(aId)
])
```

The proposed implementation of the string interpolator class is presented in the following listing.

```
ComposeState := struct {
    query :~= "";
    params: List[(int, String)] ~= nil;
    nextParam :~= 1;
};

makeTypeSuffix := (suf: Maybe[String]) -> String
    => match suf {
        Some($name) => "::${name}";
        None => "";
    };

composePart := (s: ComposeState, p: QueryPart) -> ComposeState
    => match p {
        literal($lit) => s {query = s.query+lit;};
        parameter{$val, $typeSuffix} => s {
            query = s.query+f"\$${s.nextParam}${makeTypeSuffix(typeSuffix)}";
            params = cons((s.nextParam, val), s.params);
            nextParam = s.nextParam+1;
        };
    };

impl StringInterpolator["sql"] {
    Out  = Query;
    Part = QueryPart;

    fromLit = QueryPart.literal;
    fromExpr = (val: $T) -> QueryPart
        => QueryPart.parameter {
            val = SQLType.toSql(val);
            typeName = SQLType[T].typeSuffix;
        };

    compose = (parts: List[QueryPart]) -> Query
        => Query {
            {query; params;} = foldl(composePart, ComposeState {}, parts);
        };
};
```

## 4.4 Atomicity & Transactions

By default, each query is executed in a separate transaction. Multiple queries can be executed within the same transaction using the `transaction` monad. Operations inside this monad is wrapped with begin and commit, or rollback on error.

```
transaction := (db: Connection, IO[$T]) -> IO[T] => ...;
```

An example of using this API follows. Notice that both the `INSERT` and the `SELECT` statements will be executed within the same transaction by being wrapped with `BEGIN` and `COMMIT`. If either of the queries fail, the transaction will automatically call `ROLLBACK`.

```
!withIO(do {
    db <- postgresql.connect();
    count <- transaction(db, do {
        exec(db, sql"insert into account (name) values (\"test\")");
        queryOne(db, sql"select count(*) from account");
    });
    printLn(f"${count}");
});
```

Nesting these transaction blocks causes savepoints to be used in plack of `BEGIN` and `COMMIT`.

# Chapter 5

# Compiler Design

This chapter will discuss the current implementation of the compiler for the language described in chapter 3. Certain parts of the compiler will deviate from the language description due to challenges of implementation. These deviations will be discussed where applicable and summarized in chapter 6

The compiler is structured in a conventional form, with a clear pipeline from syntactic analysis through semantic analysis and finally to code generation. This chapter will describe how these components are designed and interact to form a complete compiler.

## 5.1 Syntactic Analysis

The syntactic analysis step is a conventional tokenizer and parser setup. The output of this step is a syntax tree which is a tree-based representation of the program. See section 3.2.1 for a listing of the parser grammar rules.

The current compiler implementation uses *re2c* to generate the tokenizer, and *GNU Bison* as a compiler compiler for the parser. Re2c is a free and open-source lexer generator[23]. This system serves as the tokenizer for the compiler. GNU Bison is a free and open-source parser generator[24]. Bison generates the LALR(1) parser for the language. The output from the parser is a syntax tree that literally describes the structure of the program the user specified. As a final step the syntax tree is converted into an abstract syntax tree (AST) which will be used for the semantic analysis. The abstract syntax tree is a much terser representation of the program where most syntactic sugar is represented in terms of the core language.

## 5.2 Semantic Analysis

The semantic analysis attempts to understand and validate the statements and expressions made by the user. This step takes as input an AST and augments this with information about objects and types.

The Abstract Syntax Tree (AST) acts as the primary representation of the input program through the semantic analysis, and represents the core language (see section 3.2.2).

### 5.2.1 Composite Data Type

The composite data type is the primary user definable data type in the language. This system is used as the base for both data structures and for modules and name spaces. In fact, modules and name spaces are just structs with some additional rules.

Because of the design decision to make all values first-class citizens and to not impose requirement on the order of statements, the task of resolving member types and values requires ordering the operations required such that they have all data the values required to resolve.

At its core the composite data type system is a job system. Different tasks required to finalize the members and their values are placed into a dependency graph and dispatched in a topologically sorted order. The system is finished when there are no more jobs that can be dispatched, either because all jobs have been dispatched or because no remaining jobs have all their dependencies fulfilled. The latter case indicates one or more cycles in the job dependency graph. This triggers a failure of the composite system.

Except for modules and their name spaces, the composite solver is evaluated once for each composite declaration. Each module and all its name spaces are batched together. This is to facilitate type classes requiring all their implementations to be discovered before it can be used (see section 5.2.1).

#### Member Declaration and Assignment

Member declarations and assignments are the central feature of the composite data type. These statements allows the user to create members on the final data type that are accessible to other members or from the outside.

These statements can have several forms:

```
1  # Declaration only.
2  a: int;
3
4  # Assignment to an existing member a.
5  a = 2;
6  a ~= 2;
7
8  # Declaration with bind.
9  a: int = 2;
10 a: int ~= 2;
11
12 # Declaration with implicit type from the bound value.
13 a := 2;
14 a :~= 2;
```

Adding the tilde (~) before the assignment operator (=) marks the bind as overridable. Any other assignment statement that binds this member will take precedence.

```
a : int = 2 ;
|   |     |
(1) (2)   (3)
```

There are up to three distinct parts to a declaration or assignment: (1) The target, (2) the explicit type, and (3) the bind value. The declaration-only statement consists of a target and a type. The assignment and the declaration with implicit type has a target and a value. The declaration with a bind has all three parts.

Figure 5.1 presents the dependency graph for a member declaration with an explicit type and a value. The dashed arrow indicates the job dependency on the target job is imposed by the source job. The diamond represents all jobs the current expression is dependent on. For a member declaration without a bind, such as `a: int;` the value group of jobs would be omitted. For a member declaration with a value but no type expression, such as `a := 2;`, the type group of jobs would be omitted. In this case the dependency from the parent's target names resolved blocker would be to the member's evaluate job instead of to the type's evaluate job.

The *resolve names* jobs resolves all name references in the relevant expression and impose constraints from the dependencies to the next *typecheck* job. The value *typecheck* and type *evaluate* jobs generates a slot system and attempts to solve that system. Value expressions use this solution to populate types in its AST nodes. Type expressions use the returned type as the member's type. The target *resolve names* job attempts to determine what member its bind is targeting. If the target is found the bind's value *code generation* job is placed as a dependency for the target's *evaluate member* job.

The final task that has to be performed for a member is to resolve whether or not the member's value is unchangeably determined and constant. This is true if the member has a non-overridable bind whose values is not dependent on any non-constant members.

#### Use

Use expressions are challenging for both the name resolution system and for the data type resolution job system because of the dependencies placed upon them. A use expression introduce one or more names into the current scope. This name can either be fixed, in the case of statement 1 and 2 of the following listing. Statement 3 introduce the name of all members of the composite data type returned by `expr` into the current scope.

```
1 use test;
2 use expr as test;
3 use expr.*;
```

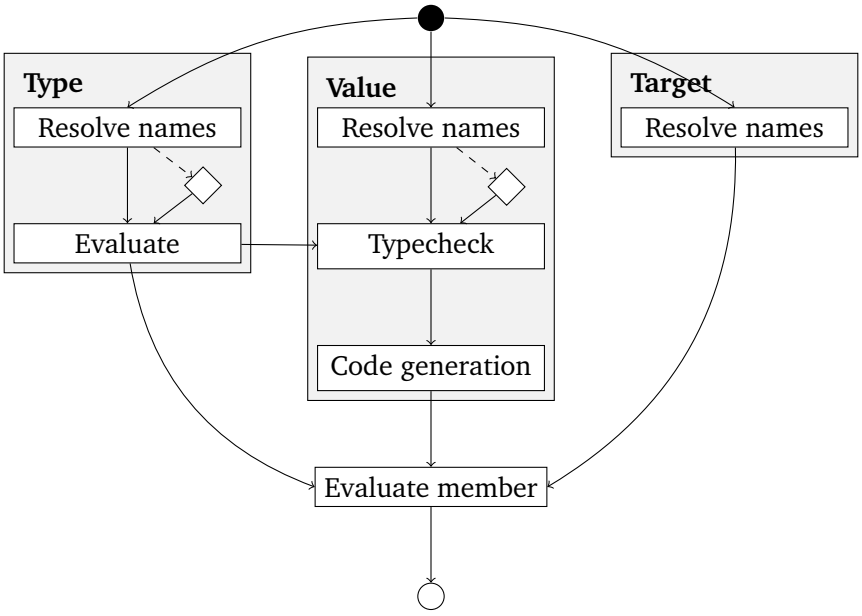



**Figure 5.1:** Dependency graph for jobs related to member declaration and assignment.

Use-all statements require type of the expression to be known so that the names are known. Because names from use-all statements take precedence over names captured from the parent as closures, the compiler does not know what names come from where before having resolved all use-all statements. Because of this, any expression that contains names that are not unambiguously determined by local members must be postponed until after all use-all statements have been resolved. The ambiguous names are the ones tagged as potential closures.

The dependency graph of use-all statements is show in figure 5.2. The expression jobs are the same as for the value of member binds. If the use expression has one or more ambiguous references a dependency must be placed from the parent's *Use resolved* job to the *Resolve names*. This would create a cycle. Because of this, use-all statements are required to have no ambiguous references in the current compiler implementation.

The other use expressions does not have to be dependencies of the *Use resolved* job, and therefore does not need to have the same restriction. Due to some quirks of the current implementation they are handled exactly the same as the use-all statements, and does receive the same restrictions.

### Type Class

Type classes provides an additional challenge for the composite data type system. Any type class is made up of two distinct statements: the declaration and the implementation statements.

```
1 # The declaraion statement.
```

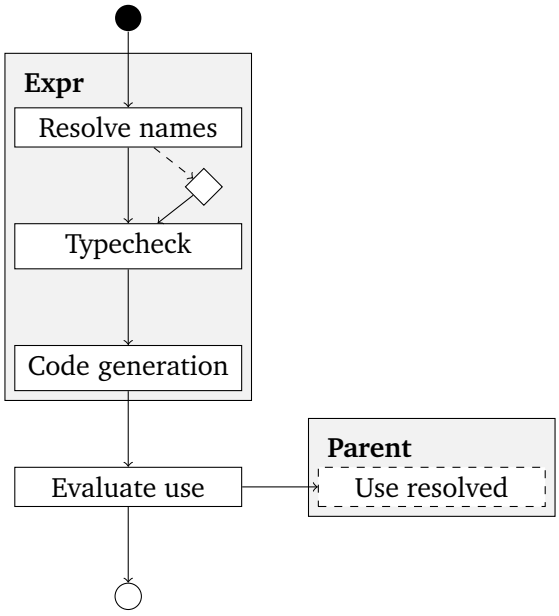



**Figure 5.2:** Dependency graph for use-all statements.

```
Eq := class [T: Type, U: type] {
    op==: (T, U) -> Bool;
};

# The implementation statements.
impl Eq[int, int] {
    op== := (...);
};
impl Eq[Bool, Bool] {
    op== := (...);
};
```

Because types can not travel up the module hierarchy we only have to consider any type class usage dependencies within the current module. This does however impose an interesting constraint: Any impl statement must be evaluated after the type class itself has been evaluated, but the type class can not be used until all of its impl statements have been evaluated.

Because all name spaces for a module are handled as part of the same batch in the composite data type system we are able to require all implementation statements to be evaluated before their respective type class is used.

Figure 5.3 shows a dependency graph for an implementation statement. The statement itself has three stages:

1. *Names* resolves the names of the target type class, the implementation parameters, and the body. This job is dependent on the parent composite's closures to be evaluated in order to determine what names can be determined through the closure. If any of the expressions in the implementation statement references any name that is not immediately found among the parent

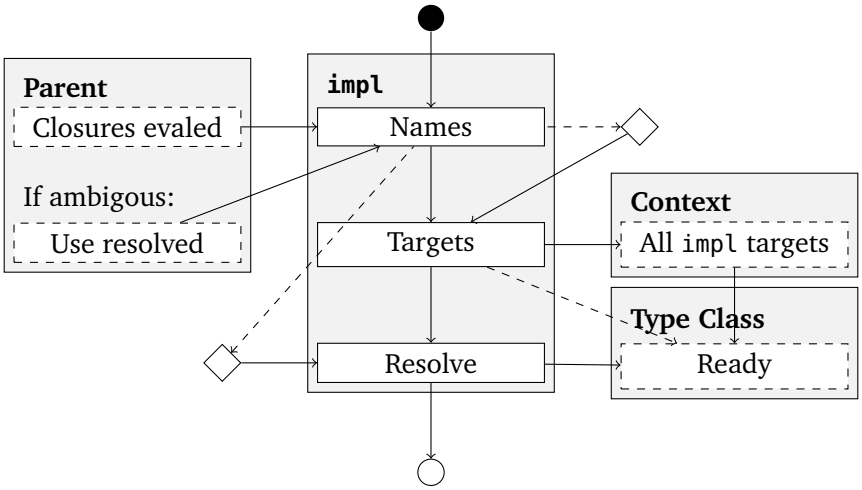



**Figure 5.3:** Dependency graph for jobs related to impl statements.

composite's members, a dependency is placed on the parent composite's *Use resolved* job.
A dependency is placed upon the *Targets* job from each member referenced from the target expression. A dependency is placed upon the *Resolve* job from each member referenced in the implementation parameters expressions or in the body.

2. *Targets* is evaluated after the type class target found during the *Names* job has been evaluated. This job is a dependency of the *All `impl` targets* job on the composite batch to block any type class from being used until we have discovered all implementation statements, to let them place dependencies from their *Resolve* jobs to their exact target type class' *Ready* job.
3. *Resolve* evaluates the parameter expressions and the body of the implementation statement, and registers this information with the type class. At this point, the implementation is ready, and after all other implementations for that type class has been resolved, the type class is ready for use.

### Init Expression

Init expressions are declared within the context of a composite data type. The expression itself is placed in a separate set of init expressions on the data type, while its usage is referenced by a special AST node. When the composite data type is instantiated the non-init expression expressions and binds are bound together with the init-expressions to create a monad that evaluates all the init expressions and then packs and returns the final object.

During the composite object resolution the init expressions are treated like normal expressions and pass through the same jobs.

### 5.2.2 Object Instantiations

Object instantiations are specialized object constructors that has additional information about how the members of the data type should be bound. This information is combined with the extra binds listed at the site of the instantiation to pack a final value. Due to the inter-dependencies between expressions and members, the expressions, binds and packs of members must be evaluated in a topological order. See section 3.2.4 for a description of the system and an algorithm for resolving the order of operations for a given instantiation.

### 5.2.3 Name Resolution

The name resolution system is responsible for determining what values are required for each expression.

The name resolution system operates in several stages: First, the name resolution system traverses the AST to determine what potentially free variables are part of each expression. These variables are stored as potential closures on the expression's root scope (function, template, or composite data type), and if the names are free in those scopes as well they propagate up the tree. An unknown name does not trigger an error at this stage because it might be discovered later as part of a `use` statement. This stage operates on the module's entire AST.

Next, during the composite data type solver, the real name references are populated. This is done expression by expression, for bind targets, bind values, and member types. Name resolution jobs for expression that contain free names are postponed until after all `use`-statements are resolved. This phase populates all lookup nodes with name references that point to specific values. When references to other members have been found, dependencies are placed on those members.

During type checking the AST is traversed again, this time to generate the type system that will be solved. Here the names and their types are propagated through the traversal. The value of any name that is known and constant is passed along as well. How these values propagate through the tree is described in section 3.2.5. The names, their types, and potential values are used in the same way during code generation.

### 5.2.4 Type Solver

The type solver is based of the Hindley-Millner type system[6][25], described in section 2.1.1.

In this language, mono types are represented as values of the type `Type`. This is a result of the design decision that all values are first-class citizens. Poly types on the other hand are represented as objects of type `cons`, or object constructors. These are special functions that, only at compile time, are able to instantiate certain objects. What is special about the constructor functions is that they are bijective, meaning the parameters can unambiguously be determined given a return value. This is called unpacking, where as calling the constructor function

using its parameters is called packing.

The type solver works in three stages: (1) The constraints are imposed based on the AST, (2) the constraint solver attempts to resolve all constraints and determine values, types, constructors, and instantiators, and (3) the relevant values are extracted from the solution and put back into the AST.

During the type solve each site in the AST that potentially can be inhabited by a value is slot. Each slot can have an explicit value attached to it. It can also derive certain information from relations to other slots. These relations are imposed by the constraints gathered during the first phase of the type solve.

The constraints that can be imposed are detailed in the following list. The **target** and **slot** parameters refer to other slots.

- **IS_OBJ(target, o:t)**: The target slot is expected to have the exact value `o` of type `t`.
- **IS_FUNC_TYPE(target)**: The **target** slot is expected to be a function type, $(t_1, \ldots, t_n) \to t_r$. Its parameter types and return type are derived from its indexed parameters, with the return type, $t_r$ at index 0 and parameter types $t_1 \ldots t_n$ at index $1 \ldots n$.
- **CONS(target, slot)**: The **target** slot value is expected to be the value of packing the target's parameter with the object constructor that is the value of the slot `slot`.
- **PARAM_NAMED(target, name, slot)**: the object constructor of **target** is expected to have a parameter called `name`, and its value should be the value of `slot`.
- **PARAM_INDEXED(target, index, slot)**: the object constructor of **target** is expected to have a parameter at index `index`, and its value should be the value of `slot`.
- **INST(target, slot)**: The value of the **target** slot should be given by an object instantiation by the instantiator as value from `slot`.
- **MEMBER_NAMED(target, name, slot)**: The object instantiator of **target** is expected to have a member named **name** with the value equal to the value of slot **slot**.
- **TYPE(target, slot)**: The type of the **target** slot is expected to be the value of `slot`.
- **EQUALS(target, slot)**: The **target** slot and **slot** are expected to be equal.

While regular type constructors are imposed using **IS_OBJ**, function type constructors have a special constraint, **IS_FUNC_TYPE**. This is because functions can have an arbitrary amount of parameters, which is something that normal cons objects does not support.

This slot-and-constraint system is merely an alternative representation of the AST that aims to provide a core semantics that is able to represent, in particular, the systems that are intended to be resolved at compile time. These include types, object constructors, and object instantiators. It is not capable of performing all run-time operations, such as calling functions.

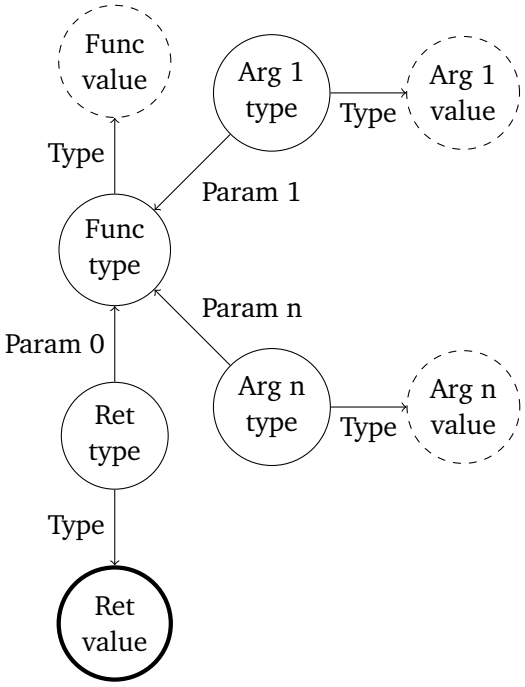



**Figure 5.4:** A graph of the slot system for a function declaration. Dashed slots are filled by the relevant child in the AST, and the bold slot is the one that is returned to the parent in the AST.

The slots and constraints constitutes a graph representation of the AST.

The **FUNC_DECL** AST node is translated into the slot sub-system shown in figure 5.4.

After the slot system is created the next step is to solve it. This is done through repeated application of a set of rules. These rules does for the most part not verify the constraint, but only attempt to propagate values. When no more rules can make changes to the system, the solve is complete.

Push Type

$$\frac{\text{value}(x) = m : t}{\text{value}(\text{type}(x)) = t : \textbf{Type}}$$

Pack Cons

$$\frac{\text{value}(\text{cons}(x)) = [t_1, \ldots, t_n] \to t \qquad \forall_{i \in 1..n} \text{value}(\text{param}(x, i)) = v_i : t_i}{\text{value}(x) = \text{cons}(x)[v_1, \ldots, v_n] : t}$$

Pack Inst

$$\frac{\text{value}(\text{inst}(x)) = (t_1, \ldots, t_n) \Rightarrow t \qquad \forall_{i \in 1..n} \text{value}(\text{mbr}(x, i)) = v_i : t_i}{\text{value}(x) = \text{inst}(x)\{v_1, \ldots, v_n\} : t}$$

Pack Function Type

$$\frac{\begin{array}{c}\text{cons}(x) = \textbf{functype} \\ \forall_{i \in 1..n} \text{value}(\text{param}(x, i)) = t_i : \textbf{Type} \\ \text{value}(\text{param}(x, 0)) = t_r : \textbf{Type}\end{array}}{\text{value}(x) = (t_1, \ldots, t_n) \to t_r : \textbf{Type}}$$

When the solve is complete, the solution is scanned for slots, $x$, where an object constructor, $c$, was given, either through **IS_OBJ**$(x, c)$ or as a member from **MEMBER_NAMED**$(p, m, x)$ where $m$ is a member of value$(p)$ and is the constructor $c$, but the solution requires another type. This is used as an indication that the object constructor should decay. Any such constraint that apply the constructor $c$ on slot $x$ is disabled, a new slot $t$ is allocated to represent the constructor, and the old constraint is replaced with **IS_OBJ**$(t, c)$ or **MEMBER_NAMED**$(p, m, t)$ respectively, and **CONS**$(x, t)$. Rules 5.1 and 5.2 present these transformations. $C$ is the set of constraints that were used to generate the current solution, and $C_1$ is the set of constraints for the next solve attempt. If any slot decayed and these transformations were applied, the current solution is discarded and the solve system is run again with these new constraints, $C_1$.

$$\frac{\textbf{IS\_OBJ}(x, c : \textbf{Cons}) \in C \quad \text{value}(x) = v : t \quad t \neq \textbf{Cons}}{\begin{array}{c}\{\textbf{IS\_OBJ}(y, c : \textbf{Cons}), \textbf{CONS}(x, y)\} \subseteq C_1 \\ \textbf{IS\_OBJ}(x, c : \textbf{Cons}) \notin C_1\end{array}} \tag{5.1}$$

$$\frac{\textbf{MEMBER\_NAMED}(p, m, x) \in C \quad \text{value}(x) = v : t \quad t \neq \textbf{Cons}}{\begin{array}{c}\{\textbf{MEMBER\_NAMED}(p, m, y), \textbf{CONS}(x, y)\} \subseteq C_1 \\ \textbf{MEMBER\_NAMED}(p, m, x) \notin C_1\end{array}} \tag{5.2}$$

After no more decay is required the next step is to validate the solution. This is done by letting each constraint check that it holds. Any validation error is reported, and the target slot of invalid constraints are tagged.

After the validation the results are sent back to the AST. Here the type of each AST node is populated, the object constructors and instantiators of **CONS** and **INST** nodes are set, and packed values are populated if available.

## 5.3 Code Generation and Optimisation

After the program semantics have been evaluated and validated we no longer need all the semantic information generated during the semantic analysis. The compiler now converts the AST into an intermediary byte code that describes the imperative steps required to evaluate each expression. This byte code representation still contains type information and is intended to be further compiled into some executable form. The intermediary byte code is not itself intended to be executed due to lacking access to absolute memory offsets and value sizes. Calculating such values at run time would be computationally expensive.

At this point the compiler will optimise the expressions. No optimizations have been implemented yet as they are considered out of scope for this thesis.

The current compiler implementation converts the intermediary byte code to a format that is more suitable for execution. This representation contains only sizes and absolute addresses for stack variables, as well as the most direct function pointers that are available.

## 5.4 Runtime System

The runtime starts during the pre-init, init, and post-init stages of the module life cycle (figure 5.5). After the module has been compiled into byte code it has the opportunity to perform any initialization operations using init monads and init expressions. The native module extension can hook in to perform any actions it needs to function, either before or after the init expressions using the pre- or post-init hook, respectively.

After all modules have been compiled and initialized the entire system is started. What happens here is up to the modules that have been loaded and have provided a start hook through their native module extensions. By default, if no module provides a start hook, the application will immediately exit.

All objects in the system is represented as a block of memory that has a fixed size given by its type. For objects that only are used during the execution of a function the object data is stored on the function stack. Any object that is required through the life time of the module is stored on the module's object store. This is an arena that objects are pushed on to permanently, and only when the module gets destroyed is the arena is freed. No memory should be pushed onto this arena during run time to avoid memory leak issues. In cases where objects must have a more specialized life time, modules can implement their own mechanisms to manage this.

While many types have a fixed size, such as integers, other types can have varying size. Examples of such types include monads, strings, and lists. These objects can utilize a heap that is passed through all function calls to store the additional data that is potentially varying in size, in addition to the fixed-size component that is stored on the stack and points to the heap.

## 5.5 Modules

Each module compiles into a composite data type, and if that data type has init expressions they will be executed as part of module initialization.

This shared object will be read before the rest of the module is compiled, and provides several hooks for altering or augmenting how and what the module loads. The life cycle is presented in figure 5.5.

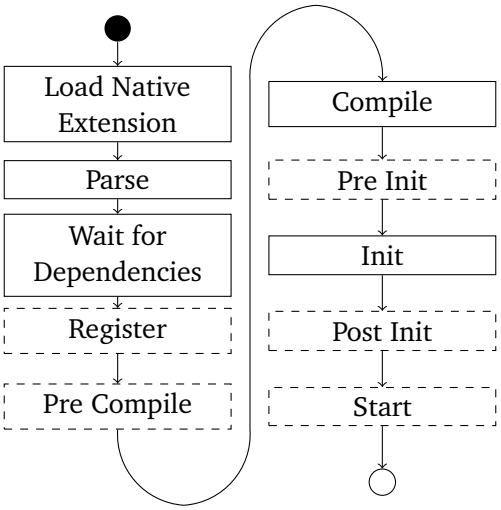



**Figure 5.5:** A flow chart describing the stages of a module. The dashed nodes are hooks for the native module extension.

### 5.5.1 Native Module Extension

The native module extensions provide a mechanism for performing advanced tasks on the system through a shared library. On Linux the extensions are shared objects. All the default extensions are written in C.

All such extensions must use the `STAGE_MODULE(modname, load_func)` macro to register themselves. This macro creates a special routine, `struct stg_module_magic *stg_module_magic()`, that provides the runtime system with basic information about the module, such as the version of the project it was compiled against, and what function should be the entry point. After the module is loaded and the runtime system has received the `stg_module_magic` struct, the module's `load_func` routine is called. This routine is responsible for registering any hooks this extension requires.

A native module extension can both directly modify the AST of the module and provide objects and functions for the run time system to pick up with the native function declaration- and native literal nodes. There are several locations in the module's life cycle where the native module extension can hook into in order to provide functionality. The life cycle is presented in figure 5.5. Hooks are shown with dashed outlines.

# Chapter 6

# Results

This chapter discuss the language design, SQL-interface, and prototype implementation.

## 6.1 Language

The style of the language has changed over the course of the project's life. Originally the system was designed to allow users to describe real-time signal processing systems, such as DSPs. This design was declarative, allowing the user to connect processing devices together to form signal chains. Other functionality that was considered for the system did not fit with this model, so more general paradigms were considered.

Another consideration was whether or not the language should be purely functional. Originally the language allowed for functions with side effects. These were intended to perform initialization, similar to the init-expressions in the current design. To avoid the issues that can arise from allowing functions with side effects they were dropped. This was in particular because of the design decision of not imposing an order on statements. Monads was chosen as the alternative. This choice was made due to the success of monads in the Haskell community[13]. It was also inspired by other functional systems, such as Cats for Scala. The monad structure also lent itself well to describing the signal systems the language was originally designed for.

Unlike Scala this language is pure functional. This means that the programmer is unable to write code that can have side effects. In Scala this is only a convention the developer can impose themselves.

The type class and monad systems in all three languages are very similar, as both Scala and this language are directly inspired by Haskell[13].

### 6.1.1 Init Expressions

Init expressions aim to make specifying programs of disparate parts easier by automating the work of ordering the parts based on their inter dependencies. It does

however pose a challenge for developers using the language. They can not know the exact order the operations occur because the compiler does not provide such a guarantee. For new developers in particular this feature might lead to unexpected behaviour. For example, if the developer expects the two write-to-file operations in the following listing to come in order, they might be surprised when they do not. The developer can in fact not even rely on `fclose(f)` to happen after the prints, despite being the last operation in the program text. The only guarantee in this listing is that the two `printf` operations and the `fclose` operation will occur after `fopen`, as the previously mentioned operations depends on the result of `fopen`.

```
f := !withIO(fopen("test", "rw"));
!withIO(fprint(f, "Hello, "));
!withIO(fprint(f, "World!"));
!withIO(fclose(f));
```

To serialize such operations one must manually bind the monad operations, either using the >>= operator or do notation.

```
!do {
    f <- withIO(fopen("test", "rw"));
    withIO(fprint(f, "Hello, "));
    withIO(fprint(f, "World!"));
    withIO(fclose(f));
}
```

One could consider restricting this mechanism to only work on order-independent or order-forcing operations, in particular by disallowing executing IO within its context with `withIO`. This does seem too restrictive as performing IO, such as loading a file or connecting to a database, can be useful tasks to perform during program loading and can be performed in an order-independent manner.

### 6.1.2 Error Handling

This proposal has not considered how errors in operation that can fail should be handled. The current prototype implementation of the SQL-interface only prints any error message and aborts immediately. This is not a satisfactory solution as errors can not be recovered from, making any application developed in the language unreliable.

Haskell handles this by letting monads throw exceptions[13]. When an exception is thrown the rest of the bound operations in the monad chain are ignored. If the monad chain is enclosed within a catch-monad, the error can be handled by a provided function and normal operation can resume.

One alternative is to have IO operations that can fail return a `Maybe`-like object. If the operation succeed, `some` result is returned. If the operation fails `none` is returned instead. This is the approach taken by Rust[22]. Routines in rust that can fail return a Result object. This object, much like Maybe, either returns OK with the returned value, or Error with an error description.

The Rust approach involves more obvious error handling by requiring checking that the result is OK before unpacking the result. This system also requires

some way of exiting a monad early to avoid forcing it to run to completion. The exception semantic of the Haskell system can lead to less obvious failure paths because the monad chain can abort without explicit syntax.

### 6.1.3 Type System

The strict typing has been a part of the language's design since its conception due to the goal of making writing programs reliable and have good error messages.

The current type system implementation functions for the most part, but systems such as constructor decay are not cleanly integrated into the solver, but rather tacked on. The type solver is limited in its ability to reason about object constructors. In particular type classes can pose a challenge for the system. This is due to the slot system only allowing for at most one value for each parameter at each slot. It is not capable of considering multiple options of values. This is an area that should be investigated further to improve the power and usability of the type system.

#### Function Generalization

When the current compiler is unable to unambiguously determine the types of a function's parameters or return type it fails. Unlike the Hindley-Milner type system it does not attempt to create a generalized template function. This means that in order to create a template function the application developer is required to explicitly tag the template parameters as such using the `$` prefix.

#### Template Qualifiers

Any template function or data type that passes name checking and has no immediately conflicting types is admitted as valid. Any type conflicts that happen as a result of template parameters is not caught until the attempt to instantiate the template and type check the resulting AST. Such an error occur at the site where the parameter is used. To provide better error reporting, the type system could instead require that the function or type should be instantiable for all types it is declared to be valid for. To be able to use constructors that are not declared for all types, the template's parameters must be sufficiently constrained to only allow values that overlap with one of the constructor's implementations.

As an example, in Haskell, the function >> could be defined as follows.

```
1 (>>) :: Monad m => m a -> m b -> m b
```

The expression `Monad m` before `=>` is a template qualifier. This indicates the function is defined for any type constructor `m` which is an instance of the `Monad` type class.

## 6.2 SQL-Interface

The inspiration from the Doobie library from scala is very clear in this language's SQL interface. From the string interpolation for specifying queries to the monadic structure of the queries.

```
1  mod sql;
2
3  getByID := (id: int)
4      => sql"select account, name from account where account=${id}";
5
6  !withIO(do {
7      db <- sql.postgres.connect("dbname=test");
8
9      res <- query(db, getByID(5));
10     printResult(res);
11 });
```

While the proposed SQL-interface provides some type checking, it still is unable to determine if a specified query is valid or not until the query is executed. This will then trigger a run-time error. It would be useful to, at compile time, determine that all queries are valid. This is difficult to achieve because whether or not a query is valid is dependent on the database the query is executed on. The validity can also change throughout the life time of the application due to changes to the database schema, such as added, altered, or removed tables or columns.

The mechanism for returning tuples and objects as rows from queries introduce some challenges for the language design. Tuples was introduced because they allow for returning multiple unnamed fields from the queries. An alternative could be to automatically fill a composite object directly, either based on their names or by the index. This was not chosen because constructing composite data types from a dictionary or list of values of different types would be very difficult to implement. It seemed more clean to first convert the rows into tuples, then construct the objects from the tuples.

The paradigm the database interface ended up with was an SQL-interface. It ended up with a version that provides better user ergonomics for the interaction between SQL and the language, but does not replace SQL. Other options such as ORMs and list comprehension schemes were also considered. The SQL-based interface was chosen for several reasons. As SQL is the primary way of interacting with modern relation DBMS it is the most direct and most powerful the interface can be. It can also support any database engine by only implementing the connection and query back end. ORMs requires an SQL translation system that must be tweaked to conform to each back end's quirks.

The string interpolation functionality was heavily influenced by it's use as part of the SQL interface. Because this system must keep the original value of the expressions until they are cast to the type passed to the database, they required support for custom data structures to store the format parts.

The current design does not discuss any way of preparing statements. Prepared statements incur less overhead in case of repeated queries with different

parameters[26]. The reduced overhead is due to the back end being able to cache the query strategy used instead of having to transmit and process the SQL query text for each query. There are two mechanisms that can be implemented to provide prepared statements. The SQL-interface can implement an explicit monad that prepares an SQL-statement and another monad that takes a prepared statement and parameters and executes the query. Such a system is incompatible with the current string interpolation system because the string interpolator captures the values of the parameters and stores them together with the query. It is thus not possible to pass different parameters to the same query string. An alternative is to use parameter indices in the query text and pass the parameters separately. This solution should require the developer to explicitly state the parameter types with the query string to enable type checking. The types can not automatically be determined as the parameters are not embedded in the query string through the string interpolator. One could also consider an automated system for preparing statements. The first time a query is executed its query string is prepared. The prepared query is then used for all subsequent executions of the that query. Such a system requires some mechanism to keep track of the prepared query strings.

It was also considered to implement a language integrated query system, similar to LINQ, in place of the string interpolator. This was dropped as it would require extensive modifications to the parse system of the language, and the result would not be very different in usability from the SQL-interface with string interpolation proposed in this thesis. One advantage an embedded domain specific language or a list comprehension system would afford is better type checking and better user feedback in an integrated development environment.

By implementing the SQL interface we lay the foundation upon which other interfaces can be built.

### 6.2.1 State of Implementation

The database connection API (section 4.1) and the simple query interface (section 4.2) have been implemented, and are available under `./modules/sql/` in the project repository. This module currently only support the PostgreSQL database back end. In the future this library should be expanded to include more database systems or provide support for plug-in database adaptors.

The proposed interface does not impose any constraints on what database back end it interfaces with, other than it must support SQL. Some functionality discussed could be lost with certain DBMS that does not support or impose as strict type requirements. Both file-based and server-based database systems should work well.

## 6.3 Compiler Implementation

At the time of writing, the current compiler implementation has most of the discussed functionality implemented.

The compiler is, however, not production-ready. Extensive testing should be performed to ensure the stability and correctness of the system.

### 6.3.1 Recursion of Polymorphic Function

While function recursion is supported for regular functions, polymorphic functions does not have this ability. This is due to a shortcoming in the system for recursion and name resolution. In a named expression, for example a bind statement, the name of the target is tagged as a self reference and it references itself both in the type solver and during code generation. Recursive functions, when generated, create a closure on itself which is a reference to itself after the rest of the function is generated. Template expressions does, however, not have this ability.

This leads to certain higher order functions, such as map, having to be declared as native functions. In the case of map this implementation might be preferable to a native implementation because map in the current implementation returns a special list that contains the map function and points to the original, unmodified list. Only when an element of the list is required is that element passed through the map function. This functionality is achieved "for free" in Haskell by virtue of it being a lazy language. The next step in the recursion is not evaluated until it is needed.

The simplest implementation of a polymorphic recursive function would be monomorphic within its body and polymorphic everywhere else[4]. This would be sufficient, at least for the higher order functions we have discussed in this thesis (map and foldl). A more permissive solution has been discussed, but imposes new issues[4].

### 6.3.2 Recursive Data Types

Recursive data types allows for linked structures, such as linked lists and trees. They are created by having an variant have one or more option with data referencing itself, and at least one option that does not. This is to allow the data type terminate.

This is challenging because of the need to know the size of every object. To implement a recursive data type, the self reference must be a reference, otherwise the data type would have infinite size.

The lack of this feature requires the List type be implemented as a primitive in the base module, and makes creating new list- or tree-like structures very difficult.

### 6.3.3 Tuple

Tuple types, while being considered early in the language's design, was one of the last features to be considered seriously. Tuple type declarations are varadic by nature. This is a challenge for the compiler due to the design decision that functions and constructors must have a fixed number of parameters. Several designs for the implementation of tuples have been considered.

- The tuple type could get an exception in the type solving system. This is how function type declaration is implemented. Such a solution is not optimal because it introduces more complexity and more cases that must be handled in the compiler.
- The language could permit varadic functions. This, again, requires substantial complexity to be added to the compiler. Such functions could also pose challenges when passed as parameters to other functions expecting a fixed number of parameters.
- We could implement tuple types in terms of a chain of tuples, where each tuple has exactly two members. Under this declaration, a tuple type such as `Tuple(int, String, Bool)` would be syntactic sugar for the recursively defined two-tuple `Tuple(int, Tuple(String, Bool))`.

### 6.3.4 Type Classes

Some type class implementations, such as the monads, are defined over all types. This requires the implementation itself to be polymorphic. Notice in the following example how the implementation for the IO monad still has a parametric type `$T`.

```
Monad := class[$M[$T]] { (...) };
impl Monad[IO[$T]] { (...) };
```

This feature is currently not implemented into the compiler.

The monad example highlights another feature that has yet to be implemented: polymorphic type class members. Some members of the type class can have additional template parameters beyond what is defined for the type class itself. In this following example, the `bind` function has the parameter `$U` which is not part of the type class.

```
Monad := class[$M[$T]] {
    bind: (a: M[T], fn: T -> M[$U]) -> M[U];
    (...)
};

impl Monad[IO[$T]] {
    bind := (a: IO[T], fn: T -> IO[$U]) -> IO[U]
        => (...);
    (...)
};
```

Another limitation is that types defined as part of the type class body can not be used by other members of the type class. For example, referencing the member `Out` as part of the type declaration for the member `fn` in the following example is not allowed. This functionality is useful for having functions return different types for different instances, such as for different string interpolators.

```
Test := class[$T] {
    Out: Type;

    fn: (T) -> Out;
};
```

### 6.3.5 Use Expressions

Currently the dependencies imposed by use statements when resolving data types are not granular enough, resulting in false cyclic dependency errors. This is due to the requirement that all use-expressions must be known before the names of expressions containing ambiguous names can be resolved. The use-all statements require this because the names they expose are dependent on the type of the use expression. This system should be further improved to admit more complex systems of use expressions.

### 6.3.6 Error Reporting

Clear error reporting is important to make the programmer understand what has gone wrong, and to make the process of figuring out how to fix the issues as easy as possible.

While many parts of the system provides proper error messages, the type solver in particular some times provides confusing errors. The type solver keeps the value for each slot that comes from the most authoritative constraint. How authoritative a constraint is is given by what imposed the constraint. When errors are to be reported, the most authoritative value is the one that is marked as expected. Without proper tuning of this score the resulting error messages will not be very useful.

For a future iteration of the compiler the authority scores should be properly adjusted. Alternatively other systems for reporting typing errors should be investigated that more effortlessly produce useful error messages.

# Chapter 7

# Conclusion

This thesis has presented the design of the new functional programming language *stage* and an accompanying SQL-interface. A prototype of the compiler for this language and the database access module have been implemented. This report has discussed the systems that compose the compiler.

In particular the design decision to allow statements to appear in any order in the program text and the decision to make all values, including types, first-class citizens had major impact on the systems for semantic analysis. The composite data type system is designed to evaluate all expressions of a program in such an order that the expression's dependencies were evaluated before the expression. The type solver is designed to be flexible enough to reason about values of any type.

The design of the SQL-interface is intended to make specifying safe programs easy and to make it difficult to make mistakes. Being safe implies both type safety and protection from security-related vulnerabilities. This principle lead to the `Query` object and query string interpolator which automatically converts and escapes parameters. Also the system for converting returned tuples to native tuples or data structures is inspired by this principle. The act of querying the database is designed to follow the convention from the rest of the language. By utilizing the `IO`-monad, programs containing database queries can be described in a pure manner.

The prototype implementation of the compiler has many of the features discussed in this thesis, but some features were left as future work. The SQL-module currently only implements a simplified query interface without the desired type-safety and usage ergonomics.

The goals for this thesis is:

- Design database access as an integrated part of a new functional programming language.
- Design an API that is a natural part of the language.
- Find a database solution that may easily be supported as a part of the functional language.
- Evaluate if a server based database (e.g. PostgreSQL) or an embedded data-

base (e.g. SQLite) is the best solution.
- Prototype a database solution for this programming language.

In terms of the goals, the new functional programming language has been designed and an SQL-interface has been integrated into it. The interface is integrated by using specialized mechanisms to allow developers to ergonomically write SQL-statements containing parameters from their program. This interface also provide type checking and error reporting for detecting issues of passing parameters of the wrong type. The SQL-interface use the monad system for performing all operations that has side effects, such as connecting to the database or querying. The monad mechanism allow a natural way of interfacing with a database in a functional language. The proposed interface only require the target database to query using SQL. Thus both server based databases, such as PostgreSQL, and embedded databases, such as SQLite, work well with this system. A prototype of the compiler and a simplified version of the SQL-interface have been implemented and are available on GitHub[1].

## 7.1 Future Work

The system for error handling in monad chains was not designed. Two solutions were considered:

1. Letting monads throw exceptions that aborts the current monad chain and can be catched by the caller.
2. Returning an object that must be unpacked to determine if the call succeeded.

A future iteration of the monad-system should evaluate which error handling mechanism is the best fit for the language.

Not all features of the compiler were completed for the prototype implementation. The implementation of these features should be investigated in future iterations. In particular tuples, parametric recursive functions, and recursive data types should be implemented. Classes also lack some functionality. Future iterations of the compiler should support classes with parametrically polymorphic implementations, parametrically polymorphic members, and inter-references between members.

The current type system solver should be improved in future iterations of the compiler. In particular, improving the system's ability to reason about type classes and the system's ability to report useful error messages should be investigated.

In some circumstances the composite data type solver currently fail to solve certain systems of composite data types due to too strict dependencies. Both classes and use-statements impose a complex and far-reaching order requirement in the system. In particular in batches of module composites, this requirement can lead to unexpected cyclic dependencies. Reducing the reach of the order require-

[1] `https://github.com/oddkk/stage`

ments imposed by, in particular, classes and use-statements would alleviate this issue. Such solutions should be investigated for future iterations of the compiler.

No effort was put into optimizing the compiler itself, the compiled programs, nor any of the libraries. All three areas should be addressed in the future.

Neither the current SQL-library design nor implementation provide optimizations such as preparing repeatedly used statements. Implementing either an explicit or implicit mechanism for preparing queries could lead to less overhead when using a query multiple times.

Besides fully implementing the proposed SQL-interface, one could look into different paradigms for such an interface. One option that can be investigated is to implement some language integrated query system. Such a system can be based around a list comprehension system[21] or an embedded domain specific language like LINQ[19].

# Bibliography


[1] A. Sabry, «What is a purely functional language?», *Journal of Functional Programming*, vol. 8, no. 1, pp. 1–22, Jan. 1998, ISSN: 0956-7968, 1469-7653. DOI: 10.1017/S0956796897002943. [Online]. Available: https://www.cambridge.org/core/product/identifier/S0956796897002943/type/journal_article (visited on 21/05/2020).

[2] S. L. Peyton Jones and P. Wadler, «Imperative functional programming», in *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '93, Charleston, South Carolina, USA: Association for Computing Machinery, 1st Mar. 1993, pp. 71–84, ISBN: 978-0-89791-560-1. DOI: 10.1145/158511.158524. [Online]. Available: https://doi.org/10.1145/158511.158524 (visited on 25/05/2020).

[3] C. Strachey, «Fundamental concepts in programming languages», *Higher-Order and Symbolic Computation*, vol. 13, no. 1, pp. 11–49, 1st Apr. 2000, ISSN: 1573-0557. DOI: 10.1023/A:1010000313106. [Online]. Available: https://doi.org/10.1023/A:1010000313106 (visited on 25/05/2020).

[4] B. C. Pierce, *Types and programming languages*. MIT press, 2002.

[5] R. Milner, «A theory of type polymorphism in programming», *Journal of Computer and System Sciences*, vol. 17, no. 3, pp. 348–375, Dec. 1978, ISSN: 00220000. DOI: 10.1016/0022-0000(78)90014-4. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/0022000078900144 (visited on 12/12/2019).

[6] R. Hindley, «The principal type-scheme of an object in combinatory logic», *Transactions of the American Mathematical Society*, vol. 146, p. 29, Dec. 1969, ISSN: 00029947. DOI: 10.2307/1995158. [Online]. Available: https://www.jstor.org/stable/1995158?origin=crossref (visited on 12/12/2019).

[7] P. Wadler, «Comprehending monads», in *Proceedings of the 1990 ACM conference on LISP and functional programming*, ser. LFP '90, Nice, France: Association for Computing Machinery, 1st May 1990, pp. 61–78, ISBN: 978-0-89791-368-3. DOI: 10.1145/91556.91592. [Online]. Available: https://doi.org/10.1145/91556.91592 (visited on 01/04/2020).

[8] E. Moggi, *An abstract view of programming languages*. University of Edinburgh, Department of Computer Science, Laboratory for ..., 1989.

[9] P. Wadler, «The essence of functional programming», in *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '92, Albuquerque, New Mexico, USA: Association for Computing Machinery, 1st Feb. 1992, pp. 1–14, ISBN: 978-0-89791-453-6. DOI: 10.1145/143165.143169. [Online]. Available: https://doi.org/10.1145/143165.143169 (visited on 25/05/2020).

[10] S. P. Jones, «Tackling the awkward squad: Monadic input/output, concurrency, exceptions, and foreign-language calls in haskell», 2001.

[11] C. E. contributors. (). «Cats Effect: Home», Cats Effect. Library Catalog: typelevel.org, [Online]. Available: https://typelevel.org/cats-effect/ (visited on 20/04/2020).

[12] (). «System.IO», [Online]. Available: https://hackage.haskell.org/package/base-4.12.0.0/docs/System-IO.html (visited on 20/04/2020).

[13] P. Hudak, J. Hughes, S. P. Jones and P. Wadler, «A history of haskell: Being lazy with class», in *Proceedings of the third ACM SIGPLAN conference on History of programming languages - HOPL III*, San Diego, California: ACM Press, 2007, pp. 12–1–12–55. DOI: 10.1145/1238844.1238856. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1238844.1238856 (visited on 01/05/2020).

[14] P. Wadler and S. Blott, «How to make ad-hoc polymorphism less ad hoc», in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '89*, Austin, Texas, United States: ACM Press, 1989, pp. 60–76, ISBN: 978-0-89791-294-5. DOI: 10.1145/75277.75283. [Online]. Available: http://portal.acm.org/citation.cfm?doid=75277.75283 (visited on 15/05/2020).

[15] (). «PostgreSQL: Documentation: 12: Chapter 33. libpq - C Library», [Online]. Available: https://www.postgresql.org/docs/12/libpq.html (visited on 22/04/2020).

[16] (). «Scala Book», Scala Documentation. Library Catalog: docs.scala-lang.org, [Online]. Available: https://docs.scala-lang.org/overviews/scala-book/scala-features.html (visited on 24/04/2020).

[17] R. Norris. (). «doobie», doobie. Library Catalog: tpolecat.github.io, [Online]. Available: https://tpolecat.github.io/doobie/ (visited on 22/04/2020).

[18] C. contributors. (). «Cats: Home», Cats. Library Catalog: typelevel.org, [Online]. Available: http://typelevel.org/cats/ (visited on 22/04/2020).

[19] B. Wagner. (). «Language-integrated query (LINQ) (c#)», [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/ (visited on 11/02/2020).

[20] E. Meijer, B. Beckman and G. Bierman, «LINQ: reconciling object, relations and XML in the .NET framework», in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '06, Chicago, IL, USA: Association for Computing Machinery, 27th Jun. 2006, p. 706, ISBN: 978-1-59593-434-5. DOI: 10.1145/1142473.1142552. [Online]. Available: https://doi.org/10.1145/1142473.1142552 (visited on 09/02/2020).

[21] E. Cooper, «The script-writer's dream: How to write great SQL in your own language, and be sure it will succeed», in *Database Programming Languages*, P. Gardner and F. Geerts, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2009, pp. 36–51, ISBN: 978-3-642-03793-1. DOI: 10.1007/978-3-642-03793-1_3.

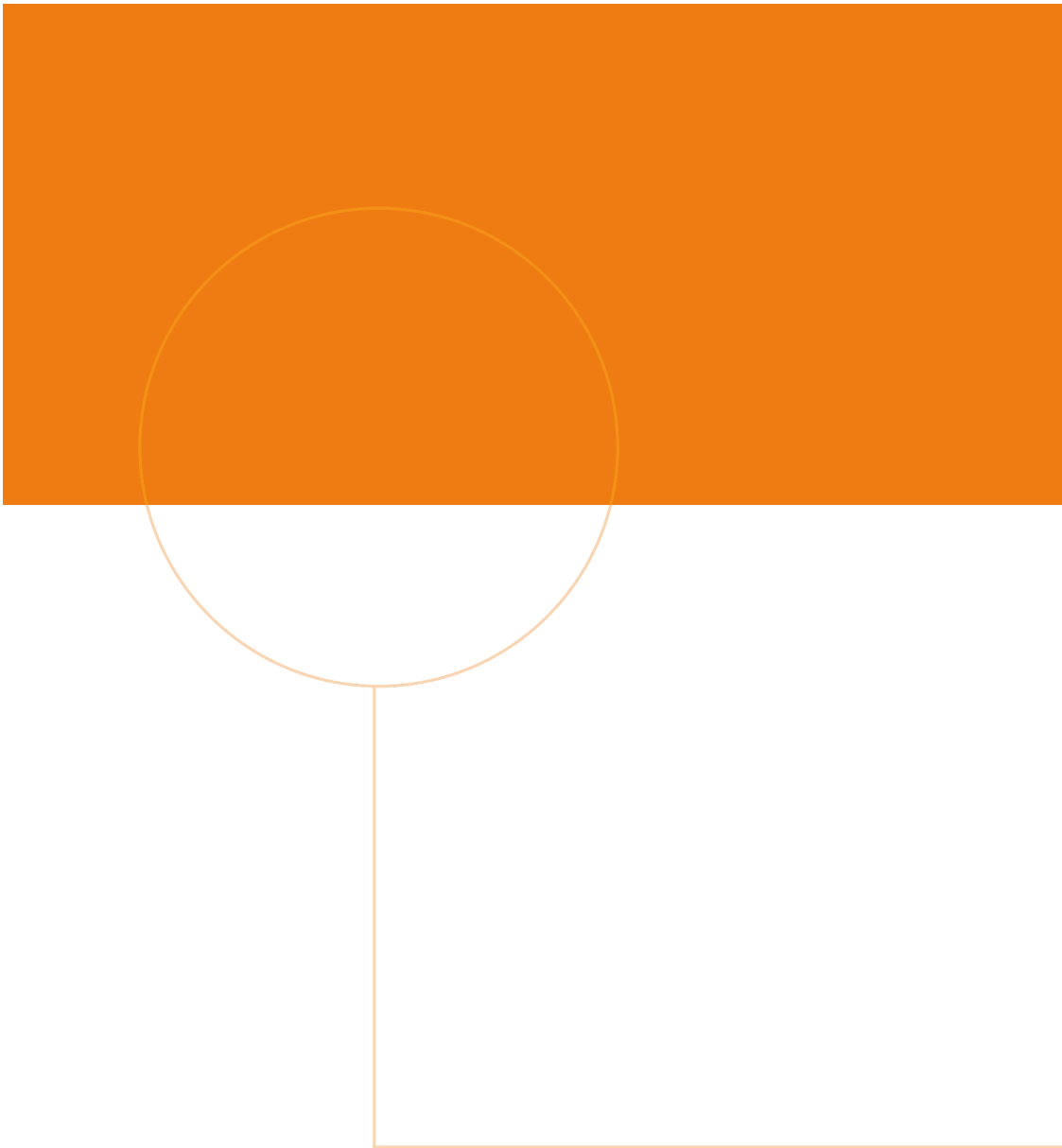
[22] (). «The Rust Programming Language - The Rust Programming Language», [Online]. Available: https://doc.rust-lang.org/book/title-page.html (visited on 26/04/2020).

[23] (). «Bison - GNU Project - Free Software Foundation», [Online]. Available: https://www.gnu.org/software/bison/ (visited on 19/04/2020).

[24] (). «re2c — re2c 1.2 documentation», [Online]. Available: https://re2c.org/ (visited on 19/04/2020).

[25] R. Milner, «A proposal for standard ML», in *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, ser. LFP '84, Austin, Texas, USA: Association for Computing Machinery, 6th Aug. 1984, pp. 184–197, ISBN: 978-0-89791-142-9. DOI: 10.1145/800055.802035. [Online]. Available: https://doi.org/10.1145/800055.802035 (visited on 09/02/2020).

[26] (). «PostgreSQL: Documentation: 12: PREPARE», [Online]. Available: https://www.postgresql.org/docs/12/sql-prepare.html (visited on 24/05/2020).



NTNU

Kunnskap for en bedre verden