Technical University of Košice

Faculty of Electrical Engineering and Informatics

# From Proofs of Formal Propositions to Executable Implementations

Master's Thesis

2016 Bc. František Silváši

Technical University of Košice

Faculty of Electrical Engineering and Informatics

# From Proofs of Formal Propositions to Executable Implementations

**Master's Thesis**

| | |
|---|---|
| Study Programme: | Informatics |
| Field of study: | Informatics |
| Department: | Department of Computers and Informatics (KPI) |
| Supervisor: | doc. Ing. Martin Tomášek, PhD. |
| Consultant(s): | |

**Košice 2016** **Bc. František Silváši**

## Abstract

Programs are often full of underutilized semantic information, often hidden in types. It is not uncommon that an implementation of a function simply mimics its type albeit with different syntax. An approach to code generation from types is presented. A signature is looked at as a preposition (courtesy of the Curry-Howard isomorphism) and a proof is then synthesized within our calculus created for this purpose using heuristic methods. The conducted analysis generates a set of strings which are then transformed into a target language, in our case Haskell. This approach has been used to generate the core of Haskell's standard Prelude library and can in general serve as an interesting way of program synthesis and automated construction.

**Keywords**

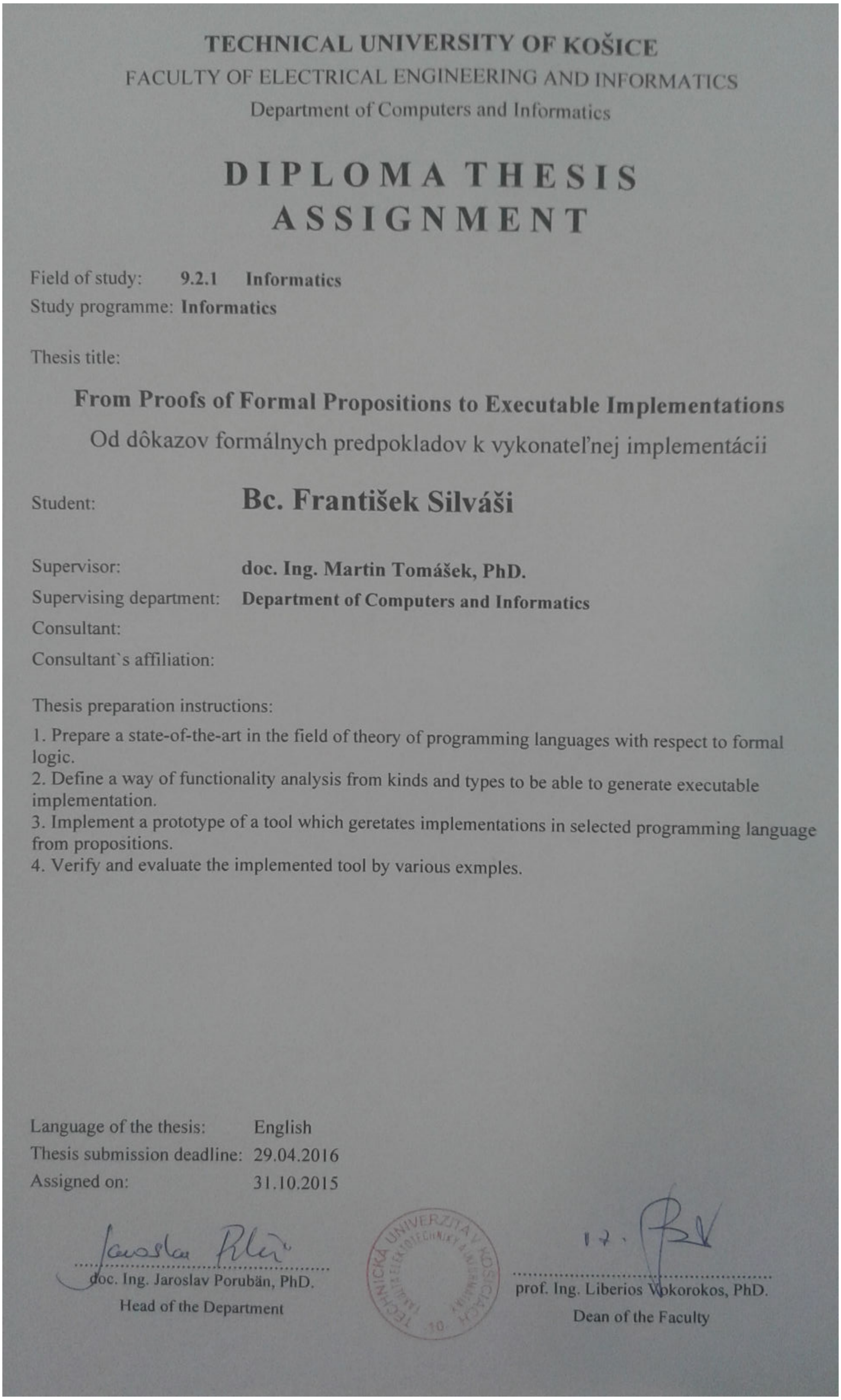
Code generation, formal logic, formal verification, Haskell

## Abstrakt

Programy sú často plné nedostatočne využitých sémantických informácii, často ukrytých v typoch. Nestáva sa zriedka, že implementácia funkcie je len prepis jej signatúry inou syntaxou. Prezentujeme prístup ku generovaniu kódu z typov, na ktoré sa pozeráme ako na logické tvrdenia (dôsledok korešpondencie Curryho a Howarda), ktorých dôkaz syntetizujeme v našom kalkule pomocou heuristických metód. Kalkul bol navrhnutý práve pre teno účel. Výsledkom analýzy je množina reťazcov, ktoré sú následne transformované na cieľový jazyk, v našom prípade Haskell. Tento spôsob je použitý na vygenerovanie časti štandardnej knižnice Haskellu - Prelude. Vo všeobecnosti takýto postup predstavuje zaujímavý pohľad na syntézu programov.

**Kľúčové slová**

Generovanie kódu, formálna logika, formálna verifikácia, Haskell


TECHNICAL UNIVERSITY OF KOŠICE

FACULTY OF ELECTRICAL ENGINEERING AND INFORMATICS

Department of Computers and Informatics

# DIPLOMA THESIS ASSIGNMENT

| | |
|---|---|
| Field of study: | **9.2.1 Informatics** |
| Study programme: | **Informatics** |

Thesis title:

**From Proofs of Formal Propositions to Executable Implementations**

Od dôkazov formálnych predpokladov k vykonateľnej implementácii

| | |
|---|---|
| Student: | **Bc. František Silváši** |
| Supervisor: | **doc. Ing. Martin Tomášek, PhD.** |
| Supervising department: | **Department of Computers and Informatics** |
| Consultant: | |
| Consultant`s affiliation: | |

Thesis preparation instructions:

1. Prepare a state-of-the-art in the field of theory of programming languages with respect to formal logic.
2. Define a way of functionality analysis from kinds and types to be able to generate executable implementation.
3. Implement a prototype of a tool which geretates implementations in selected programming language from propositions.
4. Verify and evaluate the implemented tool by various exmples.

| | |
|---|---|
| Language of the thesis: | English |
| Thesis submission deadline: | 29.04.2016 |
| Assigned on: | 31.10.2015 |

[illegible]

doc. Ing. Jaroslav Porubän, PhD.
Head of the Department

TECHNICKÁ UNIVERZITA V KOŠICIACH FAKULTA ELEKTROTECHNIKY A INFORMATIKY 10

17. [illegible]

prof. Ing. Liberios Vokorokos, PhD.
Dean of the Faculty

**Declaration**

I hereby declare that this thesis is my own work and effort. Where other sources of information have been used, they have been acknowledged.

Košice, Apríl 5, 2016 . . . . . . . . . . . . . . . . . . . . . . . . . . . .

*Signature*

**Acknowledgement**

I would like to express my sincere gratitude to my supervisor doc. Ing. Martin Tomášek, PhD. His suggestions and guidance were of paramount importance.

# Preface

The thesis has emerged from a simple observation of the relationship between function signatures and their implementations. It is in fact often the case that as long as a function is pure, its type generally contains enough information to make its implementation superfluous. The more expressive the type system in question, the more invariants about functionality it can encode; up to the point where it restricts implementations that typecheck to exactly one. Types themselves then become specifications / models that can be mechanically transcribed into executable code.

To take a look at the problem from a different point of view, we could consider types as prepositions and corresponding programs as their proofs. This relationship is known as Curry-Howard isomorphism. This rephrases the question in a way such that we are looking to automatically construct a proof of a proposition, which then corresponds to the program (implementation) that we do care about.

We present a calculus (and its implementation) that can be used to construct function implementations from their type signatures. It is a combination of formally defined rules that manipulate prepositions (not unlike conducting a proof with natural deduction) and a set of heuristics that automatically invoke said rules, attempting to construct a proof. The transformation of result of the analysis to an actual implementation in Haskell is then relatively straightforward.

Of course, it is also possible to do further transformations into more ubiquitous imperative languages. We take a brief look at an unusual way of doing so; modifying C++ in such way so that its syntax resembles a functional language, thus making it an easier target for code generation. Effectively, we create a toy quasi-functional language directly within C++.

# Contents

# List of Figures

# List of Tables

# 1 Introduction

Tremendous amount of effort is being put into discovering and inventing new ways of explicitly stating information about semantics of computer programs, rather than having the meaning be just a coincidental side effect of a series of commands or transformations that even human beings have difficulty understanding.

Not only does the extra data help document the intention of the author of a program so that other people can obtain better understanding of writer's thought process, but due to its often formalized and standardized nature that is clearly made visible, automated tools can be efficiently used to extract and consequently utilise useful information as well.

Explicit semantic description in source code has been found invaluable for various purposes ranging from documentation to advanced static code analysis [9,11,14,34].

## 1.1 So wherein hides the meaning?

There are various sources of directly (and in a way redundantly) expressed semantics in computer programs, regardless of the programming language used. We shall only concern ourselves with formally and structurally stated origins of meaning, disregarding places where advanced analysis would be required (up to the point of natural language recognition on commented lines in source codes).

Various semantic models that are often not present directly within source texts (for example because the model is represented graphically) will not be considered either. It is however important to mention that models in whatever form they happen to appear in are often used to encode semantics that is directly or indirectly used when developing software. The approach of using the notion of models in this context is called Model–driven software development (MDSD) [15]. That said, the term *model* can be viewed very broadly and could ”contain” even concepts that we shall indeed consider.

### 1.1.1 Type systems

It has been clear since the beginning of programmable digital computers that programming in assembly languages that are for all intents and purposes devoid of explicit semantics is prone to errors. Type systems have been therefore introduced into some of the earliest high level programming languages available [28]. The very basic idea behind them is to have machines check certain properties of programs and reject ones that provably[1] exhibit absence of certain useful and/or desired behaviours. This is done by categorizing values by their type; a notion closely related to a mathematical set.

It is important to realize that types are by no means necessary and statically typed languages [5] often do not even preserve type information at runtime. Let us have a look at a simple example (imported files and namespace qualifiers omitted for brevity):

```
int main() {
    int numCoins = 42;
    numCoins += 30;
    cout << numCoins;
}
```

This C++ code compiled with clang 3.7 (-O3, -std=c++1y) emits the following assembly:

```
; ... output omitted ...
```

[1]within the proof /type system present in the language

```
main:
    push rax
    mov edi, std::cout
    mov esi, 72
; ... output omitted ...
```

We tell the type system that we have a numeric value (an integer) initialized to 42. We then proceed to increment it by 30 and print it to the standard output. Compiler is smart enough to inline [35] the `operator+=(int, int)` and then fold the constants [32] 30 and 42 into 72. This is then sent to standard output (by invoking `std::basic_ostream<char, std::char_traits<char> >::operator< <(int)`. Meaning of this program is very simple and is ultimately reflected as the number 72 printed.

Let us now consider a semantically richer code fragment:

```
struct CoinPurse {
    CoinPurse(int numCoins) : numCoins{numCoins} { }
    int numCoins;

    friend int operator+(CoinPurse lhs, CoinPurse rhs) {
        return lhs.numCoins + rhs.numCoins;
    }

    friend ostream& operator<<(ostream& os, CoinPurse const&
       c) {
        os << c.numCoins;
        return os;
    }
};
```

```
auto addCoins(CoinPurse numCoins, CoinPurse newCoins) {
    return CoinPurse{numCoins.numCoins + newCoins.numCoins};
}

int main() {
    CoinPurse myPurse{42};
    myPurse = addCoins(myPurse, 30);
    cout << myPurse;
}
```

The fragment compiles to:

```
; ... output omitted ...
main:
    push rax
    mov edi, std::cout
    mov esi, 72
; ... output omitted ...
```

This assembly should look familiar to an attentive reader. Indeed, it is identical to the previous one in spite of the fact that the corresponding C++ code is entirely different and semantically richer. We create a `CoinPurse` with 42 coins. Then we add 30 coins into it and finish by streaming the resulting amount of coins to the standard output. We tell the type system about `CoinPurse`, how to add two values inhibiting said type and how to print it; but in the end, all this extra explicit information is simply thrown away and reflected in the generated assembly as a printed constant.

There are various kinds of type systems available, categorized on several basis. Some classifications describe the expressiveness of the system, that is, how much can be said about behaviour of programs using types. One such categorization is obtained by placing type systems on Lambda cube [12]. We shall be using the terminology described therein throughout the thesis.

An interesting property type systems often have is that even though they exist to provide explicit meaning, type annotations can sometimes be implicitly inferred directly from definitions. Various type inference and type deduction algorithms exist, such as Hindley-Milner systems often present in ML-like functional languages and many kinds of Local type inference generally utilized by imperative languages [26, 29, 33].

This implicitly generated type information can then be treated exactly as though it was provided by a programmer. Let us take a look at a small example:

```
auto addTwoThings = [](auto const& first, auto const&
    second) {
     return first + second;
};
```

This function (encoded as a C++ polymorphic lambda expression) does not state its return type nor the types of the arguments it takes. All this information is deduced from the definition `first + second`. We know we are using a polymorphic `operator+` (just `+` in the source code), and we also know that we are returning a value inhibiting whatever type the `operator+` returns.

We could also make this information explicit as follows:

```
template <typename T, typename U>
std::decay_t<decltype(operator+(std::declval<T>(),
    std::declval<U>()))>
addTwoThings2(T const& first, U const& second) {
     return first + second;
}
```

Do note that a polymorphic type signature is being inferred, which is then made monomorphic by instantiating appropriate templates on demand by invoking `addTwoThings2(42, 5)` (`42` and `5` are constants of type `int`), resulting in the following definition:

```
int addTwoThings3(int const& first, int const& second) {
    return first + second;
}
```

Type inference in functional languages uses similar concepts to derive type annotations from definitions. We would like to point out that once again, this step is absolutely not required for the program to work but we choose to do it in order to externalize information that are somewhat hidden within definitions.

### 1.1.2 Contracts

Contracts [25] are used to impose preconditions and postconditions on functions (in general, on values entering and leaving logically corresponding blocks of code) and are usually checked at runtime. Some languages provide direct support for contract programming. Let us take a look at an example of a contract in the D programming language [10].

```
long square_root(long x)
in
{
assert(x >= 0);
}
out (result)
```

```
{
assert((result * result) <= x && (result+1) * (result+1) >
   x);
}
body
{
return cast(long)std.math.sqrt(cast(real)x);
}
```

Some use more general features such as annotations. The following Java fragment (using the cofoja framework [6]) demonstrates this:

```
@Requires("x >= 0")
@Ensures("result >= 0")
static double sqrt(double x);
```

and some utilize various hacks (for instance coercing the type system into checking something that we understand is a contract) as demonstrated in the following C++ example:

```
constexpr int divisibleByTwo(int n) {
    return (!(n&1)) ? n : throw "Not divisble by two.";
}

void printEvenNumber(int x) {
    // Precondition
    auto r = divisibleByTwo(x);

    cout << r;
}
```

The important thing to note is that the extra information stated in contracts is just that; *extra* information. It is by no means required for the execution of programs and can also have negative impact on performance should it make its way into runtime, as is often the case with contracts. More importantly, detecting an error at runtime is sometimes not acceptable, which is a problem contracts usually share with dynamic type systems.

Sometimes the boundary between the discussed concepts is blurry. This is exactly because they serve the same purpose; semantically enrich computer programs so that they can be understood better by both human beings and by automated tools. Their definitions do overlap slightly, but type systems and contracts can be considered the two most common ways to introduce explicit (and redundant) semantic information into computer programs.

## 1.2 Semantics for fun and profit

So now that we have gone through all the trouble inventing (or abusing) language constructs to contain certain information about behaviour of programs, let us examine various ways of utilizing the information.

### 1.2.1 Semantics for fun (and much less for profit) - documentation

Formalized form of semantics can be automatically turned into documentation for human beings to read. Tools like Haddock [4] use structured comments, type annotations and project hierarchy to generate information about functions and their parts.

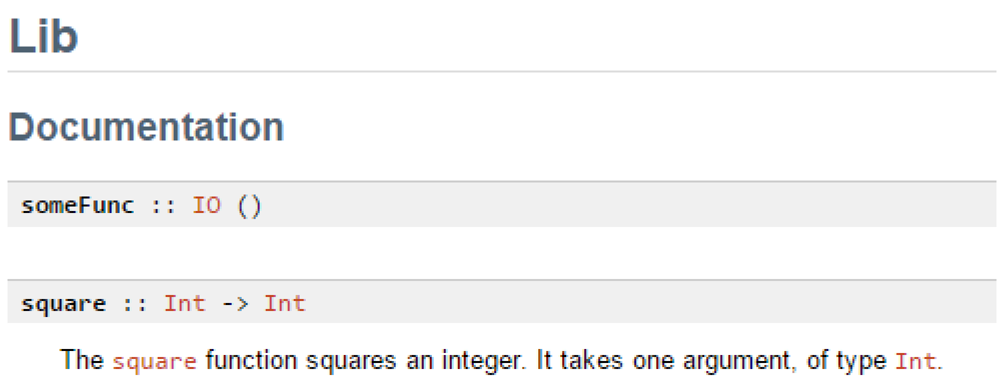
Let us take a look at a small example:

```
module Lib (someFunc, square) where
```

```haskell
-- |The 'square' function squares an integer.
-- It takes one argument, of type 'Int'.
square :: Int -> Int
square x = x * x

someFunc :: IO ()
someFunc = putStrLn "someFunc"
```

Running Haddock on the module generates the following HTML page:



Lib

Documentation

someFunc :: IO ()

square :: Int -> Int

The square function squares an integer. It takes one argument, of type Int.

**Figure 1 – 1** Haddock HTML rendering

In this case, the extra information about semantics of programs is used to create readable documentation for programmers to use. Taking this approach ensures that our documentation is never out of sync with the actual implementation. It is worth noting that type signatures are more often than not omittable in Haskell programs, as it is possible to infer them from function definitions using the Hindley-Milner type inference algorithm [33].

As useful as documentations can be, they are meant for humans to read and as such, they do not provide much benefit when it comes to automated processing.

### 1.2.2 Semantics for profit - correctness

The main reason why type systems and contracts have been introduced is code correctness. Type checking is a fundamental part of semantic analysis [9,14,34] and serves to give us some confidence about correctness of code. Contracts usually assert properties representing invariants that must hold to clearly signal what exactly has gone wrong. They often work on values at runtime, as opposed to working on types at compile time.

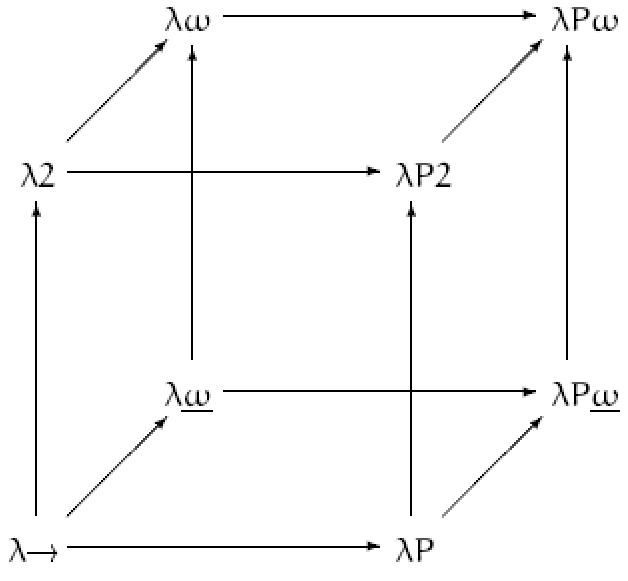


**Figure 1 – 2** Lambda cube

Let us very briefly overview Lambda cube [12] type systems and how they make use of type information. For the purposes of demonstration, the programming language Idris [16] shall be used, as it uniformly supports all the vertices of the cube. Even though we are using a functional language, the transition of the concepts to be demonstrated into the realm of imperative programming is fairly straightforward. An exception to this are type systems supporting dependent types, which are uncommon in the imperative paradigm and languages such as Xanadu [8] are still in experimental phases.

Note: while there is a very precise formal definition of what typing rules exist in the type systems present on the lambda cube itself [12], languages utilizing their concepts do not necessarily formalize everything and therefore all the examples to be shown serve only for illustrative purposes and might not make use of the exact typing relations of the corresponding type system they are based on

Note: type variables shall be denoted with lower case letters *a, b, c...*, concrete types with upper case letters *A, B, C...* and kinds (types of types) shall be denoted with asterisks ***, using a meta function constructor → to express multi-kind type functions such as *** → ***

The bottom left corner represents type systems deriving from simply typed lambda calculus [12, 28] and are depicted on the cube as $\lambda \rightarrow$. These include type checking on a very basic level, as demonstrated by the following example:

```
add : Int -> Int -> Int
add x y = x + y

add 30 12
```

Given an `add` function of type `Int` → `Int` → `Int`, function application *add 30 12* typechecks as both `30` and `12` are literals of type `Int`. We can therefore make sure we are not applying the operator `+` to values that cannot be added together.

By introducing parametric polymorphism [30] into the type system, we introduce a dependency of terms on types and form *System F* [22], depicted as $\lambda 2$ on the Lambda cube. Now we can formulate polymorphic functions that work on families of types, as demonstrated by the example:

```
id : a -> a
id x = x

id 42
```

Given an `id` function of type $a \to a$, function application `id 42` typechecks as there are no requirements imposed on the expected argument of the function and a literal of type `Int` is just as good as any other literal of some other type. By creating a parametrically polymorphic system, we have eliminated the need to have a separate `id` function for every type we could possibly require it for.

The next extension of the system (marked as $\lambda\underline{\omega}$ on the Lambda cube) introduces type functions (sometimes called type operators or somewhat less precisely type constructors), forming a Simply typed lambda calculus with type operators where types can dependent on other types. Let us have a look at a small example:

```
data NotAList : Type -> Type where
         Nil  : NotAList a
         Cons : a -> NotAList a -> NotAList a

aList : NotAList Int
aList = Cons 5 Nil
```

Given a type declaration `NotAList` of kind $* \to *$ (syntactically in Idris, `Type` $\to$ `Type`), we can invoke "function" application yielding the type `NotAList` by passing it a type parameter (in our case `a = Int`). (Do note that both `Nil` and `Cons` data constructors pass `a` to `NotAList` type constructor.) This way, we can create types that depend on other types, allowing us to generalize their behaviour. For example our `NotAList` type allows us to distinguish between `NotAList`s of `Int`s and `NotALists` of Chars, further enriching the number of invariants we can express about types (and ultimately values) at compile time.

Note: the data constructors `Cons` and `Nil` along with their application in the expression `aList = Cons 5 Nil` are only present for demonstrative purposes

Now we shall outline (by example) a vertex of the cube (denoted as $\lambda P$) representing a Dependently-typed type system, where types can depend on values (or, more specifically, on terms). Consider the following example:

```
data MyVec : Nat -> Type -> Type where
     MyNil : MyVec Z a
     (::)  : a -> MyVec k a -> MyVec (S k) a

myMap : (a -> b) -> MyVec k a -> MyVec k b
myMap f MyNil = MyNil
myMap f (x::xs) = f x :: myMap f xs
```

Here we have introduced a `MyVec` data type that is parameterized by a single type (`Type`) and indexed by a natural number (`Nat`). Do note that types are "parameterized" by a type but "indexed" by a value. Now we can create a `myMap` function of type `(a → b) → MyVec k a → MyVec k b`, which encodes a new kind of invariant: the length of the result list must be equal to the length of the input list (expressed as `k` binding the length for both the parameter list type and the result type).

The rest of the vertices on the cube are a combination of various aforementioned type systems with the most expressive one of them being Calculus of Constructions (CoC) [19].

Note: the last example actually belongs to CoC, but we find it very illustrative for what dependent types are capable of and as such has been used to outline the idea behind dependently typed systems

There exist, of course, many other type systems and type abstractions not represented on the cube. For example, subtyping is a very ubiquitous type abstraction that is often approximated in object oriented languages by inheritance, even though these concepts are not at all equivalent [18].

Most type systems are created in order to add safety and ensure additional dimension of correctness for programs they guard. They explicitly encode extra semantic information that is then used by type checkers to constrain the programmer while they also attempt to remain as expressive as possible.

Contracts are usually (but not necessarily) used to make up for what cannot be expressed within the system. Additional invariants that would otherwise be just assumed by the programmer can then be checked (generally at runtime) and provide diagnostics about what exactly has gone wrong in terms of what contract has been broken / what invariant does no longer hold.

Static analysis tools [11] are often overlaid on top of languages as well, utilizing type information (among another things) to detect possible problems with code and can sometimes advise on matters that a typechecker cannot - for example by taking into consideration factors other than types (direct code analysis, attributes, annotations, formalized comments, etc.).

Just to show a small example, here is a perfectly valid C++ program from the point of view of a C++ typechecker (qualifiers omitted for brevity):

```
int main() {
    int* x = nullptr;
    cout << *x;
}
```

A static analysis tool can use the fact that *int** is a pointer type and can be *nullptr* therefore. Such tool can then be taught that dereferencing it in its "null" state is invalid. Using this information, a static analysis tool can warn the user that a *nullptr* is being dereferenced.

Just to reiterate, all extra explicit semantic information in programs is technically not needed and is there to "make our lives easier". We voluntarily limit ourselves using type systems, contracts, annotations, etc. to inject information into our programs that is not necessary for machines, but of paramount importance for humans. We then use this information in a variety of ways; in this case to have the machine check correctness of our programs.

### 1.2.3 Semantics for profit (and profit) - implementation generation

Tools like the already mentioned Haddock [4] take (among other things) type information and transform it into something different. In this sense, we could say that types are a model and the resulting documentation is its implemented or an alternative representation; basically using ideas of MDSD [15] with generative approaches. Most type systems present in widespread programming languages are expressive enough to describe fair amount of behaviour and therefore are a good source of information for documentation generation. This is then supported and enriched by comments in natural language. Of course there is nothing preventing documentation generators to utilize other sources of semantics (from contracts up to analysing implementations themselves).

We think it is an excellent idea to take type information and transform it into something different. As a matter of fact, the main goal of the thesis is to devise a robust way to generate implementations of functions given their signatures and potentially additional information, such as specifications of data types they use. We can then map a function specification (in form of its type) to its executable implementation.

Using this approach extensively, there would be theoretically no implicit semantics in source texts whatsoever. Stating what we want in terms of types and then synthesising an implementation of such specification ideally without human intervention would lead to correct code by construction (provided the specification is accurate).

We shall return to further inspect this approach in a latter part of the thesis.

### 1.2.4 Why write it twice?

Some very powerful type systems have been devised over the years of research on the subject. We believe that even the most ubiquitous and common of them (such as the type systems present in languages used widely in the industry (in Java, C++, C# , etc.)) contain enough data so that an implementation can be derived for certain signatures.

Yet this information is very often underutilized to say the very least. As we have already mentioned, we more often than not use our type systems to check our implementations (which is, naturally, a very useful thing to do) and the next best thing they are commonly being used for is to create documentations (often with guidance of natural language in form of comments).

An important question leading to inception of this thesis is: If the machine knows that what I have written is provably[2] incorrect, why can it not give me an implementation (perhaps one that does not do *exactly* what I want) that is correct? This observation then leads to many further questions, a few of which we shall mention here:

- How do we actually do it?
- Is type information suitable for this task? (hint: it is)
- Shall we use type information only; there are other ways to encode semantics in code as it has been discussed; shall we perhaps also make use of preconditions and postconditions?
- Can a type system be expressive enough to be able to describe all invariants we would like?
- What type system do we choose?
- Should we finally decide on what sources of semantics we shall make use of, what functions can we devise an implementation for given information we have at our disposal?
- Shall we use an entirely different approach? (I have it on good authority that mathematics is good at dealing with abstract concepts - can we borrow something from it; or have we done so already?)
- How do we connect the process of analysis with code generation?
- What should the target / source language be - are there any requirements?
- Is all of this worth doing? (hint: it is)

[2]within the proof / type system present in the language

- Has anybody done something similar before?

Throughout the thesis, we shall discuss some of the aforementioned questions and perhaps even attempt to provide an answer for a few.

As the title of the enclosing chapter suggests, we would really like to go from something like this:

```
mystery : (a -> b) -> [a] -> [b]
```

to something like this:

```
mystery : (a -> b) -> [a] -> [b]
mystery _ [] = []
mystery f (x :: xs) = f x :: mystery f xs
```

automatically.

note: Idris syntax has been chosen arbitrarily; the programming paradigm not so much for reasons we shall discuss later. The example we have used has been selected very carefully. It is not something entirely trivial, nor contrived. It is a useful function that is small enough to not divert attention to details. We do hope that an attentive reader can indeed deduce the ”usual” name of the function in question.

The type of the function surely contains enough useful information to at least get to an approximation of what I would like it to do; or does it? This concept has been (albeit in a slightly different context) explored by Philip Wadler in his Theorems for free! paper [7].

# 2 From specification to executable code

As we have touched on before, the entire idea can be expressed as having a model representing a specification of some sort (a function signature, for example) and by utilising generative approaches, we can transform said model into a different representation (in this case, into a valid part of our target programming language that corresponds to an implementation of the given signature).

However, we shall not concern ourselves with approaches to this particular process (the process of Model driven software design), but instead, we shall focus on the process of transforming explicit semantics found in source texts into their ”natural” executable representations.

Mainly, we will concentrate on utilizing type systems more so than other sources of explicit semantics within source codes.

## 2.1 Preliminaries and assumptions

Before we discuss the current state of art of matters we find relevant for the thesis, we would like to point out that we shall sometimes be using concepts and terminology from mathematical logic as described by Curry-Howard isomorphism (sometimes also called Curry-Howard correspondence) [28].

Mathematics gives us an abstract and formal mechanism to approach analysing function signatures (types) as we can simply reinterpret the notion of ”type” and look at it as though it was a proposition. Then a proof of any given proposition coincides with a program of the corresponding type and proof normalization (in a way, simplification) reflects program evaluation.

Also, all analysis shall be done in the context of functional languages, as their mapping to logic calculi (which are extremely useful for constructing proofs, that happen to correspond with computer programs) is straightforward. In later chapters we shall examine various ways of doing a paradigm shift to an imperative world, which is arguably more prevalent, at least in the industry and overall more practically utilized by programmers.

The next assumptions made is that all functions shall be considered total. Reasoning about partiality would require a way to define what inputs the function does not make any sense for and as such is subject to further research. We reckon it is worth noting that partial functions can be converted to total functions, at least from the practical standpoint, at the cost of losing their isomorphic nature to their ”mathematical counterpart” (the function they are trying to model). This conversion can happen for example by wrapping return types in a meta-type that can represent a state of failure. The aforementioned type is often called *Maybe*, *Optional* or *Nullable*.

We should also mention that language ”purity” shall be assumed. This means that no arbitrary side effects are allowed and functions do what they advertise in their signatures. This property is fairly common among functional languages (Haskell, Idris, etc.) but relatively rare in the imperative paradigm. There are, however, some attempts to incorporate at least some form of purity into impure imperative languages [21]. We make this assumption mostly because it is much cleaner and simpler not having to formalize arbitrary side effects that could ”randomly” occur; as a matter of fact, no commonly used logic calculus allows arbitrary side effects. It is also worth noting that non-termination which is related to totality can be considered a side-effect.

Moreover, we will only consider a type system that is marked as $\lambda\omega$ on the lambda cube, that is, it is based on System F and also supports type operations. In view of the Curry-Howard isomorphism, its logic counterpart is second-order intuitionistic logic with universal quantification for types. Haskell programming language without any extensions is based on this system.

Even though there are more expressive type systems available today (for example in languages that are based on CoC; or in languages that follow pure type systems [24], such as Henk 2000), we believe that our choice is a good starting point with reasonable balance of expressiveness and complexity. We shall inspect possibilities introduced by more expressive type systems towards the end of the thesis.

The last assumption we shall make is that only parameterized (both constrained and unconstrained) signatures shall be taken into account. The intuition behind this is thoroughly explored in the already mentioned paper Theorems for free! [7]. The basic idea just underlines the fact that, of course, it is impossible to generate a sensible implementation (from our standpoint) for something like `Integer` $\rightarrow$ `Integer`.

## 2.2 Signature analysis

Setting out to generate an implementation from a signature, we believe it is important to "analyse" the type first to extract information it contains. The result of the analysis (whatever it might be) needs to be usable for purposes of the successive synthesis / implementation generation.

### 2.2.1 Formal logic and proof assistants as basis

The idea to describe a proof (and coincidentally, a program) in incremental steps, decomposing a proposition gradually until we can prove it, is not new. Various proof assistants such as Isabelle [27] and Coq [13] have been developed, based on numerous kinds of logic calculi, corresponding (not necessarily isomorphically) with various type systems.

Let us demonstrate why this should be of any relevance to us on a simple example, using the Coq proof assistant.

Consider the following example:

```
Variables A B C : Prop.

Lemma implication_transitive: ((A -> B) /\ (B -> C)) -> (A
   -> C).
```

Suppose we have a proposition stating that implication (denoted by $\rightarrow$ in Coq) is transitive. That is, if A implies B and ($\wedge$ in Coq) B implies C then A implies C. Coq gives us the following interactive environment (after inputting the aforementioned formulae):

$$\frac{A, B, C : Prop}{(A \rightarrow B) \wedge (B \rightarrow C) \rightarrow A \rightarrow C}$$

Note: Coq removes extra parentheses for us (logical and binds stronger than implication and implication is right associative)

We have only one goal to solve, namely the conclusion of the current tree. The horizontal line separates assumptions (listed above it) and the conclusion (shown beneath the line).

In Coq, proofs are carried out by using simple operations called *tactics*. These let us manipulate the quasi "proof trees". In order to prove something, we need to "solve all goals". Solving a goal generally means that we already have in assumptions what we are trying to prove, similar to the way that sequent form calculi [17] use the identity axiom rule.

Just to demonstrate a simple tactic and its relationship with an inference rule, considering the following *identity rule* (in propositional sequent calculus):

$$\frac{}{A \vdash A}\,\text{Id}$$

Note: A is considered atomic

One of the corresponding Coq tactics is called *assumption*, which is documented in the reference manual [13] as follows: *"This tactic looks in the local context for a hypothesis which type is equal to the goal. If it is the case, the subgoal is proved. Otherwise, it fails."*

From the wording we can see the *identity axiomatic inference rule* and the *assumption tactic* are not in one to one correspondence, but their purpose is identical. As a matter of fact, identities where the context contains more assumptions than necessary can be transformed to their "*pure*" form by using *weak structural rules [17]* that are logical consequence of commutativity of contexts (represented by the so called "exchange structural rules") and the fact that contexts can be weakened (corresponding rules are contraction and weakening).

Intuitively, given $A$, $B$ and $C$, we can certainly prove $B$, even though the *Identity rule* does not apply immediately, should we consider the sequent form $A, B, C \vdash B$. This very intuition is captured in Coq's tactic *assumption*.

Note: other tactics used shall not be explained in detail; an interested reader may refer to reference manual [13] for their exact descriptions

Continuing the example we started, a common strategy to solving goals that are in the form of implication is to assert their antecedents. We therefore move on by using the *intro* strategy.

```
// ... output omitted
intro.
```

```
// ... output omitted
```

This results in the following environment:

$$\frac{A, B, C : Prop; H : (A \to B) \land (B \to C)}{A \to C}$$

Note: tactics are allowed to introduce new arbitrary fresh names to newly constructed hypotheses (should user defined names not be provided); in this case the name $H$ is chosen by Coq

Applying *intro* one more time finally yields:

$$\frac{A, B, C : Prop; H : (A \to B) \land (B \to C); H0 : A}{C}$$

Note: assumptions shall be separated by semicolons

The only way to get C from the available assumptions is to somehow find B (so as to use B $\to$ C). The problem is that the implication we are looking for is not contained in the assumptions as an atom. Therefore we have to start by eliminating the addition, using the *elim* tactic.

```
// ... output omitted
elim H.
// ... output omitted
```

Which gets us to the following environment:

$$\frac{A, B, C : Prop; H : (A \to B) \land (B \to C); H0 : A}{(A \to B) \to (B \to C) \to C}$$

Now we introduce multiple hypotheses with a single tactic *intros*.

```
// ... output omitted
intros.
// ... output omitted
```

The resulting environment is:

$$\frac{A, B, C : Prop; H : (A \to B) \land (B \to C); H0 : A; H1 : A \to B; H2 : B \to C}{C}$$

We can finally get C from B → C using the *apply* tactic.

```
// ... output omitted
apply H2.
// ... output omitted
```

Which leads us to:

$$\frac{A, B, C : Prop; H : (A \to B) \land (B \to C); H0 : A; H1 : A \to B; H2 : B \to C}{B}$$

And finally, we get B from A → B with *apply*, resulting in the following environment:

$$\frac{A, B, C : Prop; H : (A \to B) \land (B \to C); H0 : A; H1 : A \to B; H2 : B \to C}{A}$$

At this point, we have a hypothesis H0 : A, which happens to be equal to the goal we are trying to solve. Using the already mentioned tactic *assumption*:

```
// ... output omitted
assumption.
// ... output omitted
```

solves the goal and Coq tells us that there are ”No more subgoals”. For the sake of completeness, the *Qed* tactic finishes the proof, yielding the following summary:

```
intro.
intro.
elim H.
intros.
apply H2.
apply H1.
assumption.
```

We have therefore provided a constructive proof to the lemma *implication_transitive.*

Shifting our focus to the world of types, we are looking at a function called *implication_transitive* that is of type `(a → b, b → c) → a → c`. (Using Haskell notation and the fact that product types are tuples, concretely in this case, pairs and implication represents a function type.)

The signature `f :: (a → b, b → c) → a → c` resembles the one of function composition (denoted `(.)` in Haskell), which is `(.) :: (b → c) → (a → b) → a → c`. If we *uncurry* the composition operator, we get `uncurry (.) :: (b → c, a → b) → a → c`, which is almost identical to what we were proving except for the fact that the operands for `(.)` are in reverse order, so as to go by the convention of (f ∘ g) being interpreted as f after g, that is, f(g(x)).

Now that we have the ”proposition as type” covered, we should be able to convert our constructive proof (expressed as a sequence of tactics in our case) into a meta-program (or view / reinterpret it as such). This is a key idea we are going to build upon; given a proposition, find the corresponding function type, find a constructive proof of the proposition and transform it to a meta-program which can then be translated to an executable implementation.

### 2.2.2 Automated proof construction

One of the goals of the thesis is to devise a way to create executable implementations automatically. Coq however, is mainly a proof assistant - while the way of constructing proofs can model construction of programs in a reasonably straightforward manner as will be shown shortly, the process requires a lot of human input and as such is insufficient for our purposes. We therefore need to find a way to automate the process, at least for a subset of interesting propositions / function types. This particular task has yet to be explored fully, even more so in the context relating to practical code generation potentially for mainstream languages used throughout the industry. Let us take a look at a few ways that this task has been attempted.

Firstly, Coq does offer a few tactics capable of proving numerous propositions without human intervention. (In general, most theorem provers available do.) Detailed descriptions are available in the Coq reference manual [13]. Here is an overview of a few:

- *auto* tactic tries to apply *assumption* and *intros* repeatedly with bounded recursion depth
- *tauto* tactic is based on contraction-free sequent calculus [20]
- *intuition* tactic tries to apply *tauto* on all subgoals, which are simplified beforehand
- *linear* tactic is implemented to model direct predicate calculus [23]

The general idea behind them is to take a fragment of a logic calculus (for example by restricting an existing one, such as removing the contraction rule from the first order Gentzen's sequent calculus, which is what the *linear* tactic does) and build either a formal or a semi-formal procedure (with heuristics) applying tactics in attempt to construct a proof. The heuristic approach is very pertinent for our solution.

Secondly, there are proof engines that take some of the ideas behind theorem provers and try to make them more accessible to "practical" programmers. Edwin Brady's Ivor [2] implements dependent type theory and facilitates an API that makes it accessible to Haskell. There is also Zeno [31] theorem prover, which generates proofs as Isabelle [27] theories, although it is important to note that it uses a dialect of Haskell and function definitions, rather than declarations, which is primarily in spirit of automated checking rather than actual generation. This particular proof engine is capable of incorporating recursive data types and as such can use recursive constructors as basis for inductive proving. The ability to handle user-defined recursive data types is of great importance for the thesis given its rather practical approach to implementation generation.

Lastly, there are interactive modes and hints generally integrated within IDEs for (usually dependently typed) functional languages that speed up the process of writing programs by generating various code fragments for the programmer. For instance, Idris can generate equations corresponding to constructors of a type and can also do context aware case analysis and splitting; this functionality is included in for example Atom using the Idris Mode for Atom [1].

The idea of proving by induction on an algebraic data type's constructors (corresponding to Idris generating equations for a data type) is of utmost relevance should we want to provide a way to handle user-defined data types and generate proofs / programs based on provided definitions.

To summarize and describe our approach to proof synthesis, we are going to use the idea of tactics to decompose types of functions into a series of primitives. We will also take note of user defined types. Then we develop heuristics to chain tactics together, hopefully solving the requested goal. Transformations to implementations will be discussed shortly.

### 2.2.3 Modelling the analysis

As we have mentioned already, we are going to use a model similar to what proof assistants use. Let us discuss the most fundamental concepts from the standpoint of a System F based type system (that is, we shall not concern ourselves with sigma and pi dependent types, which are extensions of universal and existential quantifiers) as well as outline the way they are handled internally.

The analysis starts by obtaining kind definitions (in a way, algebraic specifications or simply, data definitions) and a function signature, representing a type for which we would like an implementation to be synthesized. Two separate languages have been created for this purpose; one for function signatures and one for type definitions. The input is parsed, checked for semantic consistency (for example rejecting function signatures that constrain type variables that are unused) and normalized to be ready for internal processing (for example all functions are curried).

Now an initial environment is created, containing the signature that is being analysed (for the purposes of recursive invocation) and known definitions from both generic type restrictions and kind specifications.

Let us demonstrate on a simple example:

```
// Type definition
Specification List {
    parameterization {
        Any a
    }
    operations {
        Nil,
        Cons : a -> List a
    }
}
```

The fragment of code is an example of our specification definition language. We define a type called `List`, that is of kind $* \to *$. The type argument named `a` is unconstrained (represented by prefixing it in the ”parameterization” block with the typeclass ”Any”). Value constructors of this type along with their definitions are listed in the ”operations” block and are `Nil` and `Cons` in this particular scenario.

Note: return types of data constructors are always inferred, so for example `Nil` is equivalent to `Nil : List a`

```
// Signature to be analysed
(Any a, Any b) => map : (a -> b) -> List a -> List b
```

This is a small example of our signature specification language (resemblance to Haskell, Idris and other function languages is naturally purely coincidental). Here we declare a function called `map`. Its signature contains two unconstrained (in the typeclass `Any`) type variables named `a` and $b$ and it takes two arguments, namely a function from `a` to `b` and a `List` of `a`. The notation should be immediately familiar to functional programmers.

Note: as we have already hinted at, all functions are normalized to their curried forms and as such they are all treated as unary; `map` therefore takes a function `a` $\to$ `b` and returns a function from `List a` to `List b`

The resulting initial environment is as follows:

```
map  : ((a -> b) -> (List a -> List b))
Nil  : List a
Cons : (a -> (List a -> List a))
--------------------------------------
((a -> b) -> (List a -> List b))
```

Our goal (that is, what we are trying to implement) is listed beneath the horizontal line and is directly taken from the type signature of the function we are trying to find an implementation for. Above the line is our context, which contains all our available assumptions. Simply put, it has functions (or constants, nullary functions) that we can use. `Nil` and `Cons` have been provided in the specification and `map` is the function in question; as we can recursively invoke it, it is considered to be a part of the context.

Now the entire "proof construction" process is conducted by applying various tactics (term borrowed from Coq). The order in which tactics are applied is devised heuristically, more on the topic shortly. Tactics manipulate the environment in various ways. While the terminology is borrowed, the concepts are only similar, not identical. As Coq is designed with human assistance in mind, its tactics are of form *<tactic_name> <arguments...>*, with the arguments being supplied by the user.

Our analysis is however completely automatic - we heuristically devise what tactic to use and should the invocation be successful, arguments are automatically synthesized and returned as parameterized tactics, usually in form *tactic_name<hypothesis>*. Compound and derived tactics sometimes take the form *tactic_name[components...]*. Data constructors for elements of type T shall be $c_0^T$ ... $c_n^T$ and their respective arguments $a_0^{c_i}$ ... $a_m^{c_i}$. The capital letter $\Gamma$ shall be used to represent context "remainder"; let $\Delta$ be explicitly mentioned context contents, then $\Delta \cup \Gamma$ is the entire context and

$\Delta \cap \Gamma = \emptyset$; hypotheses shall be named $h_1 ... h_n$ with their respective types listed after the colon symbol ($h_1$ : A). Here is an overview of their semantics as well as notes on the way they are handled internally.

Note: some names are Coq-like but their functionality may not be in one to one correspondence

Note: internally, modified versions of the tactics to be mentioned might be used, for the sake of simplicity of implementation, clarity or efficiency

**Trivial**

The trivial tactic goes through available context and solves it if and only if there is exactly one hypothesis that shares its type with the goal. All proofs need to finish with this tactic, otherwise the proposition cannot be solved (except in rare cases, where a yet unintroduced tactic *apply* can finish the proof for nullary functions). The tactic roughly corresponds with the context/goal identity rule.

$$[\nexists k \in \mathbb{N} : (k \neq i \wedge (h_k \in \Gamma : A))] \, \frac{}{h_i : A, \Gamma \vdash A} \, \text{trivial}{<}h_i{>}$$

Internally, context is linearly searched and a hypothesis of a corresponding type (with the goal) is returned in case it is unique; its name is appended to the tactic name. We shall postpone an example of this tactic for a moment.

**Intro**

The intro tactic asserts the antecedent of the outermost implication. This means that applying it to a function introduces its first argument into the context (keep in mind that all functions are curried at this point). Every proof will contain at least one invocation of intro (or one of its derivatives). A fresh name (one that is not yet within the context) is chosen for the introduced argument. The tactic is similar to implication introduction.

$$[h_i : \_ \notin \Gamma]\,\frac{h_i : A, \Gamma \vdash B}{\Gamma \vdash A \to B}\,\text{intro}{<}h_i{>}$$

Internally, invocation of intro expects the goal is of function type, unwraps it into a pair of argument and return type, moves the argument to the context and chooses a fresh name for it (derived from its type). The return type is untouched and kept as the remaining goal. It is again important to note that the function type as the goal in question will have been curried by the time intro is invoked and therefore the resulting goal can still be of function type.

A small example demonstrating the tactics we have discussed so far:

```
id : a -> a
```

Using intro results in successful invocation, yielding intro$<a_0>$.

$$\frac{a_0 : a, \Gamma \vdash a}{\Gamma \vdash a \to a}\,\text{intro}{<}a_0{>}$$

Now the tactic trivial completes the proof, yielding trivial$<a_0>$, thus forming the following proof tree:

$$\cfrac{\cfrac{}{a_0 : a, \Gamma \vdash a}\,\text{trivial}{<}a_0{>}}{\Gamma \vdash a \to a}\,\text{intro}{<}a_0{>}$$

Note: we have omitted enumerating complete context contents (using Γ instead), please see the preceding part about ”initial environment”

Note: naming scheme shall be discussed later

### Intros

The intros tactic is functionally equivalent to calling intro repeatedly. Intuitively, it is useful for multi-argument functions (disregarding their curried representation). The tactic is similar to introducing implication several times.

$$[\alpha]\frac{h_k : A_0, h_{k+1} : A_1, ..., h_m : A_n, \Gamma \vdash B}{\Gamma \vdash A_0 \to A_1 \to ... \to A_n \to B}\text{intros}{<}\emptyset \cup \{h_k ... h_m\}{>}$$

where $\alpha$ is $\{h_k : \_ ... h_m : \_\} \cap \Gamma = \emptyset$

Internally, intros calls intro repeatedly but does not require that the goal is of function type. Instead, it simply stops its execution if it can no longer introduce a new hypothesis; the resulting environment is equivalent to calling intro zero or more times.

Consider the following example:

```
constant : a -> b -> a
```

With intros, we get intros[intro<$a_0$> intro<$b_0$>], and the following environment:

$$\frac{a_0 : a, b_0 : b, \Gamma \vdash a}{\Gamma \vdash a \to b \to a}\text{intros[intro}{<}a_0{>}\text{ intro}{<}b_0{>}]$$

### Clear

The clear tactic removes a hypothesis from the current context. It is similar to context weakening.

$$\frac{\Gamma \vdash A}{h_i : \_, \Gamma \vdash A}\text{clear}{<}h_i{>}$$

Internally, the requested hypothesis is simply discarded.

Let us take a look at the same example as above, the *constant* function:

```
constant : a -> b -> a
```

Clearly, we do not need $b_0$, even though it has been introduced already. We can therefore clear it from the context, with clear<$b_0$>, continuing from the aforementioned proof tree we get:

$$\dfrac{\dfrac{a_0 : a, \Gamma \vdash a}{a_0 : a, b_0 : b, \Gamma \vdash a}\text{clear}{<}b_0{>}}{\Gamma \vdash a \to b \to a}\text{intros[intro}{<}a_0{>}\text{ intro}{<}b_0{>}]$$

A small note on the clear tactic. It may seem completely irrelevant considering the relaxed form of our "identity" rule (trivial). We shall demonstrate its pertinence in the generative phase of the thesis later on.

**ElimSum**

The elimSum tactic deconstructs an inductively defined type into its constructors, yielding a new subgoal and a subtree for each unique constructor the type in question has. It asserts the hypothesis it is being used on is indeed of sum type. The resulting subgoals take the following form: all of the arguments of constructors are curried and the remaining corresponding subgoal is suffixed as the rightmost returned type. This tactic is similar to elimination of disjunction.

Let the number of $c$ constructor arguments be $N\ c$.
And let $M\ c\ r$ be $a_0^c \to ... \to a_{Nc-1}^c \to r$ for $N\ c > 0$, otherwise $r$.

$$\dfrac{h_i : A, \Gamma \vdash M\ c_0\ B \quad ... \quad h_i : A, \Gamma \vdash M\ c_{n-1}\ B}{h_i : A, \Gamma \vdash B}\text{elimSum}{<}h_i{>}(c_0^A...c_{n-1}^A)$$

where n is the number of constructors of type $A$

This tactic is used to coinductively view all of the parts of any given discriminated union type; that is, consider all of the possible origins of the value of the type in question. For a functional programmer, this is the equivalent of doing case analysis on an algebraic data type with multiple data constructors. For an imperative programmer with access to virtual methods (or ideally, multi-methods), this is similar to overriding a function in several subtypes of T and then inspecting an object of said type at runtime to decide what behaviour it should have, based on its concrete subtype.

Internally, elimSum looks up all available data constructors of type T and for each of them, creates a copy of the current environment. Then, based on the arguments of each of the constructors, creates corresponding subgoals by synthesizing functions, taking the original goal as the final return type and prefixing the arguments of the constructor in question to it. The resulting environments are then enqueued to a deque.

Here is a simple (and very contrived, I am sorry) example (using the already defined List specification) demonstrating the tactic:

```
g : List a -> b
```

We first intro the argument. Then we continue by invoking elimSum $list_{a0}$; this yields two new environments (with their respective contexts and goals), the first one being `elimSum<`$list_{a0}$`>(Nil)` and the other `elimSum<`$list_{a0}$`>(Cons)`. The resulting proof tree is as follows:

$$\dfrac{\dfrac{list_{a0} : List\ a, \Gamma \vdash b \qquad list_{a0} : List\ a, \Gamma \vdash a \to List\ a \to b}{list_{a0} : List\ a, \Gamma \vdash b}\,\text{ruleElim}}{\Gamma \vdash List\ a \to b}\,\text{intro}{<}list_{a0}{>}$$

where `ruleElim = elimSum<`$list_{a0}$`>(Nil) / elimSum<`$list_{a0}$`>(Cons)`

It is important to note that the split on the topmost part of the tree represents two separate environments. Naming has been kept ambiguous for brevity (as the example is complete anyway).

**Apply**

The apply tactic represents complete function application. Given an uncurried *n-ary* function *f* ultimately returning $T$ in an environment $\epsilon$ and a goal of type $T$, synthesizes $n$ new environments $\epsilon_0$ ... $\epsilon_{(n-1)}$ that are copies of the original environment $\epsilon$ except their respective goals correspond to the types of the arguments of *f*, that is, the goal of $\epsilon_k$ is set to a new type $U_k$ which coincides with the $(k+1)st$ argument type. It is important to note that value constructors of arbitrary data types are considered functions as well. The original goal of type $T$ is eliminated in the process of application. If the function in question is nullary (this is reasonably often the case with data constructors), no new environment is created. This tactic roughly corresponds to implication elimination.

Let $\rho\ f$ be the arity of *f*.

$$\frac{h_i : \alpha^A, \Gamma \vdash \beta \quad \ldots \quad h_i : \alpha^A, \Gamma \vdash \omega}{h_i : \alpha^A, \Gamma \vdash A}\ \text{apply}{<}h_i{>}(0...ar/ar)$$

where $ar$ is $\rho\ h_i - 1$ and $\alpha^A$ is either a function or a data constructor of any arity ultimately returning $A$, that is, takes the shape of $A$ (in which case the rule is an axiom) or $\beta \rightarrow \gamma \rightarrow \ldots \rightarrow \omega \rightarrow A$

Intuitively, *apply* facilitates usage of function-arguments in higher order functions. The tactic is implemented by unifying the current goal with the return type of the applied function in its uncurried form, then creating copies of the original environment and replacing their goals.

Consider the following example:

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

Using apply gives us `apply<` $func_{a\_b\_0}$`>(1/1)`, generating a single new environment. Pre-intros omitted for brevity, the end of the proof is left as an (admittedly *trivial*) exercise for an interested reader (hint: what does $\Gamma$ contain that is not explicitly listed after invocation of intros?).

$$\cfrac{\cfrac{func_{a\_b\_0} : a \to b, \Gamma \vdash a}{func_{a\_b\_0} : a \to b, \Gamma \vdash b}\ \text{apply}< func_{a\_b\_0} >(1/1)}{\dots}\ intros[intro < a_0 > intro < b_0 >]$$

### 2.2.4 Heuristics

Now that we have tactics for manipulating our definitions, we need to find a way to apply them in some order such that we can decompose them fully up to being able to use the trivial tactic, or in some cases the nullary apply, in order to finish the proof.

Our heuristics-driven analysis consists of several composed algorithms, but before we move on, let us very briefly outline the implementation so that we can be clear what the fragments of pseudocode to come mean. First of all, the entire process runs in a state monad (henceforth denoted as *M*) consisting of *environments*, *inferences* and *generations*.

- An *environment* represents one *goal* and its corresponding *context*.
- An *inference* is a string representing a sentence in our quasi-language for tactics - it records analysis that has been conducted and can also be used as an intermediate representation for the not-yet-mentioned implementation generation.

- A *generation* is a string in our target language, should we choose to utilize it during the analysis. If all we are after is the analysis which we would like to transform into an implementation in a separate step, it can be ignored.

We then apply a series of tactics-using-algorithms that gradually transform our initial environment (see the previous chapter) into a solved one, whilst transitioning between one or multiple environments and building up an inference string and possibly also a generation string. Before we begin, we lift the relevant initial environment, an empty inference string and an empty analysis string into $M$ (this operation of type `a → m a` is called *return* in Haskell), then we proceed with the following heuristic algorithms:

**Solve by Introduction**

The process begins by introducing a single argument with *intro* immediately followed by an attempt to invoke *trivial*, trying to solve the goal. Should this succeed, we *clear* the used hypothesis for the current subgoal. Do note that this step is redundant if it is also the only subgoal present, as the proof would hence be complete. Immediate removal of all hypotheses we apply any tactic on captures the intuition that generally, we only want to use each argument once for each equation resulting from elimination. If at this point the goal is unsolved, we attempt to use *elimSum* in order to deconstruct the most recently introduced proposition (a function argument) if it is a user-defined algebraic data type with specified data constructors. Finally we *clear* the eliminated hypothesis (for the aforementioned reason). This procedure is then repeated until either all goals are solved or all arguments have been introduced, that is, the resulting goals are no longer of function type (this condition shall be marked as ”finished” in the following pseudo-code).

Do note that the *elimSum* tactic can create multiple new environments, therefore also multiple new goals, which can in turn be again of function type. It is also important to note that this requires a bit of extra work for recursively defined types. Consider the `Cons` constructor of `List a`, which expands to `a` → `List a` → `List a`. We would never halt if we were to expand it several times. We also assume implicit monadic context over whatever is listed after *with* clause, using *do notation* to handle *monadic binds*, that is, operations of type `m a` → `(a` → `m b)` → `m b` ((»=) in Haskell).

```
solveByIntroduction : (initialEnvironment : M) -> M
    while not finished
        with initialEnvironment
            do intro
               h <- trivial
               clear h
               h' <- elimSum
               clear h'
```

As an example that can be solved by this step, consider the following:

```
id : a -> a
```

*Intro* is tried first, introducing the leftmost (and in this case, the only) argument of type a, yielding *intro*<$a_0$>, which is appended to the inference string to the corresponding (and again, the only one) environment. Now the *trivial* tactic is tried and we can immediately unify the goal with the hypothesis $a_0$, appending the appropriate inference step to the log and solving all of the available goals and proving the proposition. We could stop here but the implementation is designed in such way so that also *clear* is then tried, resulting in *clear*<$a_0$> being synthesized and again, appended to the inference string.

The final environment:

```
id   : (a -> a)
```

```
---------------
// <an empty line, no goals remaining>
```

Available hypothesis are above the dashed line while the goal is below. Note that this is very similar to the initial environment, except for the goal is no longer present (it has been solved). An intermediate environment between *intro* and *trivial* also contained the $a_0 : a$ hypothesis, but it was immediately *cleared*.

The final inference string:

```
intro<a_0> trivial<a_0> clear<a_0>
```

**Solve by Apply**

At this point, we either have a solution or we have introduced hypotheses that are of function type. We therefore try to use *apply*, synthesizing a new environment for each argument of the applied function. Do note that if the function is nullary (as we have mentioned before, this is not uncommonly the case with data constructors for certain types, such as `Nil` for `List`), the *apply* tactic solves the goal immediately. After a function has been applied successfully, we attempt to solve all of the created environments with *trivial*. This process is then repeated until we either have a solution or until we exit with an error.

Note: it is important to note that if we had at any point encountered any shape of identity (suppose `((a → b) → c) → (a → b) → c`), we would have solved it by *trivial* and there would be no need to use *apply*

```
solveByApply : (introducedEnvironment : M) -> M
    while not finished
        with introducedEnvironment
            do h <- apply
               clear h
```

```
                    h' <- trivial
                    clear h'
```

Consider the following example where solve by apply is required after solve by introduction:

```
flip : (a -> b -> c) -> b -> a -> c
```

After having run solve by introduction, we get the following environment:

```
flip            : (a -> b -> c) -> b -> a -> c
func_a_b_c_0    : (a -> (b -> c))
b_0             : b
a_0             : a
----------------------------------------------
c
```

So we still do have one goal to solve, namely of type `c`. We now try to use a viable candidate to invoke *apply* with. We therefore attempt to unify the rightmost return type of every hypothesis of function type with the goal. Calls that would lead to recursion (in this case `flip`) get a penalty (they "cost" more) and therefore $func_{abc0}$ is the most viable candidate, resulting in *apply<*$func_{abc0}$*>(1..2/2)* being invoked, synthesizing two new environments, one with goal of type `a` and the other with goal of type `b`. Also, their corresponding inference strings have the appropriate *apply* appended. The applied hypothesis is then cleared with *clear<*$func_{abc0}$*>*. Now we attempt to solve both of the freshly introduced goals with *trivial*, resulting in *trivial<*$a_0$*>* in one of them and *trivial<*$b_0$*>* in the other. Of course, all inference strings are adjusted accordingly.

The final environments are now two, one for each of the arguments of the applied function:

```
flip : ((a -> (b -> c)) -> (b -> (a -> c)))
b_0  : b
-------------------------------------------
// <an empty line, no goals remaining>

flip : ((a -> (b -> c)) -> (b -> (a -> c)))
a_0  : a
-------------------------------------------
// <an empty line, no goals remaining>
```

And the resulting inference strings (also including inference steps from solve by introduction):

```
intro<func_a_b_c_0> intro<b_0> intro<a_0>
   apply<func_a_b_c_0>(1/2) clear<func_a_b_c_0> trivial<a_0>
   clear<a_0>

intro<func_a_b_c_0> intro<b_0> intro<a_0>
   apply<func_a_b_c_0>(2/2) clear<func_a_b_c_0> trivial<b_0>
   clear<b_0>
```

Distilling the algorithms, we get:

```
Type = Scalar | Parameterized | Function
Context = [(String, Type)]
Env = (Type, Context)

inferImplAux : (seen : [Env]) -> (env : Env) -> [Env]
inferImplAux seen env =
  let continue e = case applyAux e of Nothing -> []
                                      Just result -> result
      (i, env')  = intro env
      (t, env'') = trivial env'
        envs     = map (clear i) $ elimSum env'' i in
          if env `elem` seen then continue env
          else case env'' of Nothing -> concatMap
             (inferImplAux (env :: seen)) envs
                           Just (h, _) -> continue $ clear
                              env'' h

applyAux : (env : Env) -> Maybe [Env]
applyAux env =
  let len = length . snd $ env in
   if len < 1 then Nothing
   else let envs = apply env (snd . last . snd $ env) in {
     case envs of Nothing -> applyAux shorterEnv
       where shorterEnv = take (pred len) . snd $ env;
         Just (h, envs') =
           let envs''   = map (clear h) envs'
               envs'''  = map trivial envs''
               solved   = filter isJust envs'''
               unsolved = filter isNothing envs''' in
                if null unsolved
```

```
                  then Just $ concatMap fromJust solved
                  else let result =
                    concatMap inferImplAux [] unsolved
                    in if not . null $ result then Just result
                       else Nothing
   }

doAnalysis = inferImplAux []
```

The algorithm is not sound due to its greedy nature of invoking *trivial* (or its friend nullary *apply*) whenever possible nor complete, considering we can end up invoking mutually recursive functions ad infinitum, should the recursive call of the function to be analysed have the lowest cost, which is represented by its position in the context.

Consider any type with a recursive data constructor, such as the `Cons` injection of `List`. It will always end up costing more than its would-be recursion base case (for `List` it is `Nil`), which would be prioritized therefore completely ignoring the recursive case. An entirely arbitrary function such as `f : a → List a → List b → (a → c) → List c` will therefore still simply give us:

```
f :: (a -> (List a -> (List b -> ((a -> c) -> List c))))
f = \a_0 -> \list_a_0 ->
    case list_a_0 of Nil -> \list_b_0 ->
        case list_b_0 of Nil ->
            \func_a_c_0 -> Nil
                        Cons b_0 list_b_1 -> \func_a_c_0 ->
                           Nil
                    Cons a_1 list_a_1 -> \list_b_0 ->
        case list_b_0 of Nil ->
            \func_a_c_0 -> Nil
```

```
                    Cons b_0 list_b_1 -> \func_a_c_0 ->
                      Nil
```

Note: please do note that this is more of an overview rather than the actual implemented algorithm considering it is written in an imperative language and is more involved, probably beyond what would be useful to describe here

### 2.2.5 On naming scheme and language definitions

Before we move on to describing transformations into executable code, we would like to take a moment to provide a brief overview of the chosen naming scheme as well as describe our specification languages.

**Naming scheme**

Names are derived from types. There are three kinds of types available, namely *scalar types* of form $T$, $U$, $a$, $b$, etc. *Parameterized types* taking shape $T\ x$, $U\ x\ y$, etc. and *function types* such as $A \rightarrow B$.

For *scalar types*, we proceed as follows: given a hypothesis of type $T$, it is assigned a name $t\_n$ where $n$ is a fresh (not yet used) natural number for the given $T$.

For *parameterized types*, a hypothesis of type $T\ a\ b\ \ldots\ z$ is first transformed by recursively applying the naming method to all its parameterizing arguments. Then it is converted into a string by intercalating the type with underscores, yielding $T\_a\_b\ldots\_z$ and only then is a fresh natural number $n$ appended, along with a preceding _, forming $T\_a\_b\ldots\_z\_n$ (for potentially non-scalar a...z).

For *function types*, a hypothesis of type $a \rightarrow b$ is modified by recursively applying the naming method to its components, then it is converted into a string by intercalating the resulting types with underscores, forming $a\_b$. Then the word ”func” along with an underscore is prefixed and an another underscore with a fresh natural number $n$ is suffixed, yielding the final form *func_a_b_n*.

**Language definitions**

We have already introduced (by example) both the specification definition language and the function declaration language. Their grammars (EBNF, tokens with capital letters are non-terminals, other tokens are terminals, potentially enclosed in apostrophes for disambiguation, assumed lexemes are enclosed in angled brackets) are as follows:

**Specifications**

```
CONCEPT   ::= <identifier>
IDENT     ::= <identifier>
TPARAM    ::= IDENT | '(' FUNCTYPE ')' | '(' DATATYPE ')'
TPARAMS   ::= TPARAM+
DATATYPE  ::= IDENT [TPARAMS]
FUNCTYPE  ::= TYPE -> TYPE (-> TYPE)*
TYPE      ::= DATATYPE | '(' FUNCTYPE ')'
KPARAMS   ::= CONCEPT | '(' CONCEPT (, CONCEPT)* ')
KIND      ::= KPARAMS IDENT
PARAMS    ::= parameterization { [KIND (, KIND)*] }
OPTYPE    ::= : FUNCTYPE
OP        ::= IDENT [OPTYPE]
OPS       ::= operations { OP (, OP)* }
SPEC      ::= Specification IDENT { [PARAMS] OPS }
```

**Declarations**

```
CONCEPT   ::= <identifier>
IDENT     ::= <identifier>
DATATYPE  ::= IDENT [TPARAMS]
```

```
FUNCTYPE  ::= TYPE -> TYPE (-> TYPE)*
TYPE      ::= DATATYPE | '(' FUNCTYPE ')'
KIND      ::= CONCEPT IDENT
KLIST     ::= [KIND (, KIND)*]
KPARAMS   ::= kind | '(' [KLIST] ')'
KINDS     ::= KPARAMS =>
SIGNATURE ::= IDENT : FUNCTYPE
OPERATION ::= [KINDS] SIGNATURE
```

## 2.3 From proofs to executable programs

Having decided on the way the analysis shall be conducted, we need an approach to transforming its result to an executable implementation.

### 2.3.1 Direct correspondence between analytical primitives and a language (Haskell)

We believe that the most elegant approach is to associate ”generative primitives” with the analytical atoms we use. This way, an implementation is being synthesized as the analysis is being conducted.

Let us take a look a look at a familiar simple example. Please do note that this section is focused on the generative part and as such will skip many a step of the analytical process, which is in detail described in the preceding chapter.

Given the analytical string for `id :  a` $\rightarrow$ `a` consisting of three atoms:

```
intro<a_0> trivial<a_0> clear<a_0>
```

We transform each of the atoms directly to its corresponding primitive in the target language. In our case, Haskell. Of course, this choice is mostly arbitrary even though a functional language has a more direct mapping; more on the topic later.

```
intro<a_0>      as \a_0 ->
trivial<a_0>    as a_0
clear<a_0>      as <nothing>
```

Thus yielding the resulting generation string:

```
\a_0 -> a_0
```

Adding the name of the proposition `id` with an equal sign representing a definition in Haskell, we get:

```
id = \a_0 -> a_0
```

Which indeed does happen to be a correct definition of the identity function in Haskell. As a matter of fact, it is "the" correct definition, as it is the *only* function valid for the signature if we consider a total function.

Note: if we were to not consider totality, we could for example, create a definition that would type check as follows: `id = id . id` - halting problem is somewhat of a hindrance sometimes (there is an infinite amount of cyclic functions for the signature in question)

Let us take a look at Haskell's generative counterparts to analytical primitives in the following table:

| **Tactic** | **Analytical atom** | **Generative atom** |
|---|---|---|
| *Trivial* | trivial<h> | h |
| *Intro* | intro<h> | \h -> |
| Intros | intros[intro<h0>... intro<hn>] | N / A (compound) |
| Clear | clear<h> | N / A (non-generative) |
| *ElimSum* | elimSum<h>(c0 ... cn) | case h of c0 ... cn -> |
| *Apply* | apply<h>(0...n) | h 0 1 ... n |

**Table 2 – 1** Analytical atoms and their generative counterparts

The generative part is language specific and can be specified for other languages as well. In order for a new language to be introduced, it is almost enough to provide a corresponding generative atom for *Trivial*, *Intro*, *ElimSum* and *Apply*. If we wanted full "integration", we would also need to add syntax for various utilities such as type introduction (:: in Haskell) or equation introduction (= in Haskell).

We have already seen *intro* and *trivial* in a generative example. Let us briefly take a look at the other tactics as well.

Given `f : a → (a → b) → b` and its analytical string:

```
intro<a_0> intro<func_a_b_0> apply<func_a_b_0>(1/1)
   clear<func_a_b_0> trivial<a_0> clear<a_0>
```

we can transcribe it to:

```
intro<a_0>         as \a_0 ->
intro<func_a_b_0> as \func_a_b_0 ->
apply<func_a_b_0> as func_a_b_0
```

```
trivial<a_0>      as a_0
```

yielding:

```
\a_0 -> \func_a_b_0 -> func_a_b_0 a_0
```

This is a valid Haskell definition that simply applies the given function `a` → `b` to the provided argument `a`.

The last remaining tactic to take a look at is the *elimSum*. Consider `first : Pair a b` → `a` with its analytical string:

```
intro<pair_a_b_0> elim<pair_a_b_0>(Make) clear<pair_a_b_0>
   intro<a_0> intro<b_0> trivial<a_0> clear<a_0>
```

with the `Pair` type being defined as:

```
Specification Pair {
    parameterization {
        Any a,
        Any b
    }
    operations {
        Make : a -> b
    }
}
```

Transformation to Haskell is then as follows:

```
intro<pair_a_b_0>      as \pair_a_b_0 ->
elim<pair_a_b_0>(Make) as case pair_a_b_0 of Make ->
```

```
intro<a_0>              as \a_0 ->
intro<b_0>              as \b_0 ->
trivial<a_0>            as a_0
```

Yielding:

```
\pair_a_b_0 -> case pair_a_b_0 of Make -> a_0 -> b_0 -> a_0
```

There is however a small problem with this particular generation. Haskell does not allow for partially applied data constructors such as `Make` → `a` → `....` This is of course fixed within the implementation provided with the thesis but it is worth pointing out the deficiency. This problem takes away from the elegance of the direct transformation but is on the other hand easily fixed within an actual implementation.

Therefore the *actual* generation string (with fully applied `Make`) is:

```
\pair_a_b_0 -> case pair_a_b_0 of Make a_0 b_0 -> a_0
```

An another deficiency of this approach is related to its context oblivious nature. Whenever we use apply, we should parenthesize the enclosing applied function. Consider the following arbitrary example:

```
foo : h -> (h -> i) -> x -> (i -> x -> w) -> w
```

If we were to directly transcribe its analysis strings:

```
intro<h_0> intro<func_h_i_0> intro<x_0> intro<func_i_x_w_0>
   apply<func_i_x_w_0>(1/2) clear<func_i_x_w_0>
   apply<func_h_i_0>(1/1) clear<func_h_i_0> trivial<h_0>
   clear<h_0>
```

```
intro<h_0> intro<func_h_i_0> intro<x_0> intro<func_i_x_w_0>
   apply<func_i_x_w_0>(2/2) clear<func_i_x_w_0> trivial<x_0>
   clear<x_0>
```

We would get:

```
\h_0 -> \func_h_i_0 -> \x_0 -> \func_i_x_w_0 -> func_i_x_w_0
   func_h_i_0 h_0 x_0
```

This attempts to apply three arguments (namely `func_h_i_0, h_0` and `x_0`) to a binary function `func_i_x_w_0`. We therefore need to somewhat artificially enclose every inner function application in parentheses. This is, of course, solved in the provided implementation. The correct generated string (along with the function name prefixed) is ultimately:

```
foo = \h_0 -> \func_h_i_0 -> \x_0 -> \func_i_x_w_0 ->
   func_i_x_w_0 (func_h_i_0 h_0) x_0
```

The last problem is also related to handling contexts. When the *elimSum* introduces more than a single constructor, a fresh equation is generated for every one of them. We therefore need to trim common prefixes in order to end up with a single equation pattern matching in the *case of* construct. We have already introduced the `List a` data type, so consider the following:

```
random : List a -> c -> c
```

The analysis:

```
intro<list_a_0> elim<list_a_0>(Nil) clear<list_a_0>
   intro<c_0> trivial<c_0> clear<c_0>
```

```
intro<list_a_0> elim<list_a_0>(Cons) clear<list_a_0>
   intro<a_0> intro<list_a_1> intro<c_0> trivial<c_0>
   clear<c_0>
```

Direct transformation incorrect for the aforementioned reason:

```
\list_a_0 -> case list_a_0 of Nil -> \c_0 -> c_0
\list_a_0 -> case list_a_0 of Cons a_0 list_a_1 -> \c_0 ->
   c_0
```

After common prefixes are trimmed, we end up with a correct definition (also including the entire equation):

```
random = \list_a_0 ->
    case list_a_0 of Nil -> \c_0 -> c_0
                     Cons a_0 list_a_1 -> \c_0 -> c_0
```

It is important to note that whatever user defined type is used in analysis needs to be also manually defined in target language.

### 2.3.2 Further transformations to imperative languages, introducing Ren

As we can see, the mapping for functional languages is very straightforward. They are ”append only” in their nature, meaning that simple concatenation of atoms more often than not leads directly to a valid sentence of the language.

On the other hand, imperative languages often come equipped with syntax that is in some sense ”more context free” (excuse the carefree usage of the term); that is, they often require for example matching sets of parentheses to introduce language constructs.

Consider (again) the omnipresent `id : a → a` example. In C++, we would ideally want something along the lines of:

```
template <typename T>
T const& id(T const& a) {
    return a;
}
```

However, we do not want to go too much into detail with C++ as a language and therefore we first conveniently forget that passing by value requires semi-regular types and we get the following definition:

```
template <typename T>
T id(T a) {
    return a;
}
```

Also, logic is not something that the C++ standardization committee is too fond of and as it turns out, exclusively with lambdas can we indeed omit the *template* syntax for introducing type parameters, therefore finally going for the following definition:

```
auto id = [](auto a) { return a; };
```

Having the analytical string:

```
intro<a_0> trivial<a_0> clear<a_0>
```

Clearly the intro<a_0> should be `auto a_0`, trivial<a_0> should be `return a_0` and clear<a_0> is non-generative as we have stated before. Equation introduction would then be `= [](` and name transformation could easily be synthesized as `f : (name : String)` → `String` with `f name = "auto" + " " + name + " "`.

It is however immediately obvious that we would need to somehow handle parentheses balancing, proper comma separation of arguments, context-aware generation with trivial sometimes being with the preceding *return* keyword and sometimes without, namely in function contexts.

We could of course handle all of this with either modifying the inference calculus to be context-aware or simply fixing the implementation itself. After all, the conducted analysis contains all the information we need; but then again, this is certainly not as elegant as having very direct transformations between fragments of the analysis and the generation.

The last option would be to adjust the target imperative language in a way such that it would syntactically (and to some extent semantically) resemble a declarative functional ”append-only” programming language and only then proceed to generate an implementation.

As a small detour, we provide a mini quasi-functional language-within-a-language for C++ named Ren to demonstrated how this would be done. We shall not go into too much detail with it considering the nature of the thesis and its focus on functional languages, or mostly Haskell. There is also no generative back-end from the analytical part implemented so while it could be used as a reasonable way to synthesize C++ from the analysis we conduct, we do not have an implementation for this connection.

Consider the following:

```
auto sumAllOdds = c_foldl < std::plus<int>{} < 0 |= c_filter
    < isOdd;
```

This is completely valid C++ with some liberal use of operator overloads and template metaprogramming. Its alternative in Haskell would be:

```
sumAllOdds = foldl (+) 0 . filter odd
```

Do note that for example arguments for c_foldl (which happens to represent the reduce operation) are provided in such way so that they can be simply appended to the function name with the preceding $<$ symbol. Also all left hand sides are of form $auto\, f =$. This "kind" of C++ which is an imperative language would be very easy to target with our framework. Ren provides seamless integration of curried functions, partial application, the ability to compose functions with convenient syntax and some functional primitives one would normally find in a functional language such as map or filter. An interested reader may inspect the enclosed source code of Ren for further reference with implementation and a few examples of its use.

Note: Ren is undocumented as it is an addendum to the thesis serving illustrative purposes beyond its scope. A very recent C++ compiler is required to compile the project. The recommended way to look at its functionality is to paste the contents of "/src/Ren/ren.cpp" into http://coliru.stacked-crooked.com/

# 3 Generating (some of the) Haskell standard "Prelude" and more

Now that we have a way of analysing signatures and also transforming results of the analysis into an executable implementation, let us take a look at some practical applications. The "Prelude" [3] module of Haskell is automatically imported unless specified otherwise to every Haskell project and forms the basis of notion of "standard library"; it is a convenient place to look at for some practical examples.

Skipping instance declarations and numeric functions that cannot be generated as they lack parametricity, we arrive at the *id* function that has been (ab)used enough already and is therefore skipped.

## 3.1 Const

The *const* function ignores its second argument whatever that may be and always returns the first one. Stealing its type:

```
const : a -> b -> a
```

We get:

```
const :: (a -> (b -> a))
const = \a_0 -> \b_0 -> a_0
```

Which is exactly what we want.

## 3.2 (.) also known as compose

The dot operator composes two functions. Taking its type from Prelude and conveniently renaming it to compose:

```
compose : (b -> c) -> (a -> b) -> a -> c
```

We get:

```
compose :: ((b -> c) -> ((a -> b) -> (a -> c)))
compose = \func_b_c_0 -> \func_a_b_0 -> \a_0 -> func_b_c_0
   (func_a_b_0 a_0)
```

This definition happens to correspond to the one provided in the official standard, modulo naming and some syntactic sugar.

## 3.3 Flip

Flip takes a function f and two arguments which are then applied to f in reverse order. Its type:

```
flip : (a -> b -> c) -> b -> a -> c
```

And the resulting Haskell implementation:

```
flip :: ((a -> (b -> c)) -> (b -> (a -> c)))
flip = \func_a_b_c_0 -> \b_0 -> \a_0 -> func_a_b_c_0 a_0 b_0
```

## 3.4 Seq

Seq introduces strictness into Haskell. It takes two arguments, returns the second one and is strict in its first argument. Generating it makes no practical sense whatsoever as we cannot enforce strictness, but we *can* nonetheless. Given its type:

```
seq :: a -> b -> b
```

We get:

```
seq :: (a -> (b -> b))
seq = \a_0 -> \b_0 -> b_0
```

Which does not really do much because its type does not say a whole lot about what this function is supposed to do, but we do have something; a function that returns its second argument and ignores the first.

## 3.5 ($) also known as apply

The ($) operator (conveniently renamed to *apply*) is infix function application with low priority and right associativity. Its type unfortunately does not contain the aforementioned information, but we can in fact synthesize a function that at least applies whatever argument it is given to the function it is given. Its type:

```
apply : (a -> b) -> a -> b
```

The synthesized function:

```
apply :: ((a -> b) -> (a -> b))
apply = \func_a_b_0 -> func_a_b_0
```

Does what it advertises, but is completely useless because it lacks its implicit infix nature and associativity.

Note: we now skip Bool and Char declarations and corresponding instance declarations, we cannot do anything about those; consider `f : Bool → Bool → Bool` - the `f` function can either be ”and” or ”or” and we have no way of figuring it out; again, lack of parametricity

## 3.6 Maybe with its corresponding maybe

Maybe is a data type often used to represent a computation that may fail. Its definition in our language:

```
Specification Maybe {
    parameterization {
        Any a
    }
    operations {
        Nothing,
        Just : a
    }
}
```

Prelude now defines a function maybe that applies any `f` in case the value had been constructed from its `Just` data constructor; otherwise it returns a user-provided default. Its type:

```
maybe : b -> (a -> b) -> Maybe a -> b
```

Gives us:

```
maybe :: (b -> ((a -> b) -> (Maybe a -> b)))
maybe = \b_0 -> \func_a_b_0 -> \maybe_a_0 -> b_0
```

This of course is not quite what we wanted, as the default value is returned even if the `Maybe a` is of shape `Just a`. This is the result of a relatively eager-to-prove heuristic that attempts to invoke *trivial* whenever possible.

Note: skipping more instance declarations as we are going over the entire Prelude [3]

## 3.7 Fst

Fst returns the first part of a pair (a tuple of size two). Pair has been defined in the previous chapter. The signature of fst rewritten in terms of our definition:

```
fst : Pair a b -> a
```

Yields:

```
fst :: (Pair a b -> a)
fst = \pair_a_b_0 -> case pair_a_b_0 of Make a_0 b_0 -> a_0
```

Which indeed works as intended.

## 3.8 Snd

Similarly to fst, snd returns the second part of a pair. Given its type using our definition of pair:

```
snd : Pair a b -> b
```

Results in the following definition:

```
snd :: (Pair a b -> b)
snd = \pair_a_b_0 -> case pair_a_b_0 of Make a_0 b_0 -> b_0
```

This is indeed the functionality we are looking for.

## 3.9 Curry

Curry converts an uncurried function to a curried function. Its type using our own pair:

```
curry : (Pair a b -> c) -> a -> b -> c
```

Synthesizes the following:

```
curry :: ((Pair a b -> c) -> (a -> (b -> c)))
curry = \func_pair_a_b_c_0 -> \a_0 -> \b_0 ->
   func_pair_a_b_c_0 (Make a_0 b_0)
```

Which is what we were looking for.

## 3.10 Uncurry

Uncurry converts a curried function into an uncurried one. It is of type:

```
uncurry :: ((a -> (b -> c)) -> (Pair a b -> c))
uncurry = \func_a_b_c_0 -> \pair_a_b_0 -> case pair_a_b_0 of
   Make a_0 b_0 -> func_a_b_c_0 a_0 b_0
```

Works as intended.

Of the remaining four functions in the main part of the Prelude, two are hacks for error handling, one contains Bool in its signature and the last one called `asTypeOf` is of type `a` $\rightarrow$ `a` $\rightarrow$ `a`, for which our heuristic cannot decide which argument to return (the first a or the second a) and fails.

## 3.11 Miscellaneous

We can of course also generate implementations from completely arbitrary signatures. Consider:

```
arbitrary : List a -> List b -> c -> c
```

Which yields:

```
arbitrary :: (List a -> (List b -> (c -> c)))
arbitrary = \list_a_0 -> case list_a_0 of Nil ->
    \list_b_0 -> case list_b_0 of Nil -> \c_0 -> c_0
                                  Cons b_0 list_b_1 -> \c_0
                                     -> c_0
                                          Cons a_0 list_a_1
                                             ->
    \list_b_0 -> case list_b_0 of Nil -> \c_0 -> c_0
                                  Cons b_0 list_b_1 -> \c_0
                                     -> c_0
```

or something along the lines of:

```
arbitrary : h -> j -> (h -> j -> i) -> x -> (i -> y -> x ->
   z -> w) -> y -> z -> w
```

Yielding:

```
arbitrary :: (h -> (j -> ((h -> (j -> i)) -> (x -> ((i -> (y
   -> (x -> (z -> w)))) -> (y -> (z -> w)))))))
arbitrary =
    \h_0 -> \j_0 -> \func_h_j_i_0 -> \x_0 ->
    \func_i_y_x_z_w_0 -> \y_0 -> \z_0 ->
        func_i_y_x_z_w_0 (func_h_j_i_0 h_0 j_0) y_0 x_0 z_0
```

Naturally, we can perhaps even make something useful for a change:

```
swap : Pair a b -> Pair b a
```

Gets us:

```
swap :: (Pair a b -> Pair b a)
swap = \pair_a_b_0 -> case pair_a_b_0 of Make a_0 b_0 ->
   Make b_0 a_0
```

```
repl : x -> Pair x x
-------------------------
repl :: (x -> Pair x x)
repl = \x_0 -> Make x_0 x_0
```

```
firsts : Pair a b -> Pair c d -> Pair a c
-----------------------------------------
firsts :: (Pair afirsts :: (Pair a b -> (Pair c d -> Pair a
   c))
firsts = \pair_a_b_0 -> case pair_a_b_0 of Make a_0 b_0 ->
   \pair_c_d_0 -> case pair_c_d_0 of Make c_0 d_0 -> Make
   a_0 c_0
```

Broken map:

```
map : (a -> b) -> List a -> List b
----------------------------------
```

```
map :: ((a -> b) -> (List a -> List b))
map = \func_a_b_0 -> \list_a_0 ->
    case list_a_0 of Nil -> Nil
                     Cons a_0 list_a_1 -> Nil
```

As we have stated before, `Nil` is perfectly valid in the `Cons` case without dependent types.

We also support basic backtracking, therefore both of the following work:

```
backtrack : c -> (a -> b) -> (c -> b) -> b
------------------------------------------
backtrack :: (c -> ((a -> b) -> ((c -> b) -> b)))
backtrack = \c_0 -> \func_a_b_0 -> \func_c_b_0 -> func_c_b_0
   c_0
```

`Backtrack` works as expected as arguments closer to the return type cost less. However, with only very basic backtracking in place, we can also make the following work:

```
trackback : a -> (a -> b) -> (c -> b) -> b
------------------------------------------
trackback :: (a -> ((a -> b) -> ((c -> b) -> b)))
trackback = \a_0 -> \func_a_b_0 -> \func_c_b_0 -> func_a_b_0
   a_0
```

If we cannot find a proof with applying the `c` $\rightarrow$ `b` function, we trace our steps back and try the other one.

## 3.12 Summary

In general, our heuristics within the calculus perform a non-exhaustive search of some sort. This means that they sometimes do find a solution and sometimes they miss it, but it is difficult to classify the set of solutions that can be found using this method. That said, using this approach we can synthesize implementations for functions used ubiquitously in Haskell (among other things), which shows the merit of the method. Informally, functions that end up being fully generated must typecheck by construction and hence provide a viable definition of the signature. It is impossible to do further analysis on whether it does what was ”intended”. When it comes to signatures that can be synthesized, the following general criteria must be met:

- purely parametric signature
- totality
- unambiguous type occurrences (consider $a \to a \to a$, which a do we return?)
- hints to disambiguate data constructors (as all of those ultimately return the type they are constructing, naturally resulting in ambiguity)

# 4 What should be possible and grounds for further research, our implementation and its limitations

As we have mentioned before, we do not support dependent types. However, with the ability for types to depend on terms (values), we could generate implementations for a whole new class of functions.

## 4.1 Generating a correct map with a dependently typed system

First consider the signature of *map* without dependent types, using Idris syntax:

```
map : (f : (a -> b)) -> (xs : List a) -> List b
```

Using SolveByIntroduction, after introducing both the function f and the xs list (corresponding with intros), we get the following:

$$f : (a \to b),\ xs : List\ a\ ,\ \Gamma\ \vdash\ List\ b$$

Deconstructing xs (as though elimSum was used) yields two new environments. For Nil:

$$f : (a \to b),\ xs : List\ a,\ \Gamma\ \vdash\ List\ b$$

For Cons:

$$f : (a \to b),\ xs : List\ a,\ x : a,\ ys : List\ a\ ,\ \Gamma\ \vdash\ List\ b$$

In the Nil environment, the only way to construct something of type *List b* is to use Nil (transitioning to Solve by Apply, as though apply with zero arguments was used), which solves the subgoal of the environment in question. Therefore we would have *intros[intro<f>, intro<xs>]*, *elimSum<xs>*, *apply<Nil>*, which we could then transform to:

```
map = \f -> \xs -> case xs of Nil -> Nil
```

Which is correct for the map implementation in case of Nil. However in the Cons case, we immediately encounter a problem: Nil is of type List b and it is a perfectly valid right hand side. This is, of course, incorrect. What we do want is:

```
map = \f -> \xs -> case xs of Cons x ys -> Cons (f x) (map f
    ys)
```

It is also important to note that there are also two hypothesis with type List a, which means that the trivial tactic would fail in the recursive call of Cons, if the Nil type match had not been a problem in first place; this could however be solved by immediately clearing the eliminated hypothesis.

As is immediately obvious, it would be very difficult to devise a general heuristic to apply functions in correct order to handle these subtle ambiguities. However, if we could encode more information in the signature itself, we would be able to disambiguate in many more cases. Consider the following definition of *List* in Idris, which also encodes its size in its type.

```
data List : Nat -> Type -> Type where
    Nil : List Z a
    Cons : a -> List k a -> List (S k) a
```

Now for the signature of map:

```
map : (f : (a -> b)) -> (xs : List k a) -> List k b
```

What we are saying is that the resulting list needs to be exactly as long as the input list xs. If we were to now assert all the arguments (beginning with Solve by Introduction), we would get:

$$f : (a \to b),\ xs : List\ k\ a\ ,\ \Gamma\ \vdash\ List\ k\ b$$

Deconstructing *xs* yields two new environments. For Nil:

$$f : (a \to b),\ xs : List\ (S\ k)\ a\ ,\ \Gamma\ \vdash\ List\ Z\ b$$

Here we see a difference already. We indicate that the resulting list needs to be of length zero and as such, the Nil constructor is the only way to create a definition that typechecks. For Cons:

$$f : (a \to b),\ xs : List\ (S\ k)\ a,\ x : a,\ ys : List\ k\ a\ ,\ \Gamma\ \vdash\ List\ (S\ k)\ b$$

We can now see, however, that Nil is no longer a valid right hand side, as k $\neq$ Z. If we also immediately cleared the deconstructed xs, we would be left with:

$$f : (a \to b),\ x : a,\ ys : List\ k\ a\ ,\ \Gamma\ \vdash\ List\ (S\ k)\ b$$

The only way to get *List k b* if we do not have Nil is to use Cons, which generates two further environments. For its first argument, we have a hole of type that corresponds with the first argument of the Cons data constructor, b (after unification a $\leftarrow$ b). Therefore:

$$f : (a \to b),\ x : a,\ ys : List\ k\ a\ ,\ \Gamma\ \vdash\ b$$

The only function in the context that has return type of *b* is *f*, applying it generates a new goal with the same context (without *f* if immediately cleared) and with a goal of type *a*:

$$x : a,\ ys : List\ k\ a\ ,\ \Gamma\ \vdash\ a$$

Now the goal a can be solved, as $x :\ a$ is already available.

The second argument of Cons is a list of size k, parameterized by b. Therefore the other environment generated by deconstructing *xs* is:

$$f : (a \to b),\ x : a,\ ys : List\ k\ a\ ,\ \Gamma\ \vdash\ List\ k\ b$$

The only function with return type unifiable with the current subgoal is *map* (its recursive invocation), applying it creates two more environments. The first argument is of type $a \to b$, therefore the new environment:

$$f : (a \to b),\ x : a,\ ys : List\ k\ a\ ,\ \Gamma\ \vdash\ (a \to b)$$

This one is trivial as *f* is the only thing that matches. The second argument of *map* is of type *List k a*, yielding the following environment:

$$f : (a \to b),\ x : a,\ ys : List\ k\ a\ ,\ \Gamma\ \vdash\ List\ k\ a$$

This one can be immediately completed by using *ys* from the context.

If we were to reconstruct the analysis, we would get something along the lines of:

```
intros[intro<f>, intro<xs>] elimSum<xs>(Nil) clear<xs>
   apply<Nil>(1/1) clear<Nil>

intros[intro<f>, intro<xs>] elimSum<xs>(Cons) clear<xs>
   apply<Cons>(1/2) intro<x> intro<ys> apply<f>(1/1)
   trivial<x> clear<x>

intros[intro<f>, intro<xs>] elimSum<xs>(Cons) clear<xs>
   apply<Cons>(2/2) intro<x> intro<ys> apply<map>(1/2)
   trivial<f> clear<f>

intros[intro<f>, intro<xs>] elimSum<xs>(Cons) clear<xs>
   apply<Cons>(2/2) intro<x> intro<ys> apply<map>(2/2)
   trivial<ys> clear<ys>
```

Which could be then transformed to (using Idris syntax):

```
map : (a -> b) -> List k a -> List k b
map = \f => \xs => case xs of Nil => Nil
                              Cons x ys => Cons (f x) (map f
                                 ys)
```

This is indeed a correct implementation of map. As we have stated before, we would need to implement a dependently typed system for this to work with the heuristics we have devised.

## 4.2 Limitations

Our prototype has various bugs at its disposal, as well as missing features and general imperfections. First and foremost, dependent types are not available.

Secondly, only one user defined type definition per analysis is allowed. For example it is impossible to have a function containing a `List` and a `Pair`.

Moreover, data constructors must contain only a single capital letter; namely the initial one.

Also, we do not import additional operations from type restrictions. As a matter of fact, there is no way to define them either. Therefore the *any* constraint (which is equivalent to no constraints) is the only one available.

No guarantees are made about what happens when an implementation cannot be synthesized. This manifests in several ways that include (but are not limited to) not halting, crashing, throwing an exception that is uncaught and generating a malformed implementation.

# 5 Discussion

## 5.1 Decisions and their impact

Both arbitrary and well thought out decisions have been made along the way, with various levels of impact on the entire process of analysis and generation. Said assumptions are discussed in the chapter ”Preliminaries and assumptions”, along with the reasoning about why they had been made in first place. Their influence on the resulting process of analysis and synthesis varies.

Functional paradigm is first and foremost different semantically, at least in terms of preferred way of solving problems, which is in some sense more aligned with mathematical logic. However, we also take advantage of the fact that in a way, their syntax to some extent reflects their declarative nature and is therefore easier to be synthesised from our analysis. As a matter of fact, it is so easy that in most scenarios, our tactics can be directly transcribed to an executable form, regardless of context they are in.

Purity is on the other hand important for the analysis phase, where we had the freedom to operate under the assumption that each function signature ”states” what it does and no other functionality was at all possible. This allows us to have some confidence that what we synthesize does in fact do what we originally wanted. This concept is also reasonably close to termination, where assuming totality allows us to avoid malformed definitions that typecheck.

There is no doubt that a more expressive type system, such as one with dependent types, would allows us to do a lot more, as we showed with *map*. As a matter of fact, our general approach does work with such system, we just have not provided an implementation thereof, as it would require more complex unification and normalization algorithms, as well as additional complexities in occurs checks and in the notion of equality, with regards to normalization of terms and types.

## 5.2 Further considerations and research

### 5.2.1 Using an existing system

An interesting approach would be to take an existing language with dependent types such as Idris and utilize its unification and other parts integral for our purposes. This would give us a framework to work with, allowing us to focus on creating more diverse heuristics, potentially with more advanced backtracking and various cost functions, which help specify order of applied tactics. We could also utilize tooling that has been created for Idris and programmers could use syntax they are familiar with to provide their specifications and declarations; this would then also remove the need to manually enter type declarations. Moreover, having a very expressive type system implemented would also allow us to ”cheat” with implementation synthesis; we could simply search in a shared database of definitions written by other users. Given that functionality is very well specified by the type itself, we assume that type directed search would be of great merit. As an interesting observation to consider, our framework is basically a programming language that does not allow definitions to be entered, only declarations (be it for types or functions).

### 5.2.2 Our framework as a proof assistant

It would be possible to turn our implementation into a proof assistant very easily. We would simply drop the part that does automatic solving and prompt users to manually enter tactics and compute environment changes based on the tactics they would provide. That said, our thesis is unique in its approach and this step is an integral part of the idea behind it. Not to mention that powerful proof assistants already exist and ours would pale in comparison.

# 6 Conclusion

We set out to find a way to automatically convert a function signature into an executable implementation. We chose an approach that utilizes the Curry-Howard isomorphism between types and propositions. Having taken advantage of the correspondence, we created a calculus with operations that manipulate contexts, allowing us to decompose a proposition and construct its proof. We then developed a set of heuristic algorithms to apply said operations in order to automate the process of proof construction. Then we devised a way of transforming proofs to executable implementations in functional languages and also outlined an approach that allows us to perform further transformations to imperative paradigm. As an interesting example of modifying a non-functional language into one that offers functional syntax, we created a toy embedded domain specific language of sorts called Ren, which is just as easy to target as any other functional language. Finally we created a set of concrete transformations to Haskell and implemented the entire functionality.

# References

[1] Brady, E.: Idris Mode for Atom.[online]. [accessed 29.1.2016]. <https://github.com/idris-hackers/atom-language-idris.

[2] Brady, E.: Ivor, a Proof Engine.[online]. [accessed 29.1.2016]. <https://eb.host.cs.st-andrews.ac.uk/writings/ivor.pdf>.

[3] Jones, S. P.: The Haskell 98 Report[online]. [accessed 26.3.2016]. <https://www.haskell.org/onlinereport/standard-prelude.html>.

[4] Marlow S., Waern D.: Haddock Documentation.[online]. [accessed 11.12.2015]. <https://www.haskell.org/haddock/>.

[5] Meijer, E. and Drayton, P.: Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages.[online]. [accessed 12.12.2015]. <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.394.3818&rep=rep1&type=pdf>.

[6] Unknown: Cofoja framework.[online]. [accessed 12.12.2015]. <https://github.com/nhatminhle/cofoja>.

[7] Wadler, P.: Theorems for free![online]. [accessed 2.2.2016]. <http://ttic.uchicago.edu/~dreyer/course/papers/wadler.pdf>.

[8] Xi, H.: A Dependently Typed Assembly Language.[online]. [accessed 11.12.2015]. <http://www.cs.bu.edu/~hwxi/academic/papers/DTAL.pdf>.

[9] Aho, A. V.; Sethi, R.; Ullman, J. D.: *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986, ISBN 0-201-10088-6.

[10] Alexandrescu, A.: *The D Programming Language*. Addison-Wesley Professional, first edition, 2010, ISBN 0321635361, 9780321635365.

[11] Ball, T.; Rajamani, S. K.: The SLAM Project: Debugging System Software via Static Analysis. *SIGPLAN Not.*, volume 37, nr. 1, January 2002: p. 1–3, ISSN 0362-1340.

[12] Barendregt, H. P.: Handbook of Logic in Computer Science (Vol. 2). New York, NY, USA: Oxford University Press, Inc., 1992, ISBN 0-19-853761-1, p. 117–309.

[13] Barras, B.; Boutin, S.; Cornes, C.; aj.: The Coq Proof Assistant Reference Manual : Version 6.1. Research Report RT-0203, INRIA, May 1997, projet COQ.

[14] Bennett, J. P.: *Introduction to Compiling Techniques: A First Course Using ANSI C, LEX and YACC*. New York, NY, USA: McGraw-Hill, Inc., 1996, ISBN 007709221X.

[15] Beydeda, S.; Book, M.; Gruhn, V.; aj.: *Model-driven software development*, volume 15. Springer, 2005.

[16] Brady, E. C.: IDRIS —: Systems Programming Meets Full Dependent Types. In *Proceedings of the 5th ACM Workshop on Programming Languages Meets Program Verification*, PLPV '11, New York, NY, USA: ACM, 2011, ISBN 978-1-4503-0487-0, p. 43–54.

[17] Buss, S. R.: *Handbook of Proof Theory*, chapter An Introduction to Proof Theory. Elsevier, Amsterdam, 1998, p. 1–79.

[18] Cook, W. R.; Hill, W.; Canning, P. S.: Inheritance is Not Subtyping. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, New York, NY, USA: ACM, 1990, ISBN 0-89791-343-4, p. 125–135.

[19] Coquand, T.; Huet, G.: The calculus of constructions. *Information and Computation*, volume 76, nr. 2, 1988: p. 95 – 120, ISSN 0890-5401.

[20] Dyckhoff, R.: Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, volume 57, 9 1992: p. 795–807, ISSN 1943-5886.

[21] Finifter, M.; Mettler, A.; Sastry, N.; aj.: Verifiable functional purity in java. In *Proceedings of the 15th ACM conference on Computer and communications security*, ACM, 2008, p. 161–174.

[22] Girard, J.-Y.: The system F of variable types, fifteen years later. *Theoretical Computer Science*, volume 45, 1986: p. 159 – 192, ISSN 0304-3975.

[23] Ketonen, J.; Weyhrauch, R.: A decidable fragment of predicate calculus. *Theoretical Computer Science*, volume 32, nr. 3, 1984: p. 297 – 307, ISSN 0304-3975.

[24] McKinna, J.; Pollack, R.: *Typed Lambda Calculi and Applications: International Conference on Typed Lambda Calculi and Applications TLCA '93 March, 16–18, 1993, Utrech, The Netherlands Proceedings*, chapter Pure type systems formalized. Springer Berlin Heidelberg, 1993, ISBN 978-3-540-47586-6, p. 289–305.

[25] Meyer, B.: Applying 'design by contract'. *Computer*, volume 25, nr. 10, Oct 1992: p. 40–51, ISSN 0018-9162.

[26] Milner, R.: A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, volume 17, nr. 3, 1978: p. 348 – 375, ISSN 0022-0000.

[27] Nipkow, T.; Paulson, L. C.; Wenzel, M.: *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[28] Pierce, B. C.: *Types and Programming Languages*. Cambridge, MA, USA: MIT Press, 2002, ISBN 0-262-16209-1.

[29] Pierce, B. C.; Turner, D. N.: Local Type Inference. *ACM Trans. Program. Lang. Syst.*, 2000: p. 1–44, ISSN 0164-0925, doi:10.1145/345099.345100.

[30] Reynolds, J. C.: Types, Abstraction and Parametric Polymorphism. In *IFIP Congress*, 1983, p. 513–523.

[31] Sonnex, W.; Drossopoulou, S.; Eisenbach, S.: Zeno: An automated prover for properties of recursive data structures. In *TACAS*, Lecture Notes in Computer Science, March 2012.

[32] Tarditi, D.; Morrisett, G.; Cheng, P.; aj.: TIL: A Type-directed Optimizing Compiler for ML. *SIGPLAN Not.*, 1996: p. 181–192, ISSN 0362-1340.

[33] Tofte, M.: Type inference for polymorphic references. *Information and Computation*, volume 89, nr. 1, 1990: p. 1 – 34, ISSN 0890-5401.

[34] Tremblay, J.-P.; Sorenson, P. G.: *Theory and Practice of Compiler Writing*. New York, NY, USA: McGraw-Hill, Inc., 1985, ISBN 0070651612.

[35] Zhou, X.; Yan, L.; Lilius, J.: Function Inlining in Embedded Systems with Code Size Limitation. In *Embedded Software and Systems*, editation Y.-H. Lee; H.-N. Kim; J. Kim; Y. Park; L. Yang; S. Kim, Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, ISBN 978-3-540-72684-5, p. 154–161.

# Appendices

**Appendix A** User's Guide

**Appendix B** Technical Documentation

**Appendix C** a CD ROM containing the thesis in digital form as well as the generator and Ren

# Appendices

# A Appendix A

**User Guide - Implementation overview**

## A.1 System Requirements

- Processor: 1 gigahertz (GHz)
- RAM: 1 gigabyte (GB)
- Hard disk space: 64 megabytes (MB)
- Graphics card: N/A

## A.2 Software Requirements

- OS: Windows 10

## A.3 Installation

No installation required. An binary executable file for the Windows platform is provided, namely ”GenGen.exe”. Please see Technical Documentation for further instructions on how to build the software for a different platform.

## A.4 Usage

There are two input files required to launch ”GenGen.exe”, both of which need to be in a directory called ”input”, located in the same folder as the executable. The first one is to be named ”input.spec”. It contains a definition of a user defined type, using our specification language.

The other file must be named ”input.cppf” and contains a type signature of a function we want to analyse, described by our declaration language.

The program then reports:

- success (or failure) of syntactic analysis (whether the specified files follow the proper grammar)
- success (or failure) of semantic analysis (certain consistency rules, such as restriction of type variables only used in the signature)
- a set of analysis strings representing results of the analytical part of the program
- transcription of the analysis to a generation string
- a set of environments after all analysis has been finished
- the resulting implementation in Haskell

An example for *flip*. The file ”input.spec” is empty as no additional definitions are required. The file ”input.cppf” contains:

```
flip : (a -> b -> c) -> b -> a -> c
```

Running ”GenGen.exe” then gives us the following output:

```
Parsing succeeded - specification
Parsing succeeded - signature
Syntactic analysis passed.
Semantic analysis passed.
```

```
intro<func_a_b_c_0> intro<b_0> intro<a_0>
   apply<func_a_b_c_0>(1/2) clear<func_a_b_c_0> trivial<a_0>
   clear<a_0>
intro<func_a_b_c_0> intro<b_0> intro<a_0>
   apply<func_a_b_c_0>(2/2) clear<func_a_b_c_0> trivial<b_0>
   clear<b_0>

\func_a_b_c_0 -> \b_0 -> \a_0 -> func_a_b_c_0 a_0 b_0

flip : ((a -> (b -> c)) -> (b -> (a -> c)))
b_0  : b
-------------------------------------------



flip : ((a -> (b -> c)) -> (b -> (a -> c)))
a_0  : a
-------------------------------------------



flip :: ((a -> (b -> c)) -> (b -> (a -> c)))
flip = \func_a_b_c_0 -> \b_0 -> \a_0 -> func_a_b_c_0 a_0 b_0
```

Please do note that if a user defined algebraic data type is used, it needs to be manually defined in the target Haskell file as well.

# B Appendix B

## B.1 System Requirements

- Processor: 1 gigahertz (GHz)
- RAM: 1 gigabyte (GB)
- Hard disk space: 64 megabytes (MB)
- Graphics card: N/A

## B.2 Software Requirements

- OS: Windows 10

## B.3 Build, Compilation and Execution

The project is written in native C++. A binary file ”GenGen.exe” is provided for platforms running Windows 10. Should one want to target a different ecosystem, a build from sources is required.

### B.3.1 Compiler

The software has been tested to compile with GCC 4.9.3 (namely with g++). This or every more recent build of GCC should therefore be used to compile the project. No guarantees for other C++ compilers are provided albeit recent versions of Clang, Visual Studio and Intel C++ Compiler with appropriate flags should be able to build the target.

The flags required for compilation: -std=c++14 and link to C++ boost libraries flags

### B.3.2 Dependencies

The project depends only on C++ boost libraries, namely 1.59.0 or newer.

- Parsing is done by *boost::spirit*
- Sum types support for ease of implementation is from *boost::variant*
- Error handling is enhanced by *boost::optional*

The used libraries are header only.

### B.3.3 Binary

The generated executable requires a folder within the same directory, containing *input.spec* and *input.cppf*. Output is sent to the standard output channel.

## B.4 Project Structure

```
\---src
    \---analysis
        \---analysis.cpp
        \---analysis.hpp
        \---tactics.cpp
        \---tactics.hpp
    \---asts
        \---funcAst.hpp
        \---implAst.cpp
        \---implAst.hpp
        \---specAst.hpp
    \---internal
        \---fmapType.hpp
        \---instancedF.cpp
        \---instancedF.hpp
        \---wheels.hpp
        \---writerContextM.hpp
        \---writerContextM.cpp
    \---parsers
        \---funcParser.hpp
        \---funcParser.cpp
    \---semantics
        \---function.cpp
        \---function.hpp
        \---semanticConsistency.cpp
        \---semanticConsistency.hpp
        \---specification.cpp
        \---specification.hpp
    \---type_operations
```

```
        \---substituion.cpp
        \---substituion.hpp
        \---unification.cpp
        \---unification.hpp
    main.cpp
```

## B.5 ./asts

### B.5.1 Module funcAst

Module describing the abstract syntax tree unique to function definitions. Each node is represented by a structure, be it a simple aggregate or a *boost::variant*.

- **Kind** represents a type of type, with two *string* fields, representing a type class they belong to and a name
- **KList** is a *std::vector* of *kinds*
- **KParams** is a sum type (*boost::variant*) either representing a *kind* or optionally a list *KList*
- **Signature** represents a function type signature from an independent AST.
- **Operation** denotes an aggregate of *kinds* and a signature, forming a full operation

### B.5.2 Module specAst

Module describing the abstract syntax tree of the specification language as well as various utility functions.

- **KParams** represents kind parameters and is a discriminated union that is either a *string* or a *vector* of *strings* if there are multiple kind parameters present
- **Kind** is an aggregate of kind parameters and a kind name, forming a kind
- **Functype** denotes a function type, which is just a *vector* of *types*
- **TParam** is a recursive variant that is either a string representing a typename, or a *funcType* or a *dataType*
- **DataType** aggregates a name of a type and its type parameters
- **Type** is then a sum type that is either a *funcType* or a *dataType*
- **Op** represents an operation with name and type
- **Spec** is the final specification, aggregating the entire AST

We provide serialization functions for each of the nodes and the following self-explanatory utilities in this translation unit:

```
isFuncType : type const& -> bool
isDataType : type const& -> bool
```

### B.5.3 Module implAst

Module describing internal ASTs for implementations and various utilities. It contains a utility type **implemented** as a synonym for a pair of optional string and a node from any AST (templated by T), as well as an injection for it:

```
makeImplemented : boost::optional<std::string> const& impl
   -> T const& -> implemented
```

and two projections:

```
getImpl : implemented<specAstNode> const& ->
   boost::optional<std::string>
getTtpe : implemented<specAstNode> const& -> T
```

Then it contains a notion of type that is curried. It is encoded in **c**urriedType structure, which wraps *type* and adds the following utilities:

```
construct : type -> curriedType
```

Which curries an uncurried type.

```
deconstruct : curriedType -> std::pair<curriedType,
   boost::optional<curriedType>>
```

Which strips one argument off a curried function and returns the rest if it exists. Then free functions for further utility:

```
curry : type -> curriedType
uncurry : curriedType -> type
```

The module also contains an **environment** class which represents the main state of analysis in the program. It has names within its context as a map from string to unsigned, which keeps track of hypotheses names and a *vector* of *curriedTypes* which represents the available hypothesis in the context, as well as their type. The *environment* keeps track of uniqueness of names and as such provides:

```
addToContextWithDefaultName : curriedType const& ->
    std::string
```

which modifies its internal state as well as returns the most recently assigned name. The rest of the class just deals with basic operations on *vector*, such as adding and removing from the context. The last thing contained in this translation unit is notion of equality for types and their respective hashing.

## B.6 ./parsers

There are two languages present, each of them with a separate parser.

### B.6.1 Module funcParser

This module represents a parser for the definition language. Its grammar is in the main part of the thesis and its implementation with *boost::spirit::x3* mirrors it closely, namely as such:

```
auto const kind_def = spec::concept >> spec::ident;
auto const kList_def = kind % ", ";
auto const kParams_def = kind | ('(' >> -kList >> ')');
auto const kinds_def = kParams >> "=>";
auto const signature_def = spec::ident >> ':' >>
    spec::funcType;
auto const operation_def = -kinds >> signature;
```

### B.6.2 Module specParser

This module represents a parser for the specification language. This parser is synthesized from the following C++ code:

```
auto const concept_def = lexeme[+alnum];
auto const ident_def = lexeme[+alnum];
auto const tParam_def = ident | '(' >> funcType >> ')' | '('
   >> dataType >> ')';
auto const tParams_def = +tParam;
auto const dataType_def = ident >> -tParams;
auto const funcType_def = type >> "->" >> type % "->";
auto const type_def = dataType | '(' >> funcType >> ')';
auto const kParams_def = concept | '(' >> concept % ',' >>
   ')';
auto const kind_def = kParams >> ident;
auto const params_def = lit("parameterization") >> '{' >>
   kind % ',' >> '}';
auto const opType_def = ':' >> funcType;
auto const op_def = ident >> -opType;
auto const ops_def = lit("operations") >> '{' >> op % ',' >>
   '}';
auto const spec_def = lit("Specification") >> ident >> '{'
   >> -params >> ops >> '}';
```

## B.7 ./semantics

This folder aggregates translation units related to representations of internal hypotheses as they are built from input strings all the way to units ready for analysis.

### B.7.1 Module function

Module representing a function of a programming language. Contains a class **function** which contains name of a function, information about its parameterized types, its arguments and its return type. It serves as an intermediate representation resulting from the declaration AST transformation and as such is fairly trivial, with getters only for all of its aforementioned fields. The module also provides two self-explanatory utility functions, namely:

```
isParametric : std::string const& -> bool
isConcrete : std::string const& -> bool
```

### B.7.2 Module semanticConsistency

This modules only contains a single function. Its purpose is to ensure basic semantic consistency of the parsed declaration, namely between kind specifications and type variables. It has the following signature:

```
ensureConsistentConstraints : (typesBegin : Fwd_It) ->
   (typesEnd : Fwd_It) -> (constrainedNames :
   std::set<std::string>) -> void
```

Given a range of types and a range of constrained names, it checks whether any nullary kind constraints are present (ones that do not actually restrict any types) and also whether we are not restricting type variables that are unused.

### B.7.3 Module specification

Similar to the function module, this one is the result of transforming the specification AST into the resulting **specification** class. It is again just an intermediary and as such does not do much more than aggregate fields with very little additional logic. Its data members contain the name of the specification, what constructors are available and what the resulting type of construction is.

## B.8 ./internal

This directory contains utility functions, helper structures and general templates.

### B.8.1 Module wheels

The wheels module has general purpose functionality useful for implementation. (Explanation for self-explanatory functions omitted.)

```
rangeEmpty : (first : Any_It) -> (last : Any_It) -> bool
```

The intersperse function intersperses an element in a range.

```
intersperse : (first : Fwd_It) -> (last : Fwd_It) -> (dest :
   Out_It) -> (elem : typename
   std::iterator_traits<Out_It>::value_type const&) -> Out_It
```

Intercalate takes a range and intercalates it with an another range.

```
intercalate : (first : Fwd_It) -> (last : Fwd_It) -> (dest :
   Out_It) -> (elemFirst : Fwd_It2) -> (elemLast : Fwd_It2)
   -> Out_It
```

Intercalate stream is a specialized version for intercalate, working on streams.

```
intercalateStream : (first : Fwd_It) -> (last : Fwd_It) ->
    std::ostream& -> (what : std::string) -> std::ostream&
```

Flatten flattens nested ranges into a single one.

```
flatten : Outer const& -> std::vector<Inner>
```

```
head : (begin : Fwd_It) -> (end : Fwd_It) ->
    Fwd_It::value_type
tail : (begin : Fwd_It) -> (end : Fwd_It) ->
    std::vector<typename Fwd_It::value_type>
init : (begin : Fwd_It) -> (end : Fwd_It) ->
    std::vector<typename Fwd_It::value_type>
last : (begin : Fwd_It) -> (end : Fwd_It) ->
    Fwd_It::value_type
```

A helper structure creating a pair of any T and its name encoded as a string called **named**, along with its injection:

```
make_named : std::string const& -> T const& -> named<T>
```

And of course, two projections:

```
getName : named<T> const& -> std::string
getObj : named<T> const& -> T
```

A function merging two hashes discarding duplicate keys:

```
hashMerge : std::unordered_map<Key, Val> const& ->
```

```
    std::unordered_map<Key, Val> const& ->
    std::unordered_map<Key, Val> const&
```

A helper formatting function removing redundant consecutive whitespace characters:

```
formatWhistespace : std::string -> std::string
```

Is prefix of is a predicate returning whether one range is a prefix of the other:

```
isPrefixOf : (begin : Fwd_It) -> (end : Fwd_It) -> (prefix :
    Fwd_It) -> bool
```

A function removing common prefixes:

```
removeCommonPrefix : (lhsBegin : Fwd_It) -> (lhsEnd :
    Fwd_It) -> (rhsBegin : Fwd_It) -> (rhsEnd : Fwd_It) ->
    (result : Out_It) -> Fwd_It
```

Split implements a string splitting on tokens.

```
split : std::string const& -> char ->
    std::vector<std::string>
```

### fmapType

A group of recursive algorithms analysing type signatures, expressed as C++ functors (callable objects). The structure **getNames** is functionally equivalent to:

```
getNames : type -> std::vector<std::string>
```

which yields all names used in a type. There is also a structure **getNamesBy**, which is basically:

```
getNamesBy : type -> (string -> bool) ->
    std::vector<std::string>
```

It takes a predicate and filters the names as well as yields them. A utility function creating a view over a type as a function type:

```
viewAsFuncType : type const& -> funcType const&
```

A structure **getFuncTypeArguments** which is functionally equivalent to:

```
getFuncTypeArguments : type const& -> std::vector<funcType>
```

That is, it returns type arguments of a function signature. Finally the module specifies a structure **getRetType**, which is conceptually:

```
getRetType : funcType const& -> dataType
```

This return the return type of a function.

### B.8.2 Module instancedF

InstancedF is a module representing a type after unification, that is, proper substitution of type variables. It is an intermediate utility (with a structure **instacedF**) representation that is in place just for type safety and all it does is ensures that the unification algorithm is invoked before the type is being further processed.

### B.8.3 Module writerContextM

This module represents a state of computation (class **notAMonad**)with the following data members:

- envsT : deque<environment<» which represents active environments
- infsT : deque<string> denoting active inferences
- gensT : deque<string> encoding active synthesis to Haskell

An operation to lift said fields into this type:

```
mreturn : envsT -> deque<string> -> deque<string> ->
   notAMonad
```

An operation to chain functions over the type:

```
mbind : notAMonad -> std::function<notAMonad(envT)> ->
   notAMonad
```

And getters for each of the fields, which are just data member names prefixed with ”get”. Also setters and some specializations of them.

## B.9 ./type_operations

Aggregates modules implementing operations on propositions / types.

### B.9.1 Module unification

An implementation of a unification and a substitution algorithm.

```
unify : type const& -> type const& -> std::set<substitution>
```

Given two types to unify, generates a set of substitutions that lead to equivalency of said types.

```
substitute : curriedType const& -> std::set<substitution>
    const& -> curriedType
```

Given a type and a set of substitutions, substitute yields a new type with all the substitutions performed.

### B.9.2 Module substitution

Module containing only a structure (named **substitution**) aggregating two string representations of a type, used as an intermediary for unification. It is for all intents and purposes just a pair of two strings.

## B.10 ./analysis

A folder containing modules that perform function analysis and implementation synthesis.

### B.10.1 Module analysis

A class called **implementation** representing an implementation of a given function signature. Its fields are as follows:

- constructors : std::unordered_map<curriedType, std::vector<wheels::named< curriedType»>representing a hash map keeping information about how every type can be constructed
- recursionInfo : std::unordered_map<std::string, std::unordered_map< std::string, std::vector<std::size_t»>is a structure keeping tracks of recursive types

It contains an important function:

```
inferImplementation : curriedType ->
   std::vector<wheels::named<implementation::curriedType>>
   -> std::vector<implementation>
```

Which orchestrates the analysis and generation by invoking algorithms that transitively invoke tactics.

### B.10.2 Module tactics

This module implements all tactics as well as their combinations in algorithms. It also contains many utility functions.

```
getRecursiveParts : dataType const& -> typeCtorsMapT const&
   -> std::unordered_map<std::string,
   std::vector<std::size_t>>
```

Gets recursive parts of a data type. There is also a structure called **occurs** that is functionality equivalent to:

```
occurs : type -> bool
```

Which is an implementation of occurs check.

```
getIgnoredNames : std::unordered_map<std::string,
   std::vector<std::size_t>> const& indices ->
   std::unordered_map<std::string, std::vector<std::string>>
   -> std::unordered_map<std::string,
   std::vector<std::string>>
```

Get ignored names in combination with occurs prevent cyclic deconstruction of recursively defined data types.

```
skipDeconstruction : envT const& ->
   std::unordered_map<std::string, std::vector<std::string>>
   const& -> std::string const& ctorName -> bool
```

SkipDeconstruction is a predicate that allows us to skip deconstruction of sum types.

```
contextIntersection : envT const& -> envT const& -> envT
```

Performs intersection of contexts treating them as sets.

```
generateApplyPlaceholders : std::deque<std::string> gens ->
   unsigned arity -> std::vector<int> const& ->
std::string name -> std::deque<std::string>
```

Generates dummy arguments to apply tactic, allowing us to fill in parameters later.

```
getArity : curriedType -> unsigned
```

Gets arity of a curried type, treating it as though it was uncurried.

```
injectGenerations : notAMonad& -> std::deque<std::string> ->
   void
```

In combination with generateApplyPlaceholders, fills in function parameters after apply.

```
fixContext : std::string -> std::string
```

Mends inconsistencies in generated strings.

```
trivial : envT -> notAMonad
```

Implements the trivial tactic.

```
attempt : F -> envT -> Ts&&... -> notAMonad
```

Attempts a tactic, does not error if invocation unsuccessful.

```
intro : envT -> notAMonad
```

Implements the intro tactic.

```
intros : envT -> notAMonad
```

Implements the intros tactic.

```
clear : envT -> std::size_t -> notAMonad
```

Implements the clear tactic.

```
elimSum : envT -> envIter -> typeCtorsMapT const& ->
   notAMonad
```

Implements the elimSum tactic.

```
elimSumIntro : envT -> std::size_t -> typeCtorsMapT const&
   -> skipT const& -> valid -> std::deque<valid>% vs ->
   std::deque<int>& -> std::pair<notAMonad, skipT>
```

Implements a part of the Solve by Introduction algorithm, generating names to skip on the next pass in case they are recursive.

```
introsElims_if : envT -> typeCtorsMapT const& ->
   std::vector<std::string> const& -> notAMonad
```

Second part of the Solve by Introduction algorithm.

```
introsFuncargs : envT -> typeCtorsMapT const& ->
   recursionMapT const& -> notAMonad
```

Finishes the Solve by Introduction algorithm.

```
apply : envT -> std::size_t -> notAMonad
```

Implements the apply tactic.

```
applySome : envT -> unsigned -> std::pair<notAMonad,
   wheels::named<curriedType>>
```

A helper function for the Solve by Apply algorithm.

```
applySolve : envT env -> typeCtorsMapT const& -> notAMonad
```

Implements the Solve by Apply algorithm.