Design and Algebraic Implementation of a Functional Programming Language

Bachelor Thesis

Nicolas Lenz

2020-07-09

(Updated version, created 2022-06-20 15:52)

TU Dortmund University
Department of Computer Science
Chair 14 for Software Engineering

Prof. Dr. Jakob Rehof
Christoph Stahl, M. Sc.

Contents

1 Introduction

In this thesis a basic functional programming language, named *Lightfold*, with a dependent type system is designed and implemented based on a previously established theoretical foundation using lambda calculus and algebraic modeling techniques.

Dependent typing allows types to depend on values, so that return types of functions can be computed from their input value. This enables writing safe programs by assigning types to functions and data structures for which that would not be possible in a non-dependent type system.

The implementation consists of multiple parts. The parser is based on monadic parsing: Monadic combinators are used to compose complex parsers from simpler ones.

The compiler that translates the parsed abstract syntax tree into a core language is constructed following the principles of algebraic modeling, a generic and abstract approach to specifying the structure and interpretation of data types while leaving the interpretation open. Algebraic modeling allows the generalization of fold functions from lists to arbitrary data types as well. These capabilities are employed to implement a compiler infrastruture that can be repurposed for different output formats with little change. The implementation of the compiler logic itself can be kept concise thanks to the repetitive code being abstracted away.

The internal representation of Lightfold is based on the dependently typed lambda calculus. It utilizes de Bruijn indices for unambiguous variable representation and a bidirectional type system that combines type checking and type inference so that less type annotations are required. This core language is processed through a type checker to ensure correct typing and an evaluator to compute the results of programs.

Two frontends for using the language are provided: an interpreter, reading input from a file, and a shell for reading input interactively.

This thesis intends to give a practical example of a low-complexity dependently typed language with simple and understandable code to show the viability of this approach to typing. Furthermore, we aim to demonstrate the advantages and the disadvantages of algebraic compiler construction for implementing a functional language in interaction with the other programming techniques used.

1.1 Structure

This work is divided into seven chapters:

1. This introduction describes the motivation and the structure of this thesis.
2. The theoretical foundation chapter lays out the theoretical methods used in this thesis. The simply typed lambda calculus and its evaluation as well as typing rules are presented. Based on that the dependently typed lambda calculus, the basis for Lightfold, is introduced. Furthermore, bidirectional typing, de Bruijn indices for use instead of named variables, the basics of algebraic modeling, and usage of eliminators are explained.
3. In the methods chapter the various practical methods used in the implementation are shown, starting with the programming language Haskell and monadic parsing. Moreover, the implementation of interactive shells and the algebraic modeling techniques from the theory chapter is explained.
4. The design chapter develops, explains and justifies the design and design choices of the language Lightfold. The architecture of the language, the core language LightfoldCore and the surface syntax are presented.
5. The implementation chapter gives an overview of the implementation and how the theory and the methods from earlier chapters are utilized. The implementation consists of a parser, an algebraic compiler to the core language, a type checker, an evaluator and two frontends: an interpreter reading input from a file, and a shell receiving inputs from the user interactively.
6. In the evaluation the implementation is tested and analyzed. Examples of its usage will be given and the advantages and disadvantages of the methods used for it are discussed. An outlook on possible future developments and research paths is given.
7. In the conclusion, the thesis results are summarized.

2 Theoretical Foundation

In this chapter the underlying theoretical methods of the language will be shown, explained and motivated. At first we look at the simply typed lambda calculus, $\lambda_{\rightarrow}$, and its type and evaluation rules. From that we build up to different typing styles and introduce bidirectional typing. The dependently typed lambda calculus λ_{Π} is presented, the calculus that will serve as foundation for *Lightfold*. Furthermore, we will introduce algebraic modeling and data types and how folds and eliminators can be used to perform computations on them. Lastly, eliminators and general custom data types are presented.

2.1 Simply Typed Lambda Calculus

First, we take a look at the *simply typed lambda calculus*, or $\lambda_{\rightarrow}$ in short. It is the common basis for other typed lambda calculi, including the one we use for for Lightfold.

Definition 2.1.1: $\lambda_{\rightarrow}$ Syntax

$\lambda_{\rightarrow}$ consists of two separate constructs: *types* and *terms*, represented by τ and e respectively and defined in Backus-Naur form. α is a predetermined set of *base types*, x denotes a variable identifier.

$$\begin{aligned} \tau ::= {} & \alpha \\ \mid {} & \tau \rightarrow \tau' \end{aligned}$$

$$\begin{aligned} e ::= {} & x \\ \mid {} & e\, e' \\ \mid {} & \lambda x.e \end{aligned}$$

[1, Sec. 2.1]

As can be seen from definition 2.1.1, a $\lambda_{\to}$ term can be one of three constructs:

- A lambda abstraction $\lambda x.e$, consisting of a variable identifier x and the inner term e. This represents a function that receives an argument that is bound to the variable x, which can then be used in the inner term e to access the argument.
- A application $e\,e'$ of an argument term e' to another term e serving as function.
- A variable denoted by an identifier x. A variable is called *bound* if it refers to a lambda binder. If there is no such binder fitting the variable, it is called a *free variable*. For example, in the term $\lambda x.x\;y$ the variable x is bound and y is free.

Types, on the other hand, can be one of two constructs:

- A base type, taken from a predetermined set of base types α.
- A function type $\tau \to \tau'$, which is comprised of a type τ for the function argument, called *domain* of the function, and a type τ' for the function result, called *range*.

We use the notation $e : \tau$ to indicate that a term e has the type τ.

Example 2.1.2: $\lambda_{\to}$ Terms

These are some examples of lambda terms and correct types for them.

- $\lambda x.x : \tau \to \tau$ (identity function)
- $\lambda x.\lambda y.x : \tau \to \tau' \to \tau$ (constant function that always returns the first argument)
- $\lambda f.\lambda x.f\,x : (\tau \to \tau') \to \tau \to \tau'$ (apply function)

Evaluation

The semantics of $\lambda_{\to}$ terms stem from their ability to represent computations by being *evaluated*. $e \Downarrow v$ denotes that the result of fully evaluating a term e is v. For example, the identity function $\lambda x.x$ returns the argument itself for every input x. When a value is applied to this function, the result is the value itself: $(\lambda x.x)\;a \Downarrow a$.

This intuitive approach to evaluation can be formalized using evaluation rules as shown in definition 2.1.3. A rule consists of premises, written above a vertical line, and a conclusion, written below that line.

Definition 2.1.3: λ→ Evaluation Rules

$$\frac{e \Downarrow \lambda x.v \qquad v[x \mapsto e'] \Downarrow v'}{e\, e' \Downarrow v'}\ (\text{E-App}) \qquad \frac{e \Downarrow v \qquad e' \Downarrow v'}{e\, e' \Downarrow v\, v'}\ (\text{E-AppTrans})$$

$$\frac{e \Downarrow v}{\lambda x.e \Downarrow \lambda x.v}\ (\text{E-Lam}) \qquad \frac{}{x \Downarrow x}\ (\text{E-Var})$$

[1, Fig. 1]

(E-App) performs *β-reduction*, the process of applying an argument to a lambda function. For that, inside the lambdas inner term all occurrences of the variable bound by the lambda are replaced with the argument.

The substitution of all x with y in a term e is denoted by $e[x \mapsto y]$. The rule states that if a term e evaluates to a lambda term $\lambda x.v$, and the result of replacing all occurrences of the variable x in v with e' is v', it can be concluded that the application $e\, e'$ evaluates to v'.

(E-AppTrans) evaluates both sides of an application without performung β-reduction if the function side of an application does not evaluate to a lambda term.

(E-Lam) evaluates lambda terms by recursively evaluating their inner term.

(E-Var) states that variables not yet substituted through a β-reduction evaluate to themselves.

By repeated application of the evaluation rules we can derive the evaluation result for a given λ→ term. An exemplary evaluation is detailed in example 2.1.4.

Example 2.1.4: λ→ Evaluation

We show that $(\lambda x.\lambda y.x)\; a\; b$ evaluates to a using the evaluation rules for λ→.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x \Downarrow x}\ (\text{E-Var})}{\lambda y.x \Downarrow \lambda y.x}\ (\text{E-Lam})}{\lambda x.\lambda y.x \Downarrow \lambda x.(\lambda y.x)}\ (\text{E-Lam}) \qquad \lambda y.x[x \mapsto a] \Downarrow \lambda y.a}{(\lambda x.\lambda y.x)\; a \Downarrow \lambda y.a}\ (\text{E-App}) \qquad a[y \mapsto b] \Downarrow a}{(\lambda x.\lambda y.x)\; a\; b \Downarrow a}\ (\text{E-App})$$

Typing

$\lambda_\to$ terms can not only be evaluated, they can also be type checked against $\lambda_\to$ types. For typing we introduce the concept of *typing contexts*. A typing context Γ is an unordered list of tuples of a variable x and their type τ, each denoted in the format $x : \tau$. A variable may only occur once in a context. We write $\Gamma \vdash x : \tau$ to denote that x is of the type τ in the context Γ. The typing rules are given in definition 2.1.5.

Definition 2.1.5: $\lambda_\to$ Typing Rules

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\;\text{(T-Var)} \qquad \frac{\Gamma \vdash e : \tau \to \tau' \qquad \Gamma \vdash e' : \tau}{\Gamma \vdash e\,e' : \tau'}\;\text{(T-App)}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'}\;\text{(T-Lam)}$$

(T-Var) states that from any context and $x : \tau$, $x : \tau$ itself can be inferred.
(T-App) is the application rule. If a term e is a function of type $\tau \to \tau'$ and e' of type τ, then the application $e\,e'$ has the type τ'.
(T-Lam) lets us derive the type of lambda expressions: If assuming that $x : \tau$ yields $e : \tau'$, the lambda term $\lambda x.e$ is a function of the type $\tau \to \tau'$.

Through repeated application of these rules it can be proven that a term has a specific type.

Example 2.1.6: $\lambda_\to$ Type Derivation

We prove that the term $(\lambda x.\lambda y.x)\,a\,b$ is of type τ in the initial context $\Gamma = a : \tau, b : \tau'$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, x : \tau, y : \tau' \vdash x : \tau}\;\text{(T-Var)}}{\Gamma, x : \tau \vdash \lambda y.x : \tau' \to \tau}\;\text{(T-Lam)}}{\Gamma \vdash \lambda x.\lambda y.x : \tau \to \tau' \to \tau}\;\text{(T-Lam)} \qquad \dfrac{}{\Gamma \vdash a : \tau}\;\text{(T-Var)}}{\Gamma \vdash (\lambda x.\lambda y.x)\,a : \tau' \to \tau}\;\text{(T-App)} \qquad \dfrac{}{\Gamma \vdash b : \tau'}\;\text{(T-Var)}}{\Gamma \vdash (\lambda x.\lambda y.x)\,a\,b : \tau}\;\text{(T-App)}$$

As we can see from the evaluation and typing rules, $\lambda_\to$ allows terms to depend on terms via application and lambda abstraction. On the other hand, it neither allows terms or types to depend on types nor types to depend on terms. This makes it the basic typed system in the λ-cube introduced by Barendregt in [2], meaning that all other lambda calculi classified in the λ-cube are built upon $\lambda_{\to}$.

It should be noted that not all syntactically valid terms have a valid type in $\lambda_{\to}$. For example, the typing rules fail for the term $\lambda x.x\,x$ as illustrated in figure 2.1. We cannot assign a type to x. As it is being used as a function, it needs to be a function type, but it also is the argument to that function. We would need to harmonize the constraints $x : \tau$ and $x : \tau \to \tau'$ at the same time. While not done in this work, that restriction can be avoided by allowing multiple types to be assigned to a single type. This concept is called *intersection types* and was among others examined by Barendregt, Coppo and Dezani-Ciancaglini in [3].

$$\dfrac{\dfrac{{\color{red} x : \tau \vdash x : \tau \to \tau'} \qquad \dfrac{}{x : \tau \vdash x : \tau}\,(\text{T-Var})}{x : \tau \vdash x\,x : ?}\,(\text{T-App})}{\vdash \lambda x.x\,x : ?}\,(\text{T-Lam})$$

Figure 2.1: Trying to derive a type for $\lambda x.x\,x$

Typing Styles

So far $\lambda_{\to}$ has been used with types as an external feature. $\lambda_{\to}$ terms do not contain types in themselves, we only check terms against externally provided types. This typing style is known as *Curry-style typing* [4].

While we can check terms against a given type, *inferring* the types of a given term is more complex in Curry-style $\lambda_{\to}$. We can try constructing a simple type inference algorithm from the typing rules. While that would work for variables and applications, when trying to write the case for lambdas, we run into problems as illustrated in listing 2.1.7 using pseudocode: It is unclear what we should have written in the places marked with question marks.

This problem can be demonstrated with the identity function $\lambda x.x$. It can be successfully type checked against multiple type terms, for example $a \to a$, $b \to b$ or $(a \to b) \to (a \to b)$. Because of variable renaming these types are equivalent, and we could output only the most general type, but this would require a more involved algorithm.

Listing 2.1.7: Type Inference in Curry-Style $\lambda_\rightarrow$

```
inferType ctx (Variable x) = lookup ctx x
inferType ctx (Application e e')
  = case (inferType ctx e, inferType ctx e') of
    (τ → τ', τ) → τ'
    _ → error
inferType ctx (Lambda x e)
  = let τ' = inferType (ctx + (x, ?)) e
    in ? → τ'
```

Adapted from [5, Sec. 1.1]

Another problem is the case for applications. For example, the algorithm will fail if the type of the function is inferred to be $a \rightarrow a$, but the type of the argument can be b. While the argument type fits the function which can be seen by renaming variables, simple syntactical comparison can not capture that.

The counterpart to Curry-style typing is *Church-style typing* [4]. Instead of understanding types only as external annotations, they are incorporated into the terms, giving each term an unambiguous type. To accomplish that the grammatical rule for lambda terms is changed from $\lambda x.e$ to $\lambda x : \tau.e$, requiring an explicit type for every lambda abstraction.

This avoids the type inference problems of Curry-style typing. With Church-style typing it is impossible to write an ambiguous identity function like $\lambda x.x$ because we have to supply the type of x right in the term. Based on this we can write a simple `inferType` algorithm, shown in listing 2.1.8: The previous problem that we couldn't easily infer the type of the variable bound by a lambda has been avoided through the supplied annotation, and type equality can be checked through simple syntactical equality since a term always has an unique type in Church-style $\lambda_\rightarrow$.

Listing 2.1.8: Type Inference in Church-Style $\lambda_\rightarrow$

```
inferType ctx (Variable x) = lookup ctx x
inferType ctx (Application e e')
  = case (inferType ctx e, inferType ctx e') of
    (τ → τ', τ) → τ'
    _ → error
```

```
inferType ctx (Lambda x τ e)  -- type of input now also supplied
  = let τ' = inferType (ctx + (x, τ)) e
    in τ → τ'
```

Adapted from [5, Sec. 1.1]

Bidirectional Type System

While Church-style typing allows us to easily and automatically infer the types of our terms, having to explicitly annotate the input of *every* lambda function is cumbersome and unnecessarily verbose. We can use another approach: bidirectional typing, introduced by Pearce and Turner in [6], combining type checking and type inference.

We differentiate between *inferable* terms, whose type can be inferred automatically, and *checkable* terms, for which a type to check against has to be supplied. When type checking, the algorithm alternates between checking and inference mode depending on the term being checked, so that type annotations are only necessary in places where the type cannot be inferred automatically. To that end we add an alternative rule for terms to the $\lambda_{\rightarrow}$ grammar from definition 2.1.1: $e : \tau$, allowing us to annotate arbitrary terms with types.

The type relation $:$ from our previous rules is split into two: $e :_{\rightarrow} \tau$, meaning that e can be inferred to have the type τ, and $e :_{\leftarrow} \tau$, meaning that e can be checked to have the type τ. We write $\tau : *$ to state that τ is a base type.

The bidirectional variant of the typing rule set seen in definition 2.1.9 is expanded from the basic Curry-style one introduced in definition 2.1.5.

Definition 2.1.9: $\lambda_{\rightarrow}$ Bidirectional Typing Rules

$$\frac{\Gamma \vdash \tau : * \qquad \Gamma \vdash e :_{\leftarrow} \tau}{\Gamma \vdash (e : \tau) :_{\rightarrow} \tau}\ (\text{T-Ann}) \qquad \frac{\Gamma \vdash e :_{\rightarrow} \tau}{\Gamma \vdash e :_{\leftarrow} \tau}\ (\text{T-Chk}) \qquad \frac{}{\Gamma, x : \tau \vdash x :_{\rightarrow} \tau}\ (\text{T-Var})$$

$$\frac{\Gamma \vdash e :_{\rightarrow} \tau \to \tau' \qquad \Gamma \vdash e' :_{\leftarrow} \tau}{\Gamma \vdash e\, e' :_{\rightarrow} \tau'}\ (\text{T-App}) \qquad \frac{\Gamma, x : \tau \vdash e :_{\leftarrow} \tau'}{\Gamma \vdash \lambda x.e :_{\leftarrow} \tau \to \tau'}\ (\text{T-Lam})$$

[1, Fig. 3]

The rules (T-Ann) and (T-Chk) have been added. (T-Ann) checks against the newly allowed type annotations, ensuring that the value actually checks against the annotated type. (T-Chk) simply states that a term that can be inferred to have a type can also be checked to have that type. Example 2.1.10 shows that the type of the term from the previous non-bidirectional example 2.1.6 can be inferred with a single type annotation added.

Example 2.1.10: $\lambda_\rightarrow$ Bidirectional Type Derivation

We prove that the term $((\lambda x.\lambda y.x) : \tau \rightarrow \tau' \rightarrow \tau)\, a\, b$ can be inferred to have the type τ in the initial context $\Gamma = a : \tau, b : \tau'$ given that τ and τ' are base types.

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \tau : * \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, x : \tau, y : \tau' \vdash x :_\rightarrow \tau}\;(\textsc{T-Var})}{\Gamma, x : \tau, y : \tau' \vdash x :_\leftarrow \tau}\;(\textsc{T-Chk})}{\Gamma, x : \tau \vdash \lambda y.x :_\leftarrow \tau' \rightarrow \tau}\;(\textsc{T-Lam})}{\Gamma \vdash \lambda x.\lambda y.x :_\leftarrow \tau \rightarrow \tau' \rightarrow \tau}\;(\textsc{T-Lam})}{\Gamma \vdash (\lambda x.\lambda y.x) : \tau \rightarrow \tau' \rightarrow \tau :_\rightarrow \tau \rightarrow \tau' \rightarrow \tau}\;(\textsc{T-Ann}) \qquad \dfrac{\dfrac{}{\Gamma \vdash a :_\rightarrow \tau}\;(\textsc{T-Var})}{\Gamma \vdash a :_\leftarrow \tau}\;(\textsc{T-Chk})}{\Gamma \vdash ((\lambda x.\lambda y.x) : \tau \rightarrow \tau' \rightarrow \tau)\, a :_\rightarrow \tau' \rightarrow \tau}\;(\textsc{T-App}) \qquad \dfrac{\dfrac{}{\Gamma \vdash b :_\rightarrow \tau'}\;(\textsc{T-Var})}{\Gamma \vdash b :_\leftarrow \tau'}\;(\textsc{T-Chk})}{\Gamma \vdash ((\lambda x.\lambda y.x) : \tau \rightarrow \tau' \rightarrow \tau)\, a\, b :_\rightarrow \tau}\;(\textsc{T-App})$$

2.2 Dependently Typed Lambda Calculus

Dependent typing is the concept of allowing types to depend on terms. It allows us to encode more restrictions on values into the type system. An example of a commonly used function whose type can only be correctly expressed using dependent types is `printf`.

`printf`, or some variant, is a string formatting function in many programming languages[1]. It takes a string containing a number of formatting specifiers and a number of other arguments, one for each formatting specifier, its type depending on said specifier.

An example call to C's `printf`[2] is:

[1]For example in C (https://linux.die.net/man/3/printf), Haskell (https://hackage.haskell.org/package/base-4.14.0.0/docs/Text-Printf.html#v:printf) or Python (https://docs.python.org/3/library/stdtypes.html#str.format).

[2]https://linux.die.net/man/3/printf

```
printf("Name: %s, Age: %d", "Otto", 42)
```

This would yield `"Name: Otto, Age: 42"`. The formatting string contains two specifiers, one for inserting a string (`%s`) and one for a decimal number (`%d`), and the two fitting arguments are provided after the formatting string. However, encoding the type of such a function in a strict type system is problematic.

Haskell being a statically typed language, its implementation of `printf`[3] uses a typeclass to allow different types and numbers of arguments. This allows the type checker to check that only arguments of valid types are given. However, it cannot check that the amount and types of the arguments actually fit the formatting string, since Haskell's type system is not dependent and has no means of examining the actual value of the formatting string. For example, `printf "%s" "foo" "bar"` and `printf "%s" 13` both compile and do not fail until runtime.

Dependent typing aims to alleviate this shortcoming: It allows binding the *value* of a function argument not only in the function result, but also in its *type* and use it to compute the rest of the type *depending* on the input value. In our example that means we can analyze the formatting specifiers and compute the type of the required arguments during type checking, enabling us to cleanly and safely specify `printf`'s type.

The type of such a type-safe `printf` function might look like this, formulated in Haskell-like pseudocode:

```
printf :: (input :: String) → printfType (getSpecifiers input)
```

Here, `getSpecifiers :: String → [String]` is a function for retrieving the formatting specifiers and `printfType :: [String] → *` is a function for constructing `printf`'s return type. A recursive implementation of `printfType` could look like this:

```
printfType :: [String] → *
printfType [] = String
printfType ("%s":specs) = String → (printfType specs)
printfType ("%d":specs) = Int → (printfType specs)
[...]
```

This means that `printf` expects one argument for each formatting specifier: a string for each `"%s"` and an integer for each `"%d"`. The return value of `printf` is always a string, which is why `printfType` returns just the type `String` for an empty list of specifiers.

[3]https://hackage.haskell.org/package/base-4.14.0.0/docs/Text-Printf.html#v:printf

As an example, `printf` called with the formatting string `"Name: %s, Age: %d"` would therefore have a final evaluated type:

```
printf :: String → String → Int → String
```

This is the expected result. `printf` takes a string argument for the formatting string, a string argument for the first formatting specifier `"%s"`, an integer argument for the second specifier `"%d"`, and returns a string as result.

We can see that for dependent types to be useful, we not only need to be able to bind values in types, but also need the ability to perform computations with types and return them as result of computations lest we would not be able to actually use the bound values.

Dependently typed programming languages do already exist and can be used: One of the earliest examples is *Cayenne*, introduced in 1998 by Augustsson in [7]. More modern, still maintained and practically usable specimen are Idris [4] or Agda [5].

Syntax

This concept of dependent types is formulated into an extended lambda calculus, building on $\lambda_{\rightarrow}$ from section 2.1. What we need to add is: annotations for bidirectional typing as already explained in section 2.1, a way to bind values in types and a way to compute types.

To this end the *dependently typed lambda calculus*, or λ_{Π} in short, is defined in definition 2.2.1. The definition is explained in the following.

[4]https://www.idris-lang.org
[5]https://wiki.portal.chalmers.se/agda/pmwiki.php

Definition 2.2.1: λ_Π Syntax

λ_Π is defined using *Backus-Naur form*. x denotes a variable identifier.

$$
\begin{aligned}
e, \rho, \kappa ::=\ & e : \rho \\
\mid\ & * \\
\mid\ & \Pi x : \rho . \rho' \\
\mid\ & x \\
\mid\ & e\, e' \\
\mid\ & \lambda x . e
\end{aligned}
$$

[1, Sec. 3.1]

There is only a single syntax rule in λ_Π. We use different nonterminals to differentiate where a term is meant as an expressions (e), a type (ρ) or as a kind (κ, kinds are types of types), but they are all identical: in λ_Π *everything is a term.* Thanks to this change, variables, functions and applications can now return type terms in just the same way they return normal terms.

As types are terms now, their type has to be expressible as well. For that purpose $*$ is introduced as *kind of types.*

The second main difference is that instead of $\tau \to \tau'$ as function type, λ_Π uses the *dependent function space* $\Pi x : \rho.\rho'$. As before it contains a domain (ρ) and a range type (ρ'). It distinguishes itself by binding the actual value of the input to a variable (x) and allowing the range to *depend* on the value of the input. The name “dependent types” stems from that. Just like the λ binder allows terms to depend on terms, Π allows types to depend on terms.

Evaluation & Typing

The evaluation rules for λ_Π, found in definition 2.2.2, are extended from those of $\lambda_\to$ from definition 2.1.3. New rules are (E-Ann), that simply ignores type annotations during evaluation, (E-Star) that evaluates $*$ to itself and (E-Pi) for the dependent function space.

Definition 2.2.2: λ_Π Evaluation Rules

$$\frac{e \Downarrow v}{e : \rho \Downarrow v}\ (\textsc{E-Ann}) \qquad \frac{}{* \Downarrow *}\ (\textsc{E-Star}) \qquad \frac{\rho \Downarrow \tau \qquad \rho' \Downarrow \tau'}{\Pi x : \rho.\rho' \Downarrow \Pi x : \tau.\tau'}\ (\textsc{E-Pi}) \qquad \frac{}{x \Downarrow x}\ (\textsc{E-Var})$$

$$\frac{e \Downarrow \lambda x.v \qquad v[x \mapsto e'] \Downarrow v'}{e\,e' \Downarrow v'}\ (\textsc{E-App}) \qquad \frac{e \Downarrow n \qquad e' \Downarrow v'}{e\,e' \Downarrow nv'}\ (\textsc{E-AppTrans})$$

$$\frac{e \Downarrow v}{\lambda x.e \Downarrow \lambda x.v}\ (\textsc{E-Lam})$$

[1, Fig. 8]

The typing rules in definition 2.2.3 are extended from the bidirectional typing rules for $\lambda_\rightarrow$, given in definition 2.1.9.

Definition 2.2.3: λ_Π Typing Rules

$$\frac{\Gamma \vdash \rho :_\leftarrow * \qquad \rho \Downarrow \tau \qquad \Gamma \vdash e :_\leftarrow \tau}{\Gamma \vdash (e : \rho) :_\rightarrow \tau}\ (\textsc{T-Ann}) \qquad \frac{}{\Gamma \vdash * :_\rightarrow *}\ (\textsc{T-Star})$$

$$\frac{\Gamma \vdash \rho :_\leftarrow * \qquad \rho \Downarrow \tau \qquad \Gamma, x : \tau \vdash \rho' :_\leftarrow *}{\Gamma \vdash \Pi x : \rho.\rho' :_\rightarrow *}\ (\textsc{T-Pi}) \qquad \frac{\Gamma \vdash e :_\rightarrow \tau}{\Gamma \vdash e :_\leftarrow \tau}\ (\textsc{T-Chk})$$

$$\frac{\Gamma \vdash e :_\rightarrow \Pi x : \tau.\tau' \qquad \Gamma \vdash e' :_\leftarrow \tau \qquad \tau'[x \mapsto e'] \Downarrow \tau''}{\Gamma \vdash e\,e' :_\rightarrow \tau''}\ (\textsc{T-App})$$

$$\frac{}{\Gamma, x : \tau \vdash x :_\rightarrow \tau}\ (\textsc{T-Var}) \qquad \frac{\Gamma, x : \tau \vdash e :_\leftarrow \tau'}{\Gamma \vdash \lambda x.e : \Pi x : \tau.\tau'}\ (\textsc{T-Lam})$$

[1, Fig. 10]

We explain these rules one by one:

(T-Ann) has been changed to first evaluate the annotated type (using the evaluation rules from definition 2.2.2) before checking the value against it.

(T-Star) has been added to give $*$ itself as type as a simplification. Note that this simplification renders the type system unsound as allows encoding of a variant of Girard's Paradox [1, Sec. 5] [8].

(T-Pi) is the main addition: a rule for checking the validity of the types used in a dependent function type. The range is checked to be a type. The domain is checked to be a type as well, assuming that the variable bound by the Π-binder being checked is of the evaluated range type.

(T-Chk) is unchanged, it simply states that if a term can be inferred to have a type, it can also be checked against it.

(T-App) is where the main dependent typing happens: It has been adjusted to substitute the variables bound by the Π-binder in the range type of the function with the value of the argument, allowing us to use said value inside the range.

(T-Var) is unchanged and allows the type of a variable to be inferred from a context containing said variable.

(T-Lam) now uses the dependent function space as type for lambda terms.

Note that our version of λ_{Π} is not equivalent to the λ_{P} calculus from the *λ-cube* [2] because of the simplification expressed in typing rule (T-Star) where we give the kind $*$ itself as type.

2.3 De Bruijn Indices

Bound variables in lambda terms are usually denoted using names. For example, the identity function can be represented as $\lambda x.x$ – the lambda binds its argument to the variable x which can then be used inside the lambda term. While this method is easy, intuitive and makes it possible to use meaningful variable names, it has some disadvantages as internal representation. For algorithmic processing a variable name and its meaning is irrelevant, but opens up the possibility of naming conflicts and makes equality checking harder. For example, $\lambda x.x$ and $\lambda y.y$ both represent the identity function – they are semantically identical, but not syntactically.

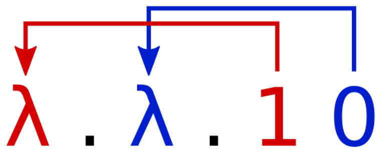

Figure 2.2: De Bruijn index usage 1

These disadvantages can be avoided using *de Bruijn indices* [9]. Bound variables are represented through indices instead of names. This index is the number of in-scope

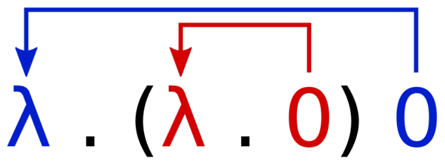

Figure 2.3: De Bruijn index usage 2

binders between (that is, we start indexing at zero) the variable and the actual binder that bound the value we want to refer to. At any given position in the term each binder in scope is unambiguously referenced by only one index, therefore there are no naming conflicts and the terms are invariant to variable renaming.

The concept is illustrated in figure 2.2 for the term $\lambda.\lambda.1\ 0$: There are two nested lambda abstractions. Inside them the de Bruijn index 1 references the outer binder, because the second lambda is between them. On the other hand, the "0" references the inner lambda binder as there are zero other binders in between. An equivalent term using named variables is $\lambda x.\lambda y.x\ y$.

Figure 2.3 shows that only lambda binders that are actually in scope are counted by taking the example $\lambda.(\lambda.0)\ 0$. If simply read from left to right, there is a lambda binder in between the first lambda and the last variable "0", but since that binder is not in its scope, the last "0" references the first lambda. The inner "0" simply references the lambda right in front of it. An equivalent named term is $\lambda x.(\lambda y.y)\ x$.

Note that in λ_Π "binder" does not only refer to lambda abstractions, but also dependent function types with Π: Inside the range type the Π is a binder for the function input value that has the type given as domain.

Example 2.3.1: De Bruijn Indices

These are some examples of lambda terms in the usual name-based representation and de Bruijn index-based representation.

- $\lambda x.x \rightsquigarrow \lambda.0$ *(identity function)*
- $\lambda x.\lambda y.x \rightsquigarrow \lambda.\lambda.1$ *(constant function that returns the first argument)*
- $\lambda f.\lambda x.f x \rightsquigarrow \lambda.\lambda.(1\ 0)$ *(apply function)*

A downside to using de Bruijn indices is that variable substitution is more complex than with named variables. In section 2.1 we used straightforward replacement: To substitute the variable x with the variable y in the term e, denoted $e[x \mapsto y]$, all occurrences of x in e are directly replaced by y.

However, for terms with de Bruijn indexed variables we have to consider that a variable referencing a specific binder will have a different index depending on its position. Take $\lambda.(\lambda.1)\ 0$ for example: both variables reference the same outer lambda binder despite having different indices.

We also have to pay attention to the term being filled in: Variables inside that term need to be adjusted so they still reference the same binder. Take $\lambda.((\lambda.\lambda.1)\ 0)$ for example. The correct β-reduction of the marked lambda with the "0" would be $\lambda.(\lambda.1)$: The 0 had to be incremented to still reference the outside lambda after being pasted into another lambda term.

Moreover, variables pointing outside of the term being reduced also have to be adjusted. For example, $\lambda.((\lambda.1)\ 0)$. When β-reducing the marked lambda with the "0", we do not have to fill in anything as no variable referenced the marked binder. However, $\lambda.1$ would be incorrect as result. Because the marked lambda was removed during substitution, the variables inside it have to be decremented to still reference the same outside binders: The correct result is $\lambda.0$.

2.4 Algebraic Modeling

In this section the idea of folding lists known from various programming languages is introduced. It is shown that the concept of folds can be expanded to arbitrary algebraic data types. Signatures and Σ-algebras are presented as a formal way of specifying such data types in an abstract fashion for usage in generalized fold functions and more.

Lists

The type of recursive lists over a type `a`, written `[a]`, is defined as an inductive data type using Haskell syntax as follows:

```
data [a] = [] | a : [a]
```

A list is either the empty list `[]` or a value added to another list using `:`, also called cons constructor. By nesting these constructors we can model lists of values of arbitrary lengths.

Programs on lists can be written using pattern matching and recursion as shown in example 2.4.1. Generally, two cases are needed: one for the empty list with a constant and one for the cons constructor, that includes the actual recursive call.

Example 2.4.1: List Functions as Recursions

These are some common list functions expressed by means of recursion:

```
length :: [a] → Int
length [] = 0
length (x:xs) = 1 + length xs

sum :: [Int] → Int
sum [] = 0
sum (x:xs) = x + sum xs

or :: [Bool] → Bool
or [] = False
or (x:xs) = x || or xs
```

While this approach is intuitive, it also is repetitive. The recursion pattern has to be written each time over, but the functions differ only in what the constant value for the empty list is and what the computation on the recursive call in the cons case is. Conveniently, the recursion pattern can be abstracted away using folds.

Folding Lists

Not only from functional programming languages like Haskell[6], but also from languages like Python[7] or Java[8], we know the *fold* function on lists, also known as *accumulation* or *reduction*. The idea is to successively apply a function to the list elements until the list is completely *folded*. A function `foldList` that folds a list over the type `a` from right to left into a result of the type `b` can be written as follows:

[6]https://hackage.haskell.org/package/base-4.14.0.0/docs/Prelude.html#v:foldr
[7]https://docs.python.org/3/library/functools.html#functools.reduce
[8]https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html#reduce(java.util.function.BinaryOperator)

```
foldList :: (a → b → b) → b → [a] → b
foldList f x []     = x
foldList f x (y:ys) = f y (foldList f x ys)
```

`foldList` allow us to express list functions very concisely without having to write the actual recursion pattern ourselves. What would be expressed as a reduction case in the list recursion is given as first argument to foldList, and the empty case is supplied as second argument. `foldList` then performs the actual recursion, as shown for multiple functions in example 2.4.2, which also illustrates the conciseness of using `foldList` in comparison to the direct recursion used in example 2.4.1.

Example 2.4.2: List Functions as Foldings

These are some common list functions expressed by means of `foldList`:

```
length :: [a] → Int
length = foldList (const (+1)) 0

sum :: [Int] → Int
sum = foldList (+) 0

or :: [Bool] → Bool
or = foldList (||) False
```

A particular instance of a list can be visualized as a tree structure where the tree's nodes represent a constructor each, its children being the constructor's arguments. The calling tree built by a fold or a recursive function can be represented in the same way: A node represents a function call with the node's children as arguments.

Figure 2.4 shows this for the example list `[1,2,3]` (or written using the constructors `1:2:3:[]`) and the summation function `sum`. We can see that the structure is identical in both trees: Recursion and foldings essentially work by replacing all constructors in the term being reduced by values or functions fitting in the constructor's place.

`[]` has no arguments, so for `sum` the simple value `0` fits in its place. `:` has two arguments: one for the next element of the list, here a number, and one for the rest of the list. When reducing, that is the sum of the rest of the list, which is a number as well. `(+)` takes two numbers as arguments, which lets it fit in the place of `(:)`.

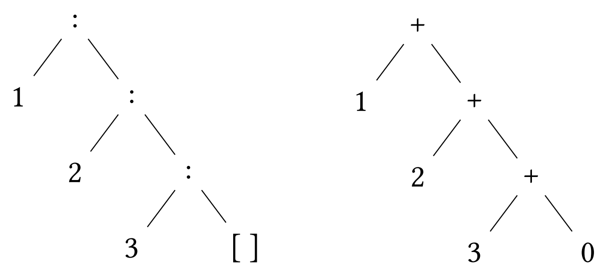

Figure 2.4: A list and its folding

Generalizing Folding

The concept of folding works for other algebraic data types built with constructors as well. The constructors are replaced with fitting functions. Example 2.4.3 demonstrates that for natural numbers and binary trees with data in the leafs.

Example 2.4.3: Other Foldable Data Types

Note that this code is case-sensitive: We use the lowercased variants of the constructor names as variable names for the respective fold arguments.

```
data Nat = Zero | Succ Nat

foldNat :: a → (a → a) → Nat → a
foldNat zero succ Zero     = zero
foldNat zero succ (Succ n) = succ (foldNat zero succ n)

isEven :: Nat → Bool
isEven = foldNat True not

data Tree a = Leaf a | Fork (Tree a) (Tree a)

foldTree :: (a → b) → (b → b → b) → BinaryTree a → b
foldTree leaf fork (Leaf x) = leaf x
foldTree leaf fork (Fork left right)
  = fork (foldTree leaf fork left) (foldTree leaf fork right)

sumTree :: BinaryTree Int → Int
sumTree = foldTree id (+)
```

The process is illustrated for `isEven (Succ (Succ (Succ Zero)))` ⇝ `False` (that is, checking whether three is even) in figure 2.5 and for `sumTree (Fork (Leaf 17) (Fork (Fork (Leaf 1) (Leaf 5)) (Leaf 9)))` ⇝ 32 in figure 2.6. It can be seen that folds on natural numbers and trees essentially work by replacing the constructors with the given functions just like for lists.

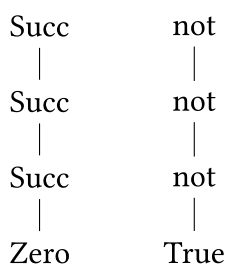

Figure 2.5: A natural number and its fold

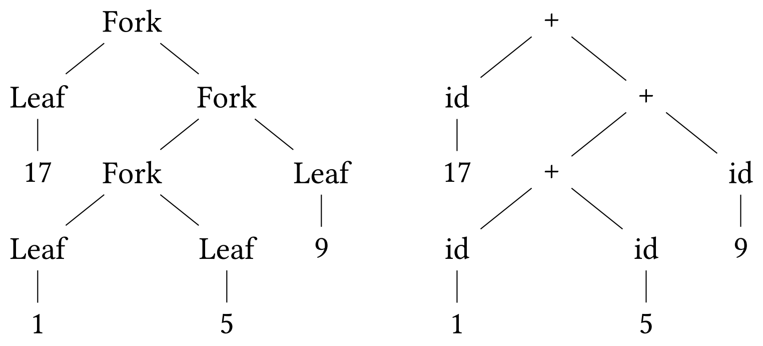

Figure 2.6: A tree and its fold

The programming language Haskell has a feature like this: for simple data types with exactly one type variable the Glasgow Haskell Compiler features the language extension `DeriveFoldable`[9] that allows automatic generation of fold functions without additional implementation.

[9]https://downloads.haskell.org/~ghc/8.8.3/docs/html/users_guide/glasgow_exts.html#deriving-foldable-instances

Abstract Data Types

Up to now, we talked about data types, folds and their generalization in a mostly informal way. In this subsection we will formalize these concepts using *signatures*.

The types like the lists we presented are *concrete* data types. They not only define the structure of lists, but also simultaneously define the concrete set of lists and functions to build elements of it: for lists that are the constructor functions `[]` and `:`.

Abstract algebraic data types, on the other hand, define the structure of data type while leaving the choice of interpretation open. Such abstract types are defined via *signatures* in definition 2.4.4. Signatures can be constructive or destructive [10, Sec. 2.3], but we only cover constructive ones.

Definition 2.4.4: Signatures

A constructive *signature* $\Sigma = (S, F)$ consists of:

- a set $S = \{s_1, \dots, s_n\}$ of symbols called *sorts*; and
- a set of named function types $F = \{f_1 : e_1 \to e'_1, \dots, f_m : e_m \to e'_m\}$ called *operations*.

  The e_i, called *domain* of the respective operation, are of the form $d_{i,1} \times \dots \times d_{i,k_i}$, with each d_i being either a set or a sort. k_i is called the *arity* of the operation f_i and must be zero or larger.

  The e'_i are always sorts and are called *range* of the corresponding operation.

$x \times y$ denotes the cartesian product of x and y.

Adapted from [10, Sec. 2.3].

Each sort of a signature corresponds to a single concrete data type: A signature can model multiple interdependent concrete data types. The operations on the other hand correspond to a constructor for the sort in the respective operation's range. Some examples of signatures are shown in example 2.4.5.

Example 2.4.5: Signatures

- *Nat* models the natural numbers.

$$\begin{aligned} Nat = (S = \{nat\}, F = \{&zero : nat, \\ &succ : nat \to nat\}) \end{aligned}$$

- *List*(X) models lists over the set X.

$$\begin{aligned} List(X) = (S = \{list\}, F = \{&empty : list, \\ &cons : X \times list \to list\}) \end{aligned}$$

- *Tree*(X) models trees with an arbitrary amount of children per node and node marks from X (note that these are different from the previously used leaf marked binary trees).

$$\begin{aligned} Tree = (S = \{tree, trees\}, F = \{&join : X \times trees \to tree, \\ &nil : trees, \\ &cons : tree \times trees \to trees\}) \end{aligned}$$

[10, Sec. 2.4]

A signature $\Sigma = (S, F)$ induces a family of *Σ-terms* defined in definition 2.4.6: these correspond to "normal" concrete data types. For example, the terms for the *Nat* or *List*(X) signatures can be used as expressions representing natural numbers or lists respectively, and can be visualized using trees just as in figure 2.5 and figure 2.4.

A *Σ-algebra* for a signature Σ as defined in definition 2.4.7 is the interpretation of the abstract data type defined by Σ into concrete values. For each sort in the signature, there is a *carrier set* in the Σ-algebra, which can be regarded as the result type of interpreting operations with the corresponding sort as range.

Definition 2.4.6: Σ-Terms

Let $\Sigma = (S, F)$ be a constructive signature. The family of *Σ-terms* $T_\Sigma = (T_{\Sigma,s})_{s \in S}$ is inductively defined for each sort $s \in S$ as the smallest set constructed by the following rules:

- $f : s \in F \implies f \in T_{\Sigma,s}$
- $f : d_1 \times \ldots \times d_n \to s \in F$ and t_i is *compatible* to $d_i \implies f(t_1, \ldots t_n) \in T_{\Sigma,s}$

  We call a term t compatible to a sort s if $t \in T_{\Sigma,s}$ and compatible to a set S if $t \in S$.

T_Σ is therefore the set of all terms that can be constructed from Σ.

Definition 2.4.7: Σ-Algebras

Let $\Sigma = (S, F)$ be a constructive signature.

A *Σ-algebra* $\mathscr{A} = (A, Op)$ consists of:

- a family of *carrier sets* $A = (A_s)_{s \in S}$ (one for each sort in S);
- a set of functions (also called operations) $Op = \{f_1^{\mathscr{A}} : A_{e_1} \to A_{e'_1}, \ldots, f_m^{\mathscr{A}} : A_{e_m} \to A_{e'_m}\}$ (one for each operation in Σ), where for any e, A_e is e with all sorts replaced by their respective carrier sets from A.

Example 2.4.8 shows some algebras.

Example 2.4.8: Σ-Algebras

The *Nat*-algebra $\mathbb{N}$ with the carrier set $\mathbb{N}$ has the operations

$$zero^{\mathbb{N}} : \mathbb{N}, succ : \mathbb{N} \to \mathbb{N}$$

defined as

$$\begin{aligned} zero^{\mathbb{N}} &= 0, \\ succ^{\mathbb{N}}(n) &= n + 1. \end{aligned}$$

The *List*($\mathbb{N}$)-algebra *Sum* with the carrier set $\mathbb{N}$ has the operations

$$empty^{Sum} : \mathbb{N}, succ^{Sum} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$

defined as

$$\begin{aligned} empty^{Sum} &= 0, \\ cons^{Sum}(n, m) &= n + m. \end{aligned}$$

The $List(X)$-algebra $Length$ with the carrier set $\mathbb{N}$ has the operations

$$empty^{Length} : \mathbb{N}, succ^{Length} : X \to \mathbb{N} \to \mathbb{N}$$

defined as

$$\begin{aligned} empty^{Length} &= 0, \\ cons^{Length}(x, n) &= n + 1. \end{aligned}$$

[10, Sec. 2.6]

Looking at the Nat or $List(X)$ signatures, we see that the operation types in signatures correspond to the argument types of the fold function as we used in this chapter. Algebras collect a set of these arguments. Σ-terms correspond to constructor expressions. Using this, we formulate a general definition for the fold functions on Σ-terms of a signature in definition 2.4.9 that work by replacing the constructors in a term with the corresponding operations, in this case taken from an algebra.

Definition 2.4.9: Fold Function

Let $\Sigma = (S, F)$ be a constructive signature and $\mathscr{A} = (A, Op)$ be a Σ-algebra.

The family of fold functions over $\mathscr{A}$ $(fold_{\mathscr{A},s} : T_{\Sigma,s} \to A_s)_{(s \in S)}$ (that is, there is one fold function per sort) is defined for each sort $s \in S$ and each signature operation $f : d_1 \times \ldots \times d_n \to s$ as follows:

$$fold_{\mathscr{A},s}(f(p_1, \ldots, p_n)) = A_f(fold_{\mathscr{A},d_1}(p_1), \ldots, fold_{\mathscr{A},d_n}(p_n))$$

(p_i that are from sets and not from sorts are taken over unchanged into A_f instead of being folded.)

These generalized fold functions over algebras behave similarly to the specific ones. For example, $fold_{Sum,list}$ is equivalent to the function `sum` implemented with `foldList` in example 2.4.2. An example application of `sum` can be seen in figure 2.4.

The implementation of signatures, algebras and generalized folds in the functional programming language Haskell will be explained in section 3.4.

2.5 Custom Data Types and Eliminators

While λ_Π allows us to formulate functions and types, it does not provide a way to specify new types. It also does not allow recursivity, pattern matching or other techniques used in programming languages to enable writing actual computations on data. For that reason, Löh, McBride and Swierstra manually extend their abstract syntax in [1] with some basic types, so that values and computations are abstractly respresented and can be handled in the evaluator backend without having to resort to a base calculus that allows this intrinsically. We follow the same approach, but extend it to work for arbitrary user-specified algebraic types instead of only a few pre-supplied ones.

With algebraic types as explained in section 2.4 we can model data structures and notate them using constructors. We use a simple monomorphic variant of algebraic types that requires no change in the core calculus language by using named free variables for types and constructors. These are left unchanged during type checking which is exactly what we want. For type checking, a type X is introduced in the type context as $X : *$, and for each constructor C with the parameter types $p_1, ..., p_n$ we add $C : p_1 \to ... \to p_n \to X$.

Take the natural numbers for example. Simply adding the bindings $Nat : *$, $zero : Nat$ and $succ : \Pi n : Nat.Nat$ to the context before type checking allows us to formulate natural number terms like $succ(succ(zero))$ and type check them.

To provide a means of performing computations on these constructor terms we opt for an approach based on fold functions as introduced in section 2.4. A folding operation is added to our core language that folds a term according to operations.

However, for a dependently typed language the fold functions we used until now lack an important feature: the ability for the output type to depend on the input value. To remedy that, we expand the concept of folds to allow exactly that. We call these dependently typed fold functions *eliminators*.

As an example, we take a look at the natural numbers once again. As a reminder, the type of a normal fold function for natural numbers `Nat` looks like this:

```
foldNat :: a → (a → a) → Nat → a
```

We have a type variable `a` that is the result type, a constant value of that type that is filled in for `Zero`, and a function that is filled in for `Succ`.

The type of a dependently typed eliminator for `Nat` in Haskell-like pseudocode like in section 2.2 could be written as follows:

```
elimNat :: (motive :: Nat → *)
        → motive Zero
        → ((n :: Nat) → motive n → motive (Succ n))
        → Nat
        → motive input
```

Instead of a simple type variable, the eliminator needs a function that maps any input value to the elimination's output type for that input. We call that function of the type `Nat → *` the *motive* [1, Ch. 4]. The motive gives the result type of the elimination for a specific input. Because of that the value for `Zero` has the type `motive Zero`: it must be of the output type for the input `Zero`.

The function for `Succ` has changed as well: It now takes two arguments. The value inside the `Succ` is passed over both in its original form *and* in its already eliminated form. The original form is needed so the motive can be used to compute the types of the eliminated inner value and the elimination result.

Lastly, the eliminator takes the actual input of the type `Nat` and returns a value of the type `motive input`.

Eliminators extend folds. A folding like `foldNat False not` for a function that checks whether a number is even can also be expressed by means of an elimination. As motive a constant function returning the output type is used and the unchanged versions of the recursive results in the operations are ignored. For example: `elimNat (const Bool) False (const not)`.

Generalized Eliminators

Just like folds, eliminators can be generalized for all algebraic data types. The types of the parts of an eliminator for a data type X can be derived as follows:

- The motive has the type $X \to *$.
- For each constructor C with the parameter types $p_1, ..., p_n$ the eliminator has an operation parameter.

  For each p_i that is not X, the operation requires an argument of the type p_i. For each p_i that is X, the operation takes two arguments, one of the type X and one of the type $motive(X)$.
- The input has the type X.

- The output has the type $motive(X)$.

Evaluation of an eliminator is done by checking what constructor the input is and applying the fitting operation to the recursively evaluated arguments, if applicable. The evaluation rules used for that are shown in definition 2.5.1.

Definition 2.5.1: Eliminator Evaluation

For each data type X we add the following rule:

$$\frac{}{X \Downarrow X} \qquad \frac{x \Downarrow v}{elimX\ m\ o_1\ \ldots\ o_n\ x \Downarrow elimX\ m\ o_1\ \ldots\ o_n\ v}$$

For each constructor C_i from X with the parameters $p_1, \ldots, p_n$ we add the following rules:

$$\frac{x_1 \Downarrow y_1 \qquad \ldots \qquad x_n \Downarrow y_n}{C_i\ x_1\ \ldots\ x_n \Downarrow C_i\ y_1\ \ldots\ y_n}$$

$$\frac{x \Downarrow C_i\ x_1\ \ldots\ x_n \qquad op_i\ x_1\ (elimX\ m\ o_1\ \ldots\ o_n\ x_1)\ \ldots\ x_n\ (elimX\ m\ o_1\ \ldots\ o_n\ x_n) \Downarrow v}{elimX\ m\ o_1\ \ldots\ o_n\ x \Downarrow v}$$

(for x_1 that are not of the type X, the eliminator call is left out.)

Based on [1, Fig. 13]

3 Methods

In this chapter the various methods used in the implementation of the language will be presented, explained and motivated, starting with the programming language Haskell and the libraries Megaparsec for parsing and Haskeline for interactive shells. We will then show how to implement signatures, algebras and folds from section 2.4 and close this chapter by explaining how Lightfold realizes custom types and dependently typed computations on them.

3.1 Haskell

Haskell[1] is our programming language of choice for the implementation of *Lightfold*. Haskell is purely functional, which lends itself for implementing the algebraic folding approach to compiler construction as will be shown in section 3.4. It also allows monadic parsing that will be presented in section 3.2.

We will use Haskell together with the build tool *Stack*[2] for convenient dependency management and deterministic builds. Instead of the default Haskell prelude we use the module `ClassyPrelude`[3] that removes partial functions and employs more modern data structures, for example the optimized `Text` type[4] for strings instead of the default `String`.

[1]https://www.haskell.org/
[2]https://docs.haskellstack.org/en/stable/README/
[3]https://hackage.haskell.org/package/classy-prelude
[4]https://hackage.haskell.org/package/text

3.2 Parsing

Parsing is the process of turning input strings into structured data for further processing. For parsing we utilize the Haskell library *Megaparsec*[5]. Megaparsec is based on the concept of *monadic parsing*, where parsers are monads with the parsing output as monadic result. Monadic operations and parser combinators can then be used to combine basic parsers into parsers for complex languages.

For example, a basic Megaparsec parser is `char :: Char -> Parser Char` that parses a single character and returns it as monadic result in the `Parser` monad. In the parser `ab :: Parser String` from example 3.2.1 we see how simple parsers like `char` can be combined into more advanced ones: The alternative combinator

`(<|>) :: Parser a -> Parser a -> Parser a`

is used to allow both a's and b's. The combinator `many :: Parser a -> Parser [a]` then uses that to allow an arbitrary amount of a's and b's and returns them in a list.

The monadic usage of Megaparsec parsers is also shown in example 3.2.1 with the parser `ab3 :: (String, String, String)`: We use the do-notation to parse three ab-strings interspersed with commas and close with the parser `eof :: Parser ()` which expects the end of the input. The ab-strings are bound to variables and returned as the final monadic result in a tuple.

Example 3.2.1: Megaparsec Code

```haskell
ab :: Parser String
ab = many (char 'a' <|> char 'b')

ab3 :: Parser (String, String, String)
ab3 = do
    str1 <- ab
    _ <- char ','
    str2 <- ab
    _ <- char ','
    str3 <- ab
    eof
    return (str1, str2, str3)
```

[5]https://hackage.haskell.org/package/megaparsec

In example 3.2.2 some strings and their parse result with `ab3` can be found.

Example 3.2.2: Megaparsec Usage

Some inputs and their parse result when parsed with `ab3`:

- `"ba,bbb,ba"` $\rightsquigarrow$ `("ba","bbb","ba")`
- `"a,b,"` $\rightsquigarrow$ `("a","b","")`
- `",,"` $\rightsquigarrow$ `("","","")`
- `"ab,ab"` $\rightsquigarrow$ error: unexpected end of input
- `"ab,ac,ab"` $\rightsquigarrow$ error: unexpected ‘c’

3.3 Shell

A language shell, also known as *read-eval-print loop (REPL)* is an interactive shell interface to the language: the user enters a term and is presented with the type and evaluated result of that term, or an error message. This allows testing of the language without having to edit and run a source file multiple times.

The shell for lightfold will be implemented using the *Haskeline* library[6]. Haskeline allows easy implementation of shell-like interfaces providing features like line-editing, history and more.

When executed, the example code from example 3.3.1 presents the user with the prompt `echo>` using `getInputLine` inside `loop`. When the user presses enter without having entered any text, the program is terminated using `return ()`. On the other hand, if the user has entered some text, the program will output “You entered:” and the input. Afterwards `loop` calls itself, thus starting over and prompting the user for input.

Example 3.3.1: Haskeline Usage

A simple echo loop can be implemented using Haskeline as follows:

```
main :: IO ()
main = runInputT defaultSettings loop where
    loop :: InputT IO ()
```

[6]https://hackage.haskell.org/package/haskeline

```haskell
    loop = do
        input <- getInputLine "echo> "
        case input of
            Nothing → return ()
            Just input → outputStrLn ("You entered: " ++ input) >> loop
```

An example usage of this echo shell can look like this:

```
echo> Hello World!
You entered: Hello World!
echo> Goodbye.
You entered: Goodbye.
echo>
```

For Lightfold we use Haskeline to develop a shell frontend that allows the user to give Lightfold terms that are then parsed and returned after evaluation.

3.4 Algebraic Modeling

We introduced the theory behind signatures, algebras and folds in section 2.4. We implement this formalization of general recursive data types and computations on them using Haskell.

The signature of an abstract data type is implemented using a polymorphic data type with a single constructor as shown in definition 3.4.1.

Definition 3.4.1: Signatures in Haskell

Let $\Sigma = (S = \{s_1, \ldots, s_m\}, F = \{f_1 : e_1 \to e'_1, \ldots, f_m : e_m \to e'_m\})$ be a signature with each $e_i = d_{i,1}, \ldots, d_{i,k_i}$. Σ can be implemented in Haskell as follows:

```haskell
data Sigma s1 ... sn = Sigma
    { f1 :: d1_1 → ... → d1_k1 → e'1
    , ...
    , fm :: dm_1 → ... → d1_km → e'm
    }
```

Instances of this data type Sigma are Σ-algebras.

For each sort s_i with $f_1, ..., f_j \in F$ as all operation types with s_i as range, the corresponding Σ-terms T_{Σ,s_i} can be implemented using Haskell types as well:

```haskell
data Si = F1 d1_1 … d1_k1 | … | Fj dj_1 … dj_kj
```

The fold functions are implemented for each sort $s_i \in S$ and each signature operation $f : d_1 \times ... \times d_n \to s_i$ like this:

```haskell
foldS :: Sigma s1 … sm → Si → si
foldS alg (F p1 … pn)
  = (f alg) (foldD1 alg p1) … (foldDN alg pn)
[…]
```

(p_i that are from sets and not from sorts are filled in unchanged as argument for `(f alg)` instead of being folded.)

Adapted from [10, Ch. 9].

The examples 3.4.2 and 3.4.3 show how this method looks in practice by means of the signatures Nat and $List(X)$ introduced in example 2.4.5.

Example 3.4.2: Signature Nat in Haskell

The signature for natural numbers Nat:

$$\begin{aligned} Nat = (S = \{nat\}, F = \{&zero : nat, \\ &succ : nat \to nat\}) \end{aligned}$$

can be implemented like this:

```haskell
data NatAlg nat
  = NatAlg {zero :: nat, succ :: nat → nat}
```

Its terms and an algebra for those:

```haskell
data Nat = Zero | Succ Nat
```

The corresponding fold function:

```haskell
foldNat :: NatAlg nat → Nat → nat
foldNat alg Zero = (zero alg)
foldNat alg (Succ n) = (succ alg) (foldNat alg n)
```

An algebra with the carrier Int for translating Nats into native Haskell integers:

```
algebraInt :: NatAlg
algebraInt = NatAlg {zero = 0, succ = (+1)}
```

Example 3.4.3: Signature List(X) in Haskell

List(X), the signature for lists over X:

$$\begin{aligned} \mathit{List}(X) = (S = \{\mathit{list}\}, F = \{&\mathit{empty} : \mathit{list}, \\ &\mathit{cons} : X \times \mathit{list} \to \mathit{list}\}) \end{aligned}$$

can be implemented like this:

```
data ListAlg x list
  = ListAlg {empty :: list, cons :: x → list → list}
```

Its terms are the native Haskell lists [x] and an algebra for those:

```
data [x] = [] | x : [x]

algebraList :: ListAlg x [x]
algebraList = ListAlg {empty = [], cons = (:)}
```

The corresponding fold function:

```
foldList :: ListAlg x list → [x] → list
foldList alg [] = (empty alg)
foldList alg (x:xs) = (cons alg) x (foldList alg xs)
```

An algebra with the carrier Int for translating Nats into native Haskell integers:

```
algebraSum :: ListAlg Int Int
algebraSum = ListAlg {empty = 0, cons = (+)}

algebraLength :: ListAlg x Int
algebraLength = ListAlg {empty = 0, cons = \x xs → xs + 1}
```

Apart from using the fold functions, abstract data types can also be used in functions directly: Any function that would normally output a term of the abstract data type, for example lists or natural numbers, can be changed to take an algebra and use its operations instead of the constructors. This allows to employ the interpretation of an abstract data type provided by an algebra without ever constructing the actual terms. For example, a function

```
dividers :: Int → [Int]
```

that computes all dividers of a number can be rewritten to

```
dividersA :: ListAlg Int list → Int → list
```

`dividersA algebraList` then returns normal lists like the original function, and `dividers algebraLength` calculates the amount of dividers.

We see that abstract data types and fold functions allow us to quickly repurpose functions and data by only exchanging the algebra defining the desired interpretation of the data type.

4 Design

In this chapter we introduce and explain the design as well as the design decisions of Lightfold. We start by setting the design goals and explaining the basic architecture of Lightfold and its internal representations. We continue by presenting the core language for Lightfold expressions and their surface syntax, and finish by describing how Lightfold programs are structured and introducing their syntax.

4.1 Design Goals

Lightfold is supposed to be a proof-of-concept to demonstrate dependent typing and algebraic compilation techniques. Therefore both the implementation and the usage experience aim to be simple and comprehensible as primary design goal. Another objective to keep Lightfold easily extensible for future development. Furthermore we aim for syntax and semantics that allow concise, readable, repetition-free and safe code.

4.2 Architecture

Lightfold programs are processed in multiple steps.

Parsing is the step of reading Lightfold source code and structuring it into an abstract syntax tree algebra (LightfoldAST).
Compilation to LightfoldCore is performed using an AST-to-LightfoldCore-algebra.
Type checking is done on LightfoldCore terms.
Evaluation is done on type checked LightfoldCore terms.

The two internal representations, LightfoldAST and LightfoldCore, are defined as abstract data types using signatures with Σ-terms and fold functions as introduced in section 2.4. Parsing is done directly into an LightfoldAST-algebra as explained in section 3.4. Using an algebra targeting LightfoldCore, we get a LightfoldCore program.

LightfoldCore as the main internal representation of Lightfold programs is closely based on the dependently typed lambda calculus λ_Π introduced with its rules for evaluation and typing in section 2.2. We use a core language close to the base calculus to allow for easier application of the various theoretical methods from chapter 2.

Type checking and evaluation is performed on LightfoldCore terms using recursive algorithms based on the typing and evaluation rules. That shows another advantage of using a small core language: It keeps Lightfold extensible. Features can be added without changes to the core part of the language, as long as they can be translated into LightfoldCore.

Note that while both LightfoldAST and LightfoldCore are defined through signatures as abstract data types and used as such, in the following we will only show the concrete Σ-term types for easier understanding and brevity.

4.3 Terms

The most important part of Lightfold are terms: They are needed to represent values and types. LightfoldCore's abstract syntax for terms in listing 4.3.1 is closely based on the abstract syntax of λ_Π shown in section 2.2.

Since our type system for λ_Π uses the bidirectional typing from section 2.1, we follow the same approach as Löh, McBride and Swierstra in [1] and introduce separate types for representation of checkable (`TermCheck`) and inferable terms (`TermInfer`) for improved type safety: The type inferrer is able to ensure it gets an inferable term as input on the type level and vice versa.

Listing 4.3.1: LighfoldCore Terms Abstract Syntax

This is the abstract syntax of LightfoldCore terms defined as Haskell data types.

```
data TermCheck
    = Infer TermInfer
    | Lambda TermCheck

data TermInfer
```

```
    = Annotation TermCheck TermCheck
    | Star
    | Pi TermCheck TermCheck
    | Bound Int
    | Free Name
    | Application TermInfer TermCheck
    | Elimination Text TermCheck [TermCheck] TermCheck

data Name = Global Text | Local Int
```

Applications are represented as pairs of a function and an argument, functions with multiple arguments are nested. LightfoldCore employs the representation of bound variables using de Bruijn indices introduced in section 2.3. Bound and free variables have separate constructors: bound variables have an integer for the de Bruijn index in the `Bound` constructor, free variables have either a de Bruijn index with the `Name` constructor `Local` if it is a variable pointing to a binder outside of the current scope or it can reference a value definition by name with `Global`. Data types and their constructors are represented using named free variables as explained in section 2.5.

Because binders are referenced using de Bruijn indices, the `Lambda` and `Pi` constructors do not need an identifier string.

Eliminations from section 2.5 have their own constructor. An elimination has a text with the name of the type that is being eliminated. The first argument term is the motive, the list of terms contains the operations, there should be one for each constructor of the type. The last argument term is the elimination input.

Surface Syntax

Lightfold's surface syntax for terms is based on the following principles:

- Anything between /* and */ or following a // in the same line is regarded as comment and ignored. This comment syntax is the same as in many languages, including C, C++ and Java.
- As identifiers any strings consisting of numbers and lower-case or upper-case latin letters are allowed as long as the first character is a letter and no number.

- Named variables are used instead of de Bruijn indices. While de Bruijn indices are handy for internal representation, named variables are better suited for the surface syntax as they allow for meaningful names to be used and do not have to be changed when adding or removing binders.

- Type annotations are written using a single colon, for example `x : Nat`, which is concise and what is used in mathematics. This syntax is, among others, used by Idris. Haskell, that we previously used as base for pseudocode, uses two colons.

- As common in functional programming languages like Haskell, function application does not require parentheses to keep it as terse as possible: a blank is enough, for example: `f x`. In λ_Π and LightfoldCore application is always between exactly one function and one argument, so a chain of applications like `f x y z` is actually nested: `((f x) y) z`. However, in our syntax, we model application chains as a simple repetition. The conversion into nested one-on-one applications happens during the compilation to LightfoldCore.

- Parentheses, as usual, can be used to denote what belongs to what in nested terms.

- Lambda terms are written with a double arrow ⇒ between argument and result, for example: `x ⇒ f x`. This is the same syntax that C#[1] and JavaScript[2] use for lambda terms.

  Some imperative languages like Java[3] use a single arrow for lambdas: `x → f(x)`. Haskell[4] uses a single arrow together with a backslash symbolizing a λ, for example `\x → f x`, Python[5] uses “lambda” and a colon: `lambda x: f(x)`.

  However, for Lightfold we decided on the double arrow syntax from C#, because it is both as concise as Java’s syntax and easy to visually distinguish from function terms like Python’s syntax and unlike Java’s and Haskell’s syntax.

- Function types are denoted by an identifier, a colon and the range type in parantheses, followed by a single arrow and the domain type, for instance `(n : Nat) → Nat`. This fits well with the annotation syntax, as `n` indeed has the type `Nat` when bound. The single arrow for function types is commonly used in functional programming

[1]https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/lambda-operator#lambda-operator
[2]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions
[3]https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
[4]https://wiki.haskell.org/Anonymous_function
[5]https://docs.python.org/3/tutorial/controlflow.html#lambda-expressions

languages like Haskell, and the dependently typed language Idris uses this syntax as well[6].

That both lambdas and function types use arrows also symbolizes their similarities: Both bind a value to the identifier written before the arrow in the following term.

- Variables are simply denoted by their identifier. The identifier `_` is special: It can be used as identifier in binders but causes an error when used as variable.

  Using `_` in binders for inputs therefore ensures and shows intuitively that the variable bound will not be used. This behavior is inspired by Haskell.

- The keyword `Type` is used to denote the type of types which is processed to `Star` in LightfoldCore.

- Lastly, the keyword `elim` is processed as elimination. It expects an identifier with the type to be eliminated and some arguments. The first argument is processed as motive, the last one as elimination input and all in between are processed as the operations.

These rules lead to the formal syntax specified in definition 4.3.2.

Definition 4.3.2: Lightfold Term Syntax

The syntax for terms is defined using Backus-Naur form. x^* means that x can occur zero or more times, x^+ means that x must occur at least once.

$$\begin{aligned} \mathit{term} &::= \mathit{termInner} : \mathit{term} \\ &\mid \mathit{termInner}^+ \end{aligned}$$

$$\begin{aligned} \mathit{termInner} &::= \texttt{Type} \\ &\mid \texttt{elim}\ \mathit{identifier}\ \mathit{termInner}^* \\ &\mid (\ \mathit{term}\ ) \\ &\mid \mathit{identifier} \Rightarrow \mathit{term} \\ &\mid (\ \mathit{identifier} : \mathit{term}\ ) \rightarrow \mathit{term} \\ &\mid \mathit{identifier} \end{aligned}$$

[6] https://docs.idris-lang.org/en/latest/reference/syntax-guide.html#dependent-functions

We use two non-terminals in the grammar, *term* and *termInner*. We do that to allow the grammar to be parsed in a left-to-right fashion. For example, without the separation the rule for annotations would look like this: $term ::= term : term$. This is a rule for the non-terminal *term* that has *term* itself as first symbol: it is left-recursive [11]. This is problematic, as a left-to-right parser would descend into an endless loop when trying to parse a term, because when parsing a term, it would try to parse a term next, and so on. By separating terms into normal and inner terms, we can circumvent this problem.

Note that a single *termInner* can be embedded in a *term* through *term*'s second rule: $termInner^+$ also allows for a single repetition.

4.4 Programs

The general structure of a Lightfold program is inspired by functional programming languages like Haskell: It consists of an arbitrary amount of *definitions*, either defining a value or a type. Definitions can access other definitions, allowing the user to compose values and types from other values and types. As the base calculus λ_Π does not include recursion though, a definition can only access previous definitions, except for data type definitions that can access themselves to allow recursive types as explained in section 2.5. The types in LightfoldCore for structuring a program are shown in listing 4.4.1.

Listing 4.4.1: LightfoldCore Abstract Syntax

```
data Lightfold = Lightfold [DefType] [DefValue]

data DefType
    = DefType
        Text
        [DefConstructor]

data DefConstructor
    = DefConstructor
        Text
        [Maybe Text]

data DefValue = DefValue Text TermCheck TermCheck
```

As can be seen, a Lightfold program consists of some type definitions and some value definitions. A type definition has a name and some constructors, and a constructor consists of a name and and a list of parameters. A parameter is represented by the type `Maybe Text`: We use `Nothing` to denote a recursive parameter, that is, a parameter that has the type that is currently being defined. `Just` is used together with an type identifier to denote a parameter of another type.

On the other hand, a value definition consist of a name, a type term and a value term. To allow a Lightfold program to be run, a value named `main` must be defined. When executed, the program will output the result of evaluating `main`.

Surface Syntax

Using a value definition a term with a type is bound to an identifier. As syntax we use the identifier and the type separated by a colon in the first line. Indented in the next line follows the actual term, for example:

```
id : (n : Nat) -> Nat
    n => n
```

This syntax deviates from the ones used in Haskell, Idris and others in that the value is indented (inspired by Python) to designate that it belongs to the declaration above instead of requiring the user to write the value name again, following the design goal to avoid unnecessary repetition.

Type definitions look similar to value definitions. They start with the type identifier followed by `: Type`, declaring the new type as a type. The constructors with their parameter types follow, each indented in their own line, for example:

```
Nat : Type
    zero
    succ Nat
```

Note that while identifiers are case-sensitive, Lightfold does not impose any restrictions on the case of the identifiers used in definitions, binders and variables. However, we generally use upper-case identifiers for types and lower-case identifiers in all other occasions.

The resulting formal syntax rules are shown in definition 4.4.2.

Definition 4.4.2: Lightfold Syntax

This is the syntax of Lightfold specified in Backus-Naur form. x^* means that x can occur zero or more times, *indent* denotes an indentation (four or more spaces), *newline* a line break.

$$\mathit{lightfold} ::= \mathit{def}^*$$

$$\begin{aligned} \mathit{def} ::= {} & \mathit{identifier} : \texttt{Type}\ \mathit{newline}\ \mathit{defConstructor}^* \\ \mid {} & \mathit{identifier} : \mathit{term}\ \mathit{newline}\ \mathit{indent}\ \mathit{term}\ \mathit{newline} \end{aligned}$$

$$\mathit{defConstructor} ::= \mathit{indent}\ \mathit{identifier}\ \mathit{identifier}^*\ \mathit{newline}$$

Example 4.4.3 shows a syntactically valid Lightfold program using all introduced syntax constructs.

Example 4.4.3: Lightfold Syntax

```
Nat : Type
    zero
    succ Nat

id : (a : Type) -> (_ : a) -> a
    a => x => x

plus : (_ : Nat) -> (_ : Nat) -> Nat
    x => y => elim Nat (_ => Nat) x (_ => n => succ n) y

main : Nat
    ((id Nat) : (_ : Nat) -> Nat) (succ zero)
```

5 Implementation

Lightfold is implemented in the functional programming language Haskell, using the build tool Stack, as introduced in section 3.1. The implementation exists in a single source code repository that consists of multiple parts. The common Lightfold library is the main part and can be found in the `src` folder. It provides data types, parsers, compilers, the evaluator and the type checker.

There are two executables. There is the interpreter `lightfold-run` in the folder `run`, that can read files and outputs the evaluated result of the `main` definition in the file, and the interactive shell `lightfold-shell`, found in `shell`, that provides a command-line interface to try out Lightfold terms, optionally loading the definitions of a file beforehand.

The implementation generally aims to be well documented and concise while maintaining readability and comprehensibility.

The source code is attached to this thesis as an archive file. It can also be found online at https://git.eisfunke.com/software/lightfold/lightfold/-/tree/thesis (or in the always up to date version at https://git.eisfunke.com/software/lightfold/lightfold).

Note that the listings shown in the following are pseudocode and not direct copies of the actual implemention. Comments, error handling and other technical details are omitted from the original code for brevity and legibility when not required for understanding. The respective file that contains the original and complete code is noted at the beginning of each section.

5.1 Abstract Syntax Tree

See `src/Lightfold/AST.hs`

The abstract syntax tree used for parsing is implemented via a signature as explained in section 3.4. The signature can be seen in listing 5.1.1. For brevity, we only show the subsignature for Lightfold terms.

Listing 5.1.1: Term Abstract Syntax Tree Signature

```
data AlgebraTerm term termInner = AlgebraTerm {
    algebraAnnotation :: termInner → term → term,
    algebraApplication :: NonEmpty termInner → term,

    algebraSubterm :: term → termInner,
    algebraLambda :: Text → term → termInner,
    algebraPi :: Text → term → term → termInner,
    algebraVariable :: Text → termInner
}
```

This signature corresponds to the grammar for terms from definition 4.3.2. There are two sorts, *term* and *termInner*: one for each non-terminal in the grammar. For every rule, there is an operation with its range corresponding to the non-terminals in the right-hand rule side. Identifiers are modeled using `Text` values. `NonEmpty` in the operation `algebraApplication` is the type of non-empty lists and models that an application chain requires at least one element (a single element simply embeds a `termInner` into a `term` without actual application).

5.2 Parser

See `src/Lightfold/Parser.hs`

The parser's task is to read input according to the Lightfold grammar as specified in definition 4.3.2 and structure it into the algebraic signature for the abstract syntax tree presented in section 5.1. As explained in section 3.2, the Haskell library Megaparsec is used to implement the parser monadically.

We explore the parser implementation by taking the example of the parser for terms, shown in listing 5.2.1, as terms are the most important part of a program.

In our grammar we had two non-terminals for terms: `term` and `termInner`, there is a parsing function for each of those. The functions take an algebra for terms as argument (that is, an instance of the signature data type from listing 5.1.1). The operations from that algebra are mapped to the parsing results using applicative operators[1] according to the grammar.

For example, a lambda term has to be parsed into the operation `algebraLambda :: Text → term → termInner`. A lambda expression in Lightfold consists of an identifier, parsed with the parser `identifier`, an arrow ⇒ parsed with `arrowDouble` and a term, parsed with `term alg` (the algebra `alg` used for parsing is handed over to the term parser). The results of these sub-parsers are mapped onto `algebraLambda`. `arrowDouble` is an exception, as its result is always ⇒ and does not have to be saved. Its result is instead ignored using `<*`.

Multiple alternatives for parsing are concatenated using the alternative operator `<|>`. Note that for alternatives their order does matter: The parser for variables, parsing just an identifier, must come after the parser for lambdas. It would succeed on an identifier and consume it, even if it belongs to a lambda term. Because of that we have to try parsing an inner term as lambda expression first and only if that does not work we try the variable parser.

Alternative parsers that consume input but might still fail after that are wrapped in `try`, a parser that backtracks to the previous parser state if the inner parser fails.

Listing 5.2.1: Term Parser

```haskell
term :: AlgebraTerm term termInner → Parser term
term alg@AlgebraTerm{..}
    =   (algebraStar <$ symbol "Type")
    <|> try (algebraElimination <$ symbol "elim" <*> identifier
        <*> many (termInner alg))
    <|> try (algebraAnnotation <$> termInner alg <* colon
        <*> term alg)
    <|> (algebraApplication <$> NE.some (termInner alg))

termInner :: AlgebraTerm term termInner → Parser termInner
termInner alg@AlgebraTerm{..}
    =   try (algebraPi <$ symbol "(" <*> identifier <* colon
        <*> term alg <* symbol ")" <* arrow <*> term alg)
    <|> (algebraSubterm <$> roundBrackets (term alg))
    <|> try (algebraLambda <$> identifier <* arrowDouble
```

[1]https://hackage.haskell.org/package/base-4.14.0.0/docs/Control-Applicative.html

```
        <*> term alg)
    <|> (algebraVariable <$> identifier)
```

5.3 Compilation

See `src/Lightfold/AST.hs`

The compiler is a LightfoldAST-algebra with the LightfoldCore types (see chapter 4) as carrier sets. It allows us to directly parse Lightfold code into LightfoldCore programs by using this algebra with the parser functions from the previous section. It can also be used with LightfoldAST's fold functions. Again, we examine only the implementation for terms, shown in listing 5.3.1.

Both `term` and `termInner` have `[Text] → Core.TermCheck` as carrier set. The list of string parameters is the list of known bindings. When passing a binder, we add the name of its variable in front of the list and when reaching a variable, we look up its name in the list. If the name is in the list, we know the de Bruijn index to be used for LightfoldCore through the current index in the list.

Applications have to be transformed from LightfoldAST's representation of application chains as non-empty lists into the nested one-on-one representation of LightfoldCore as mentioned in section 4.3. For that, `curryApplication` (shown in listing 5.3.2) is used. It always applies the first two arguments of a non-empty list to each other until there is only one left.

Also noteworthy is the operation for eliminations. It separates the argument list of the elimination into the motive (the first argument), the input (the last one) and the operations (all in between).

Listing 5.3.1: AST-Algebra for LightfoldCore

```
algebraTermCore :: AlgebraTerm
                   ([Text] → TermCheck)
                   ([Text] → TermCheck)
algebraTermCore = AlgebraTerm {
    algebraStar = \binds → Infer Star,
    algebraElimination = \ty args → \binds
```

```
        → getElimination ty (($ binds) <$> args),
    algebraAnnotation = \te ty → \binds
        → Infer $ Annotation (te binds) (ty binds),
    algebraApplication = \xs → \binds
        → curryApplication (($ binds) <$> xs),

    algebraSubterm = id,
    algebraLambda = \var term → \binds
        → Lambda $ term (var : binds),
    algebraPi = \var domain range → \binds
        → Infer
            $ Pi (domain binds) (range (var : binds)),
    algebraVariable = \var → \binds → case var of
        "_" → error "_ can't be used as variable"
        _ → case var `elemIndex` binds of
            Just i → Infer $ Bound i
            _ → Infer $ Free $ Global var
}
```

Listing 5.3.2: curryApplication

```
curryApplication :: NonEmpty TermCheck → TermCheck
curryApplication (x :| []) = x
curryApplication (Infer f :| x:xs) = curryApplication
  $ Infer (Application f x) :| xs
curryApplication _ = undefined
```

5.4 Evaluator

See `src/Lightfold/Evaluator.hs`

After a Lightfold program has been compiled into a LightfoldCore instance, an evaluator is required to be able to type check the program and then compute its result. The evaluator mainly implements the evaluation rules for λ_Π presented and explained in section 2.2.

To recap: These rules each specify the result of *fully* evaluating a term of a specific form to something derived from evaluating parts of the term by themselves. This lends itself to implementation as a recursive function that evaluates terms by recursively calling itself on subterms.

λ_Π's rules can be implemented in a syntax-directed way: For every possible construct of a λ_Π term (see definition 2.2.1) there is only one corresponding rule, except for applications with the rules (E-App) and (E-AppTrans). The former applies an argument to a lambda function using β-reduction, the latter simply evaluates both sides of the application.

However, for these rules the decision which one to apply can be made through looking at the syntax of the evaluated function side: If it is a function term, we perform β-reduction with (E-App), otherwise we apply (E-AppTrans). The evaluator can therefore use Haskell's pattern matching to choose between rules to apply.

The implementation of the evaluator is oriented after [1].

Substitution

The evaluation rules cannot be turned into an algorithm for LightfoldCore evaluation directly. As LightfoldCore uses de Bruijn indices for variable representation, we need to pay attention to the problems with substitution described in section 2.3. Variables referencing a specific binder have different indices depending on their position. Variable indices in terms being filled in have to be incremented when passing further binders, and when removing a binder, variables inside it pointing outwards have to be decremented.

Another problem is that in LightfoldCore's bidirectional type system (see sections 2.1 and 4.3) straightforward substitution of variables with terms is impossible. Variables are checked in inference mode (they have the type `TermInfer`) while function arguments are not generally inferable and therefore have the type `TermCheck`. This type mismatch prevents us from simply filling in function arguments.

Both substitution problems are solved by distinguishing bound from free variables and carrying an *environment* with us while evaluating terms. An environment contains the known bindings of variable names to their values. The code excerpts relevant to substitution can be found in listing 5.4.1.

Listing 5.4.1: Evaluator: Substitution

```
evalCheck env t = case t of
  [...]
  Lambda term → Lambda $ substituteFree
    (evalCheck (incrementEnv env) (substituteBound term))

evalInfer env@(Env free defTypes) t = case t of
  [...]
  Pi input output → Infer $ Pi (evalCheck env input)
    (substituteFree $ evalCheck (incrementEnv env)
    (substituteBound output))
  Bound i → error "Unknown bound variable reference"
  Free s → case s `lookup` free of
    Just term → evalCheck env term
    Nothing → Infer $ Free s
  Application f x → case evalInfer env f of
    Lambda term → if hasBound 0 result
      then error "Unreplaced var in reduction"
      else result
      where
        result = decrement $ evalCheck
          ((Local 0, evalCheck env $ increment x)
            : incrementEnv env)
          (substituteBound term)
    Infer term → Infer $ Application term (evalCheck env x)
  [...]
```

The `Bound` constructor for bound variables is only used when the referenced binder actually is in the currently used term. When we recursively evaluate a subterm inside a binder, all bound variables referencing that binder are replaced with locally free variables of the same index using the `substituteBound` function. This is possible since both bound and free variables are inferable terms. We also increment all locally free variables and references in the environment using `incrementEnv`.

When applying an argument to a binder we also substitute the bound variables with free variables. Instead of directly filling in the argument we add it to the environment as locally free variable with the index zero. When descending into binders, all free variables in names and terms in the environment are incremented to keep them pointing to the same outside binder. When a free variable is reached during evaluation, its name is looked up in the environment and filled in if it exists. Unlike direct substitution, this is possible as the result of an evaluation is always of the type `TermCheck`, allowing us to fill in the checkable argument value.

When leaving a binder, we replace locally free variables with the index `0` with bound variables again, except when removing the binder after application. In that case we ensure that no free variables with the index zero are left using `hasBound` and decrement all free indexed variables inside the binder being removed with `decrement`.

Bound variables should therefore never be directly encountered by the evaluator. They should have been replaced by locally free variables when the corresponding binder was passed.

Elimination

Another interesting part of evaluation are eliminations. The theory behind eliminators was introduced in section 2.5. Generally, an elimination is evaluated by taking the input, checking which constructor of the type being eliminated it is, and replacing the constructor with the fitting operation while recursively applying the elimination to the constructor's argument.

To that end we check whether the input is a free variable or an application chain beginning with a named free variable. Because constructors are represented as named free variables, we then look up the variable's name in the constructors of the elimination type. We use the resulting index to pick the corresponding operation from the operations list of the elimination.

Using the function `getArgs` (see listing 5.4.2) the parameter list of the constructor definition and the concrete arguments of the constructor being processed are traversed. To recap from section 4: The parameters of a constructor definition are modeled as a list of the type `Maybe Text`. `Nothing` represents a parameter of the constructor's own type. `Just x` is a parameter of the type with the name `x`.

The parameter list from the constructor definition and the argument list of the constructor itself should have the same length because a constructor's arguments have to fit its definition. Type checking assures that.

`getArgs` passes through the two lists one by one, assembling a result list. For each `Just` in the parameter list, the argument from the second list is added directly to the result list. For each `Nothing` (that is, an argument of the type to eliminate), this argument is added to the result twice, once unmodified and once recursively eliminated.

The argument list resulting from `getArgs` is then applied to the previously looked up operation.

This reflects the principles of folding from section 2.4 and elimination from section 2.5. The constructors are recursively replaced by the corresponding operations. The unchanged arguments are also given to the operation.

Listing 5.4.2: Evaluator: Elimination

```
getArgs :: [Maybe Text] → [TermCheck] → [TermCheck]
getArgs [] [] = []
getArgs ((Just _):params) (arg:args)
  = arg : getArgs params args
getArgs (Nothing:params) (arg:args)
  = arg
  : evalInfer env (Elimination elimType
    (evalCheck env motive)
    (evalCheck env <$> ops)
    arg)
  : getArgs params args
```

5.5 Type Checker

See `src/Lightfold/Type.hs`

The type checker is the part of Lightfold's implementation ensuring that all terms have a correct type that fits their annotations. It is based on the typing rules for λ_Π presented in section 2.2.

In λ_Π everything is a term and types can depend on values. Because of that the type checking process needs to evaluate some terms: The type of a function can only be computed after the actual argument value of a call of that function is known, as it is bound inside the type. At that point the domain type has to be evaluated with that bound value.

The type checker is implemented recursively. It consists of two functions, `inferType` for inferring types of inferable terms and `checkType` that checks a checkable term against a type. The results are wrapped in `Either Text` to allow for failure with messages and monadic failure handling.

Type checking uses contexts. In the typing rules, a typing context consisted of bindings of variable names to their types. In the implementation, the context also includes a list of bindings of variable names to values. These bindings contain the previous value definitions, so that they can be used in types. The list of type definitions is also added to the context because to type check eliminators the type checker has to know the definition of the corresponding type.

The implementation of the type checker is illustrated with the case for application shown in listing 5.5.1, because that is the part of the implementation enabling dependent types: When applying an argument to a function, the argument value is bound in the range of the function's type to compute the final return type.

Applications are inferable terms, therefore their type is inferred in the function `inferType`. To infer the type for an application `Application f x` at first the type of the function `f` is inferred. It must be a function type of the form `Pi range domain`, if not, the type inferring fails.

In the next step, the argument `x` is checked against the range type. If that succeeds, `x` is evaluated to `x'`. The next step is where dependent types come in: The domain type is evaluated with the evaluated argument in the evaluation environment (using the same substiution techniques as in the evaluator). This evaluated domain is returned as type for the complete application.

The implementation of the type checker is oriented after [1].

Listing 5.5.1: Type Checker – Application

```
inferType ctx@(Context binds bindVals defTypes) term
  = case term of
    [...]
    Application f x → case inferType ctx f of
      Infer (Pi range domain) → do
        checkType ctx x range
        let x' = evalCheck (Env bindVals defTypes) x
        let domain' = decrement $ evalCheck
```

```
          (Env ((Local 0, increment x') : bindVals) defTypes)
          (substituteBound domain)
        if hasBound0 domain'
          then error "unreplaced"
          else return domain'
      _ → fail "appliction to a non-function"
    [...]
```

5.6 Frontends

See `run/Main.hs` and `shell/Main.hs`

The interpreter frontend reads a Lightfold program from a file specified on the command line. It type checks all definitions in the file and then outputs the result of evaluating the `main` definition.

The other frontend is the interactive shell. It optionally accepts a file as argument when starting, which will then be read, type checked and its definitions provided for use in the shell session.

The interactive shell is implemented using the Haskeline library presented in section 3.3. The user input is parsed as term and type checked as well as evaluated. If applicable, the definitions from the optionally loaded file are used. The shell then prints the type and result of the term.

6 Evaluation

In this chapter we look at examples of the usage of Lightfold via the interpreter and the interactive shell. We then analyze and evaluate Lightfold with regard to dependent typing and the techniques used in the implementation. Finally, we discuss the results and provide an outlook for further development and research.

6.1 Usage

Among other examples, the excerpts of Lightfold code used in this chapter can be found in the `examples` folder in the source code repository. The respective file is noted at the beginning of each section.

Lightfold can be used in two ways. `lightfold-run file` loads a file, type checks it and prints the evaluated result of the `main` definition. It fails if there is no such definition.

`lightfold-shell file` loads a file, type checks it and opens an interactive shell with the definitions from the file available for usage. The file can also be omitted for a shell in a blank environment.

Numbers

See `examples/nat.lightfold`

As a basic example, we implement the natural numbers and some exemplary functions on them in Lightfold. As shown in section 2.4, the type of natural numbers consists of two constructors: one for zero and one for constructing the successor of another natural number. Listing 6.1.1 models this as the type `Nat` in Lightfold.

Using eliminators (see section 2.5) we can implement functions on `Nat`, for instance the addition function `plus`. It takes two `Nat`s as input, `x` and `y`. The addition is performed using an elimination of Nat on x. The motive is a function that always returns Nat: an addition returns a number regardless of the inputs.

The zero operation of the elimination is x: when adding zero to something, it stays unchanged. The succ operation is a function with two arguments: the first one is the previous number and the second one is the previous elimination result. We return the successor of that previous result.

This means that for each succ in y, a succ is added to x, resulting in the addition of x and y.

Listing 6.1.1: Lightfold Natural Numbers

```
Nat : Type
    zero
    succ Nat

plus : (_ : Nat) -> (_ : Nat) -> Nat
    x => y => elim Nat (_ => Nat) x (_ => n => succ n) y
```

The following is an example call of plus in the Lightfold shell:

```
λ> plus (succ zero) (succ (succ zero))
Type: Nat
Term: (succ) ! ((succ) ! ((succ) ! (zero)))
```

This is indeed correct: one plus two equals three, which is a Nat.

Dependent Types

See `examples/dependent.lightfold`

We illustrate the possible usage of dependent types with a simple example in listing 6.1.2.

Listing 6.1.2: Lightfold Dependent Types

```
Bool : Type
    true
    false

ifThenElse : (out : Type) -> (_ : Bool) -> (_ : out) -> (_ : out) -> out
    out => b => x => y => elim Bool (_ => out) x y b

oneOrTrue : (b : Bool) -> ifThenElse Type b Nat Bool
    b => elim Bool (x => ifThenElse Type x Nat Bool) (succ zero) (true) b
```

First the type `Bool` that represents a boolean value is defined: it is either `true` or `false`.

Using that `ifThenElse` can be defined. It requires a `Type` argument named `out` that is the type of the choices and the result. The next arguments are `b : Bool` and two `out` values. `ifThenElse` then returns the first choice if `b` is `true` and the second if `b` is `false`. Two example calls in the Lightfold shell are shown in the following:

```
λ> ifThenElse Nat true (succ zero) zero
Type: Nat
Term: (succ) ! (zero)

λ> ifThenElse Nat false (succ zero) zero
Type: Nat
Term: zero
```

The dependent example function is `oneOrTrue`. Its only argument is of type `Bool`, the return type is `ifThenElse Type b Nat Bool` where `b` is the input value. That means that `oneOrTrue` returns a `Nat` if the input is `true` and a `Bool` otherwise.

The function itself is defined by means of an elimination. This time, the motive is not a constant function, it is the same `ifThenElse` call as in the type declaration. The operation for `true` is `succ zero`, a `Nat`, and the one for `false` is `true`, a `Bool`. An example usage of `oneOrTrue` looks as follows:

```
λ> oneOrTrue true
Type: Nat
Term: (succ) ! (zero)

λ> oneOrTrue false
Type: Bool
Term: true
```

It can be seen that `oneOrTrue` correctly returns a value of a differing type *depending* on the input value.

6.2 Discussion and Outlook

Lightfold allows for the implementation of functions and values using value and type definitions. Defined values can be used in following definitions to compose programs gradually. Writing correct programs is assisted by the type checker. The bidirectional type system requires a relatively small amount of type annotations (see section 2.1). De Bruijn indices enable usage of simple syntactical equality checking for terms and avoid problems with variable naming (see section 2.3).

However, de Bruijn indices and the bidirectional type system introduce notable additional complexity with variable substitution in the implementation of both the evaluator and the type checker (see section 5.4).

Lightfold's goals were to develop a simple to use and extensible proof-of-concept demonstrating dependent typing with concise and understandable code. In chapter 4, the language architecture and syntax is designed with these goals in mind: The usage of a λ_Π-based small core language should allow for easier future additions (see section 4.2). A simple dependently typed program can be written concisely in a few lines of Lightfold code (see example 6.1.2).

Having said this, Lightfold is a relatively low-level language. This induces some compromises regarding simplicity. For example, in polymorphic functions each type argument has to be provided explicitly although they could be inferred [1, Ch. 5]. Data structures like natural numbers and lists can be represented using algebraic data types, but instances of them have to be tediously encoded with all their constructors (see section 6.1). Programs cannot include functions from other files. Rectifying these and other flaws could be a promising future development path for Lightfold, allowing for more convenient usage.

The implementation of Lightfold with abstract data types based on algebraic modeling as shown in section 3.4 allowed for a concise implementation of the compiler logic in an algebra as seen in section 5.3. The parser can parse Lightfold code into any algebra, for example to construct LightfoldCore programs from the source code without ever having to build up a complete abstract syntax tree.

The abstract algebraic compiler infrastructure should allow for easy extension. For example, Lightfold can currently only be interpreted or used in a shell. A compiler backend targeting another language could be added in the future. That should be possible with few modifications of the existing code by writing a LightfoldCore-Algebra targeting the target language.

Proper unit tests and correctness proofs for the implementation of LightfoldCore and its tools could be a worthwile endeavor. Because LightfoldCore is a small core language close to λ_Π proving its compliance with λ_Π 's rules should be feasible with a low amount of work.

The core language system offers opportunities for improvement as well. The current type system is unsound because $*$ has $*$ itself as type (see section 2.2). λ_Π could be exchanged for the sound and more powerful calculus of constructions that constitutes the top of the λ-cube [2].

Lightfold's algebraic data types are only monomorphic. They could be extended to allow type and value variables so that data structures like polymorphic lists can be represented. This would allow for the implementation of *vectors*: lists with their length encoded on the type level. They are a common example of the usage of dependent types, but are currently not representable in Lightfold.

The current system of data types and eliminators is implemented as an external augmentation to the core calculus (see section 2.5). An alternative approach is to encode data in the calculus itself as pure lambda terms, called *lambda encodings*. There is interesting research regarding practical usage of lambda encodings and derivation of eliminators: for example by Stump in [12], demonstrated with the language Cedille[1]. A system like this could be adapted for Lightfold.

It could also be interesting to examine ways of incorporating the advantages of the abstract algebraic data types shown in section 2.4 into the language. The usage in Haskell as done in this thesis (see section 3.4) requires a lot of infrastructure code, like defining the concrete data type for the constructors manually, that could become unnecessary in a language designed with abstract data types in mind.

[1]https://cedille.github.io/

7 Conclusion

This thesis introduced the simply typed lambda calculus $\lambda_{\rightarrow}$ as base of functional programming. Evaluation and typing of $\lambda_{\rightarrow}$ terms was presented, the difference between Church- and Curry-style typing was shown, and bidirectional typing was introduced as an attempt of combining their advantages.

We looked at the dependently typed lambda calculus λ_{Π} with its type and evaluation rules as an extension of $\lambda_{\rightarrow}$. λ_{Π} allows types to depend on terms, for example allowing the output type of function to depend on the input, which can be useful when trying to assign a proper type to a function like `printf`.

De Bruijn indices were presented as a means of representing bound variables in lambda terms without unambiguous indices instead of arbitrary names. We also introduced the theoretical foundation of algebraic modeling, including signatures of abstract data types, Σ-algebras as interpretations of signatures and generalized fold functions. We also showed the theory behind Lightfold's approach to custom data types and computations on them through generalized eliminators.

In the methods chapter, we described the techniques used in the implementation. Haskell as language of choice, Megaparsec as parsing library and Haskeline as library for interactive shells were introduced. We showed how to implement the theoretical methods of algebraic modeling from before.

We looked at Lighfold's design goal, simplicity and conciseness, and the language architecture with the λ_{Π}-based core language LightfoldCore. We introduced the internal representation of the language. The language syntax was explained and justified.

The implementation of Lightfold was introduced, starting with the parsing of terms and their compilation to LightfoldCore. We explained the implementation of the evaluator, type checker and the two frontends: the interactive shell and the interpreter.

Finally we evaluated the results using some usage examples, discussed them and provided an outlook of possible future research work.

References

[1] A. Löh, C. McBride, and W. Swierstra, "A Tutorial Implementation of a Dependently Typed Lambda Calculus," *Fundamenta Informaticae*, vol. 102, no. 2, pp. 177–207, 2010, doi: 10.3233/FI-2010-304. [Online]. Available: https://www.andres-loeh.de/LambdaPi/LambdaPi.pdf. [Accessed: 25-Jun-2020]

[2] H. Barendregt, "Introduction to generalized type systems," *Journal of Functional Programming*, vol. 1, no. 2, 1991, doi: 10.1017/S0956796800020025. [Online]. Available: https://www.researchgate.net/publication/216300104_An_Introduction_to_Generalized_Type_Systems. [Accessed: 26-Jun-2020]

[3] H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini, "A filter lambda model and the completeness of type assignment," *Journal of Symbolic Logic*, vol. 48, no. 4, 1983, doi: 10.2307/2273659. [Online]. Available: https://www.cambridge.org/core/journals/journal-of-symbolic-logic/article/filter-lambda-model-and-the-completeness-of-type-assignment1/1F49872479426AFCB5DC777FA22509E0. [Accessed: 25-Jun-2020]

[4] G. Barthe and M. H. Sørensen, "Domain-free pure type systems," in *Logical foundations of computer science*, 1997, doi: 10.1007/3-540-63045-7_2 [Online]. Available: https://link.springer.com/chapter/10.1007/3-540-63045-7_2. [Accessed: 24-Jun-2020]

[5] D. R. Christiansen, "Bidirectional Typing Rules: A Tutorial." 17-Sep-2013 [Online]. Available: https://davidchristiansen.dk/tutorials/bidirectional.pdf. [Accessed: 02-Mar-2020]

[6] B. C. Pierce and D. N. Turner, "Local Type Inference," *ACM Transactions on Programming Languages and Systems*, vol. 22, no. 1, pp. 1–44, Jan. 2000, doi: 10.1145/345099.345100. [Online]. Available: https://dl.acm.org/doi/10.1145/345099.345100. [Accessed: 22-Jun-2020]

[7] L. Augustsson, "Cayenne — a language with dependent types," in *Proceedings of the third ACM SIGPLAN international conference on functional programming*, 1998, doi: 10.1145/289423.289451 [Online]. Available: https://dl.acm.org/doi/10.1145/289423.289451. [Accessed: 25-Jun-2020]

[8] T. Coquand, “An Analysis of Girard’s Paradox,” in *Proceedings of the first annual IEEE symposium on logic in computer science (LICS 1986)*, 1986.

[9] N. G. de Bruijn, “Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem,” *Indagationes Mathematicae (Proceedings)*, vol. 75, no. 5, 1972, doi: 10.1016/1385-7258(72)90034-0. [Online]. Available: https://www.sciencedirect.com/science/article/pii/1385725872900340. [Accessed: 17-May-2020]

[10] P. Padawitz, “Übersetzerbau (Algebraic Compiler Construction),” Fakultät für Informatik, TU Dortmund [Online]. Available: https://fldit-www.cs.uni-dortmund.de/~peter/CbauFolien.pdf. [Accessed: 25-Jun-2020]

[11] R. C. Moore, “Removing Left Recursion from Context-Free Grammars,” in *Proceedings of the 1st north american chapter of the association for computational linguistics conference*, 2000, doi: 10.5555/974305.974338 [Online]. Available: https://dl.acm.org/doi/abs/10.5555/974305.974338. [Accessed: 06-Jul-2020]

[12] A. Stump, “The Calculus of Dependent Lambda Eliminations,” *Journal of Functional Programming*, vol. 27, 2017, doi: 10.1017/S0956796817000053. [Online]. Available: https://www.cambridge.org/core/journals/journal-of-functional-programming/article/calculus-of-dependent-lambda-eliminations/1D0BDA070E9273AC56C108D8F6F2B078. [Accessed: 06-Jul-2020]