Eötvös Loránd University

Faculty of Informatics

Department of Programming Languages and Compilers

A functional programming language based on dependent type theory

Supervisor:
András Kovács
PhD Student

Author:
Zongpu Xie
Computer Science BSc

Budapest, 2021

EÖTVÖS LORÁND UNIVERSITY
FACULTY OF INFORMATICS

Thesis Registration Form

Student's Data:
Student's Name: Xie Zongpu
Student's Neptun code: RLKNMA

Course Data:
Student's Major: Computer Science BSc

I have an internal supervisor

Internal Supervisor's Name: *Kovács András*
Supervisor's Home Institution: *Eötvös Loránd University*
Address of Supervisor's Home Institution: *1053 Budapest, Egyetem tér 1–3.*
Supervisor's Position and Degree: *PhD Student*

Thesis Title: A functional programming language based on dependent type theory

Topic of the Thesis:
(Upon consulting with your supervisor, give a 150-300-word-long synopsis os your planned thesis.)

For the thesis, a functional programming language based on dependent type theory will be implemented in Haskell. The software will contain the definition of the language with a type checker and an interpreter.

The first part of the program is lexical and syntactic analysis. This will be implemented with the help of the monadic parser combinator library called Megaparsec.

The next part is semantic analysis, which uses type checking and type inference to check if a term is well-typed. Due to the dependent type system, terms can appear in types as well, thus type checking involves evaluation of certain parts of the program. The language can contain holes and implicit arguments, which can be inferred by the software.

If the type checking succeeds, then the program can be evaluated to its normal form.

The software will have a terminal user interface. The type checker reports any type errors and unfillable holes to the user. Simple functional programs will be written in the language to test the type checker and the interpreter.

Budapest, 2020.11.23.

Contents

Chapter 1

Introduction

This thesis specifies and implements a type checker and interpreter of a simple functional programming language with dependent types. It features row polymorphism and extensible records.

Chapter 2 intoduces the background concepts for the thesis.

Chapter 3 contains the usage and the specification of the language.

Chapter 4 covers the implementation of the type checker and the interpreter.

Chapter 2

Background

2.1 Functional programming

Functional programming is the idea of structuring software by composing and applying functions, where mutable state and side effects are isolated and kept track of. The functions are similar to mathematical functions, they take in values as parameters and return new values based on the given arguments. In contrast to imperative procedues, which are defined by sequences of statements with side effects, function definitions are expression trees of functions, operators, and values [6, 7].

Functions are treated just like any other values in functional languages, they can be given in function arguments, returned from functions, stored in data structures, and defined in any context [1, 6].

If one defines a function that takes other functions as arguments, it is called a *higher-order functions*. For example,

$$\mathit{twice}(f, x) := f(f(x)) \tag{2.1}$$

is a higher order function, it takes a function and a value, returns a function applied to that value twice [6, 12]. Higher-order functions allow one to refactor functions with similar structures by having parts of the definition be parameters of the new function.

The ability for a function to return another function gives rise to the technique called *currying*, where instead of having function arguments be given in tuples of values, the function is instead defined to have a single argument then directly return another function that takes another argument and so on [2, 6, 12]. For example,

$$f(x)(y) \tag{2.2}$$

is a curried function applied to two arguments.

A desirable trait in functional programming is *referencial transparency*. It allows one to replace any variable with its definition or factor out parts of the expressions into a new variable without changing the semantics of the program [2, 6, 7]. This property is lost if side effects are unrestricted in functions.

Imperative loops need mutable variables to function, so to avoid mutable state, recursion is used instead in functional programming [6]. Often higher-order combinators which use recursion under the hood are used instead of explicit recursion, such as maps, folds, and recursion schemes, using them one can be sure that a particular function terminates [8, 11]. With an optimizing compiler, recursive functions can be just as performant as imperative loops [1].

2.2 Lambda calculus

Lambda calculus is a model of computation based on mathematical functions, it is the basis of functional programming languages. The simplest untyped version only has three syntactic constructs (see fig. 2.1). Lambda abstraction binds or captures variables and creates anonymous functions, those variables are *bound*, variables which are not bound with regards to a lambda abstraction are called *free variables* [2, 5, 6, 12]. Bound variables can be renamed without changing the behavior, it is called *α-equivalence* [6, 12]. For example,

$$(\lambda x.\ \lambda y.\ (x\,x)\ y)\ (\lambda x.\ y\,x) \tag{2.3}$$

is a lambda term, which is α-equivalent to

$$(\lambda a.\ \lambda b.\ (a\,a)\ b)\ (\lambda c.\ y\,c) \tag{2.4}$$

Note that the y here cannot be renamed, since it is a free variable, and that the xs were bound to different binders, which is why they can become both a and c.

Lambda calculus has a single rule for computation called *β-reduction* [2, 3, 5, 6, 12], the rule is as follows:

$$(\lambda x.\ t)\ u \mapsto t[x := u] \tag{2.5}$$

One needs to be careful about variable names when substituting to avoid accidental

$$
\begin{aligned}
t, u ::=\ & x && \text{variable} \\
\mid\ & \lambda x.\, t && \text{lambda abstraction} \\
\mid\ & t\, u && \text{function application}
\end{aligned}
$$

Figure 2.1: The syntax of untyped lambda calculus

capture of free variables. For example, the lambda term in eq. (2.3) will become

$$\lambda z.\, ((\lambda x.\, y\, x)\, (\lambda x.\, y\, x))\, z \tag{2.6}$$

after one β-reduction step. Note that the bound y is renamed to z to avoid clashing with the free y variable.

Another concept is the *η-equivalence*, which says the following [3, 5, 6, 12]:

$$(\lambda x.\, f\, x) \simeq f \tag{2.7}$$

where x is not a free variable in f. The two sides are equated since they are equal after applying to an argument.

Lambda terms which can be β-reduced are called *reducible expression* (*redex*), terms which do not have a redex are said to be in *normal form* [2, 3, 5, 6, 12]. At each step there can be many ways to apply β-reduction, but according to the Church–Rosser theorem, the normal forms after repeated reduction (if it terminates) are always equivalent regardless of the order of β-reduction steps, this property is called *confluence* [3, 5, 6, 12].

There are different strategies to select which β-reduction step to take. Mainstream programming languages take the *call-by-value* strategy, where one first evaluates the argument before reducing a redex, however it can lead to nontermination for some terms even though there is a sequence of reductions which lead to a normal form. Another strategy is *call-by-name*, where one always reduces the leftmost outermost redex. This will always produce a normal form where it exists, however there can be redundant computations on identical terms [6].

Some lambda terms do not have normal forms at all, for example:

$$\Omega = (\lambda x.\, x\, x)\, (\lambda x.\, x\, x) \tag{2.8}$$

General recursion can also be represented, for example with the *Y combinator*:

$$Y = \lambda f.\, (\lambda x.\, f\, (x\, x))\, (\lambda x.\, f\, (x\, x)) \tag{2.9}$$

however, it is inefficient in practice, so programming languages do not use it to implement recursion. One can show that the untyped lambda calculus is Turing-complete.

For practical programming, data types can be encoded with lambda functions. There is the *Church encoding*, which encodes inductive data types with their folds or recursor. Natural numbers can be represented by functions which takes a function and a constant, then using the function as the successor function and the constant as zero.

One can use *de Bruijn indices* or *de Bruijn levels* to avoid variable names and have trivial α-equivalence. Variables are represented by numbers based on which accessible lambda abstraction they are bound to. De Bruijn indices count from the innermost lambda abstraction [4], while de Bruijn levels count from the outermost. For example, with de Bruijn indices, the lambda term in eq. (2.3) can be written as

$$(\lambda\,\lambda\,(1\;1)\;0)\;(\lambda\;1\;0) \tag{2.10}$$

Note that the free y variable becomes a number outside of the range of its accessible lambda abstractions. Likewise, one can write the term with de Bruijn levels as

$$(\lambda\,\lambda\,(0\;0)\;1)\;(\lambda\;1\;0) \tag{2.11}$$

2.3 Types

To avoid non-termination or ill behaved terms, one can add types into lambda calculus.

The simplest typed version of lambda calculus is the *simply typed lambda calculus* ($STLC$).

Dependent types

Proofs

Chapter 3

User documentation

3.1 Installation

1. Install Stack [13].
2. Clone or download the source code repository.
3. Enter the repository and run the following:

   `stack install --flag unnamed:release`.

3.2 Usage

An Unnamed program can be written in a file with any source code editor.

Comments in the language are similar to Haskell and its derived languages, with both line comments and block comments.

```
-- This is a line comment
{- This is a
   block comment -}
```

The identity function can be defined like so:

```
id = λ (A : Type)
```

Types can be defined with their Church encoding

```
Bool  : Type = ∀ (A : Type) → A → A → A
```

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B[x := t]}{\Gamma \vdash \mathsf{let}\{x : A = t;\ u\} : B}$$

$$\frac{}{\Gamma \vdash \mathsf{U} : \mathsf{U}} \qquad \frac{\Gamma \vdash A : \mathsf{U} \qquad \Gamma, x : A \vdash B : \mathsf{U}}{\Gamma \vdash \forall(x : A) \to B : \mathsf{U}}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x \to t : \forall(x : A) \to B} \qquad \frac{\Gamma \vdash t : \forall(x : A) \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\,u : B[x := u]}$$

$$\frac{\Gamma \vdash A : \mathsf{U}}{\Gamma \vdash \mathsf{Row}\,A : \mathsf{U}} \qquad \frac{}{\Gamma \vdash \#\{\} : \mathsf{Row}\,A} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash r : \mathsf{Row}\,A}{\Gamma \vdash \#\{l : t \mid r\} : \mathsf{Row}\,A}$$

$$\frac{\Gamma \vdash R : \mathsf{Row}\,\mathsf{U}}{\Gamma \vdash \mathsf{Rec}\,R : \mathsf{U}} \qquad \frac{}{\Gamma \vdash \mathsf{rec}\{\} : \mathsf{Rec}\,\#\{\}} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : \mathsf{Rec}\,R}{\Gamma \vdash \mathsf{rec}\{l = t \mid u\} : \mathsf{Rec}\,\#\{l : A \mid R\}}$$

$$\frac{\Gamma \vdash t : \mathsf{Rec}\,\#\{l : A \mid R\}}{\Gamma \vdash t.l : A} \qquad \frac{\Gamma \vdash t : \mathsf{Rec}\,\#\{l : A \mid R\}}{\Gamma \vdash t.{-}l : \mathsf{Rec}\,R}$$

Figure 3.1: Typing rules

```
true  : Bool = λ A x y → x
false : Bool = λ A x y → y
```

3.3 Errors

3.4 Lexical structure

The language is indentation sensitive, similarly to Haskell [10].

3.5 Syntax

$$::=$$

3.6 Type system

The typing rules of the language are presented in fig. 3.1.

$$\frac{}{x \Downarrow x} \qquad \frac{u[x := t] \Downarrow u'}{\mathsf{let}\{x = t;\ u\} \Downarrow u'}$$

$$\frac{}{\mathsf{U} \Downarrow \mathsf{U}} \qquad \frac{A \Downarrow A' \qquad B \Downarrow B'}{\forall(x : A) \to B \Downarrow \forall(x : A') \to B'}$$

$$\frac{t \Downarrow t'}{\lambda x \to t \Downarrow \lambda x \to t'} \qquad \frac{t \Downarrow \lambda x \to v \qquad v[x := u] \Downarrow v'}{t\,u \Downarrow v'} \qquad \frac{t \Downarrow n \qquad u \Downarrow u'}{t\,u \Downarrow n\,u'}$$

$$\frac{A \Downarrow A'}{\mathsf{Row}\,A \Downarrow \mathsf{Row}\,A'} \qquad \frac{}{\#\{\} \Downarrow \#\{\}} \qquad \frac{t \Downarrow t' \qquad r \Downarrow r'}{\#\{l : t \mid r\} \Downarrow \#\{l : t' \mid r'\}}$$

$$\frac{R \Downarrow R'}{\mathsf{Rec}\,R \Downarrow \mathsf{Rec}\,R'} \qquad \frac{}{\mathsf{rec}\{\} \Downarrow \mathsf{rec}\{\}} \qquad \frac{t \Downarrow t' \qquad u \Downarrow u'}{\mathsf{rec}\{l = t \mid u\} \Downarrow \mathsf{rec}\{l = t' \mid u'\}}$$

$$\frac{t \Downarrow \mathsf{rec}\{l = u \mid v\}}{t.l \Downarrow u} \qquad \frac{t \Downarrow \mathsf{rec}\{l' = u \mid v\} \qquad l \neq l' \qquad v.l \Downarrow w}{t.l \Downarrow w} \qquad \frac{t \Downarrow n}{t.l \Downarrow n.l}$$

$$\frac{t \Downarrow \mathsf{rec}\{l = u \mid v\}}{t.{-l} \Downarrow v} \qquad \frac{t \Downarrow \mathsf{rec}\{l' = u \mid v\} \qquad l \neq l' \qquad v.{-l} \Downarrow v'}{t.{-l} \Downarrow \mathsf{rec}\{l' = u \mid v'\}} \qquad \frac{t \Downarrow n}{t.{-l} \Downarrow n.{-l}}$$

Figure 3.2: Big-step operational semantics

3.7 Semantics

Figure 3.2

Chapter 4

Developer documentation

4.1 Project structure

The project is implemented in Haskell, using the Glasgow Haskell Compiler with many of its extensions enabled.

Stack [13] is used to manage the project and its dependencies.

Algorithm ideas are taken from [9].

Figure 4.1

4.2 Raw Syntax

Listing 1

4.3 Core Syntax

Listing 2

4.4 Parsing

The Haskell library Megaparsec is used to do both lexing and parsing at the same time. Megaparsec is a monadic parser combinator library.

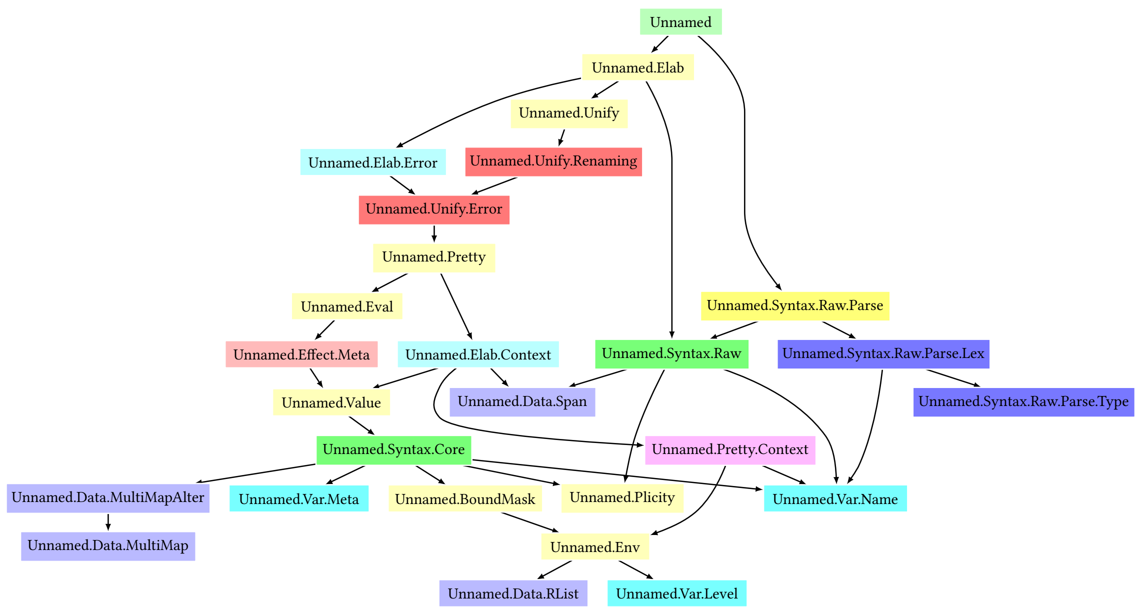

Figure 4.1: Transitively reduced dependency graph of the Haskell modules

```
data Term
  = Span Span Term
  | Var Name
  | Hole
  | Let Name (Maybe Term) Term Term
  | U
  | Pi Name (Maybe Term) Term
  | Lam Name (Maybe Term) Term
  | App Term Term
  | RowType Term
  | RowEmpty
  | RowExt Name Term Term
  | RecordType Term
  | RecordEmpty
  | RecordExt Name (Maybe Term) Term Term
  | RecordProj Name Term
  | RecordRestr Name Term
```

Listing 1: Raw syntax ADT

```haskell
data Term
  = Var Level
  | Meta Meta (Maybe BoundMask)
  | Let Name Term Term
  | U
  | Pi Name Term Term
  | Lam Name Term
  | App Term Term
  | RowType Term
  | RowLit (MultiMap Name Term)
  | RowExt (MultiMap Name Term) Term
  | RecordType Term
  | RecordLit (MultiMap Name Term)
  | RecordProj Name Int Term
  | RecordAlter (MultiMapAlter Name Term) Term
```

Listing 2: Core syntax ADT

4.5 Evaluation

Evaluating to normal form is done with normalization-by-evaluation (NBE)

4.6 Unification

4.7 Elaboration

Bidirectional type-checking algorithm is used.

4.8 Main

4.9 Testing

Chapter 5

Conclusion

Bibliography

[1] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs, Second Edition.* MIT Press, 1996. ISBN: 0-262-01153-0.

[2] Hendrik Pieter Barendregt. “Functional Programming and Lambda Calculus”. In: *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics.* Ed. by Jan van Leeuwen. Elsevier and MIT Press, 1990, pp. 321–363. DOI: 10.1016/b978-0-444-88074-1.50012-3.

[3] Hendrik Pieter Barendregt. *The Lambda Calculus – Its Syntax and Semantics.* Vol. 103. Studies in Logic and the Foundations of Mathematics. North-Holland, 1985. ISBN: 978-0-444-86748-3.

[4] N. G. de Bruijn. “Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church–Rosser Theorem”. In: *Indagationes Mathematicae (Proceedings)* 75.5 (1972), pp. 381–392. DOI: 10.1016/1385-7258(72)90034-0.

[5] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and Lambda-Calculus.* Cambridge University Press, 1986.

[6] Paul Hudak. “Conception, Evolution, and Application of Functional Programming Languages”. In: *ACM Comput. Surv.* 21.3 (1989), pp. 359–411. DOI: 10.1145/72551.72554.

[7] John Hughes. “Why Functional Programming Matters”. In: *Comput. J.* 32.2 (1989), pp. 98–107. DOI: 10.1093/comjnl/32.2.98.

[8] Graham Hutton. “A Tutorial on the Universality and Expressiveness of Fold”. In: *J. Funct. Program.* 9.4 (1999), pp. 355–372. URL: http://journals.cambridge.org/action/displayAbstract?aid=44275.

[9] András Kovács. *elaboration-zoo*. 2021. URL: `https://github.com/AndrasKovacs/elaboration-zoo/tree/dc5904b092657267ef9eda22c56c8d907ff811db`.

[10] Simon Marlow, ed. *Haskell 2010 Language Report*. 2010. URL: `https://www.haskell.org/onlinereport/haskell2010/`.

[11] Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. "Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire". In: *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26–30, 1991, Proceedings*. Ed. by John Hughes. Vol. 523. Lecture Notes in Computer Science. Springer, 1991, pp. 124–144. DOI: `10.1007/3540543961_7`.

[12] György E. Révész. *Lambda-Calculus, Combinators, and Functional Programming*. Vol. 4. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1988. ISBN: 978-0-521-34589-7.

[13] Stack contributors. *The Haskell Tool Stack*. Version 2.7.1. 2021. URL: `https://docs.haskellstack.org/en/v2.7.1/`.

List of Figures

List of Listings