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 called *Pirec* (/ˈpaɪ.ɹɛk/). It features row polymorphism and extensible records.

Chapter 2 intoduces the background concepts to use the program.

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 [16, 17].

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, 16].

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

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

is a higher order function, it takes a function and a value, returns a function applied to that value twice [16]. 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 [3, 12, 16]. For example,

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

is a curried function applied to two arguments.

A desirable trait in functional programming is *referential 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 [3, 16, 17]. 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 [16]. 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 [18, 29]. 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 [16]. The simplest untyped version only has three syntactic constructs (see fig. 2.1) with parentheses for grouping. 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* [3, 9, 15]. Bound variables can be renamed without changing the behavior, it is called *α-equivalence* [4, 32, 33]. 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*, the rule is as follows [15, 16]:

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

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 [4, 16]. 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 [4, 16]:

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

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* [3, 15, 33]. 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* [10, 15, 33].

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 non-termination for some terms even though there is a sequence of reductions which lead to a normal form [1, 32, 34]. 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 [1, 34]. A compromise between the two strategies is *call-by-need*, where function arguments are memoized when needed, therefore they are evaluated at most once, but untouched when not needed [2, 16], thought it can lead to space leaks when arguments are unnecessarily stored in unevaluated thunks [30].

Some lambda terms do not have normal forms at all, for example [15, 16, 32]:

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

General recursion can also be represented, for example with the *Y combinator* [15, 16, 33]:

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

however, it is inefficient in practice, so programming languages do not typically use it to implement recursion. One can show that the untyped lambda calculus is Turing-complete [32, 36].

For practical programming, data types can be encoded with purely lambda functions. There is the *Church encoding*, which encodes inductive data types with their folds or recursor [20, 21]. For example, a natural number can be represented by a function which takes a function and a constant, then using the function as the successor function and the constant as zero [9, 32, 33]. The number 5 can be represented as follows:

$$\lambda f.\ \lambda x.\ f(f(f(f(f(x))))) \tag{2.10}$$

Operations which can be defined with folds, such as addition and multiplication, can be defined simply on the Church-encoded numbers as well, however operations such as the predecessor function has to be defined in a more complicated way and its time complexity is $O(n)$ [32, 33]. *Scott encoding* avoids this problem by encoding recursive data types with their destructors, however recursive operations need to be defined using general recursion and recursive types are required to represent them in typed settings [12, 20, 21]. The two encodings coincide for non-recursive data types.

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, while de Bruijn levels count from the outermost [6]. For example, with de Bruijn indices, the lambda term in eq. (2.3) can be written as

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

Note that the free y variable becomes a number outside of the range of its accessible lambda abstractions.

$$
\begin{aligned}
A, B ::= {} & \iota && \text{base type} \\
\mid {} & A \to B && \text{function}
\end{aligned}
$$

Figure 2.2: The syntax of STLC types

$$
\frac{}{\Gamma,\ x : A \vdash x : A}
\qquad
\frac{\Gamma,\ x : A \vdash t : B}{\Gamma \vdash \lambda(x : A).\ t : A \to B}
\qquad
\frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B}
$$

Figure 2.3: The typing rules of STLC

2.3 Types

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

The simplest version of typed lambda calculus is the *simply typed lambda calculus* (STLC) [5, 8, 32]. Types which are separate from terms are added to untyped lambda calculus. There are only two constructs, a base type and function types (see fig. 2.2). Each lambda abstraction binder is annotated with a type. For example, the *twice* function from eq. (2.1) can be written as follows:

$$
\lambda(f : \iota \to \iota).\ \lambda(x : \iota).\ f\,(f\,x) \tag{2.12}
$$

One can check if a term is correctly typed with the rules from fig. 2.3, and all correctly typed terms terminate.

Unfortunately STLC is too restrictive, as it does not allow parametric polymorphic functions.

A more advanced version with parametric polymorphism is System F.

Dependent types [27]

Proofs

Chapter 3

User documentation

Pirec is a dependently typed functional language with row polymorphism and extensible records. It can be used to write programs without side effect and prove simple theorems. The program is command line tool that can type check and interpret Pirec programs.

3.1 Installation

1. Install Stack [35].
2. Clone or download the source code repository.
3. Enter the repository and run the following:

   ```
   stack install --flag pirec:release
   ```

3.2 Usage

A Pirec program can be written into a file with any source code editor.

Comments in Pirec 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) → λ (a : A) → a
```

It can then be applied to for example an empty record:

```
id (Rec #{}) rec{}
```

where `Rec #{}` is the type of empty records and `rec{}` is an instance of an empty record. These two lines can be put into a file, for example into `id.pirec`, then run `pirec id.pirec`. The program will then output the following, prettyprinting the normal form of the last line of the input and its inferred type:

```
rec{  }
  : Rec #{  }
```

Here is another way to define the identity function:

```
id2 : ∀ (A : Type) → A → A = λ _ a → a
```

The definition is given a type signature, the lambda binders are grouped together with a single lambda and their types are removed since they are optional, the unused lambda parameter is ignored with an underscore.

The first parameter can be made into an implicit parameter, and its type is also optional:

```
id3 : ∀ {A} → A → A = λ a → a
```

With this, the type of the argument does not need to be written out:

```
id3 rec{}
```

Data types can be defined with their Church encoding, for example Booleans:

```
Bool  : Type = ∀ {A} → A → A → A
true  : Bool = λ x y → x
false : Bool = λ x y → y
```

3.3 Errors

3.4 Lexical structure

Pirec is indentation sensitive, similarly to Haskell.

Unicode	ASCII
→	->
∀	forall
λ	\

Table 3.1: ASCII alternatives to Unicode syntax

$$
\begin{array}{rcl}
\mathsf{let}\{x_1 = t_1;\ \ldots;\ x_n = t_n;\ u\} & \mapsto & \mathsf{let}\{x_1 = t_1;\ \ldots \mathsf{let}\{x_n = t_n;\ u\} \ldots\} \\
\forall (x_1\ :\ A_1) \ldots (x_n\ :\ A_n) \to B & \mapsto & \forall (x_1\ :\ A_1) \to \ldots \forall (x_n\ :\ A_n) \to B \\
\lambda x_1 \ldots x_n \to B & \mapsto & \lambda x_1 \to \ldots \lambda x_n \to B \\
\#\{l_1\ :\ t_1;\ \ldots;\ l_n\ :\ t_n\} & \mapsto & \#\{l_1\ :\ t_1 \mid \ldots \#\{l_n\ :\ t_n \mid \#\{\}\} \ldots\} \\
\#\{l_1\ :\ t_1;\ \ldots;\ l_n\ :\ t_n \mid r\} & \mapsto & \#\{l_1\ :\ t_1 \mid \ldots \#\{l_n\ :\ t_n \mid r\} \ldots\} \\
\mathsf{rec}\{l_1 = t_1;\ \ldots;\ l_n = t_n\} & \mapsto & \mathsf{rec}\{l_1 = t_1 \mid \ldots \mathsf{rec}\{l_n = t_n \mid \mathsf{rec}\{\}\} \ldots\} \\
\mathsf{rec}\{l_1 = t_1;\ \ldots;\ l_n = t_n \mid u\} & \mapsto & \mathsf{rec}\{l_1 = t_1 \mid \ldots \mathsf{rec}\{l_n = t_n \mid u\} \ldots\} \\
\mathsf{rec}\{l_1 \coloneqq t_1;\ \ldots;\ l_n \coloneqq t_n \mid u\} & \mapsto & \mathsf{rec}\{l_1 \coloneqq t_1 \mid \ldots \mathsf{rec}\{l_n \coloneqq t_n \mid u.{-}l_n\} \ldots .{-}l_1\}
\end{array}
$$

Figure 3.1: Syntactic sugar in Pirec

3.5 Syntax

```
atom = identifier | "_"
```

3.6 Type system

The typing rules of Pirec are presented in fig. 3.3 without implicit arguments.

[24]

3.7 Semantics

Figure 3.5

$t, u, v, w, r, A, B, R ::= x$	variable
$\mid \_$	hole
$\mid \mathsf{let}\{x = t;\ u\}$	let expression
$\mid \mathsf{U}$	universe
$\mid \forall(x\ :\ A) \to B$	dependent function
$\mid \lambda x \to t$	lambda abstraction
$\mid t\ u$	function application
$\mid \mathsf{Row}\ A$	row type
$\mid \#\{\}$	empty row
$\mid \#\{l\ :\ t \mid r\}$	row extension
$\mid \mathsf{Rec}\ R$	record type
$\mid \mathsf{rec}\{\}$	empty record
$\mid \mathsf{rec}\{l = t \mid u\}$	record extension
$\mid t.l$	record projection
$\mid t.\!-\!l$	record restriction

Figure 3.2: The desugared terms of Pirec

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

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

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

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

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

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

Figure 3.3: The typing rules of Pirec

$n ::= x$	variable
$\mid n\,t$	function application
$\mid \#\{l\ :\ t \mid n\}$	row extension
$\mid \mathsf{rec}\{l = t \mid n\}$	record extension
$\mid n.l$	record projection
$\mid n.{-}l$	record restriction

Figure 3.4: The neutral terms of Pirec

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

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

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

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

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

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

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

Figure 3.5: The big-step operational semantics of Pirec

Chapter 4

Developer documentation

4.1 Project structure

The project is implemented in Haskell [26], using the *Glasgow Haskell Compiler* (GHC) with many of its extensions enabled [13].

Stack [35] is used to manage the project and its dependencies.

Project structure and algorithm ideas are from [22].

[23]

Figure 4.1

4.2 ?

[14, 31, 37, 38]

4.3 Raw Syntax

Listing 1

4.4 Core Syntax

Listing 2

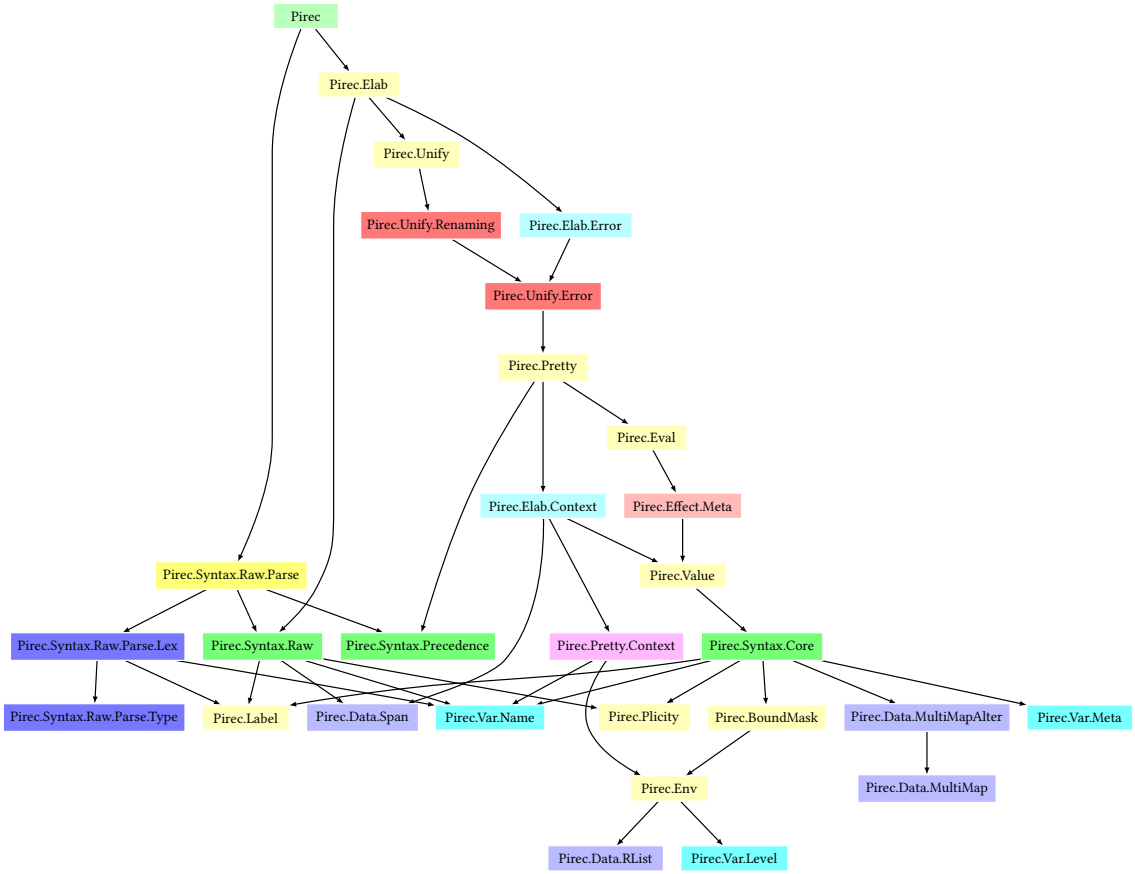

Figure 4.1: Transitively reduced dependency graph of the Haskell modules

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

Listing 1: Raw syntax ADT

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

Listing 2: Core syntax ADT

4.5 Parsing

The Haskell library Megaparsec [28] is used to do both lexing and parsing at the same time. Megaparsec is a monadic parser combinator library [19].

4.6 Evaluation

Normalization-by-evaluation (NBE) with explicit closures is implemented to evaluate lambda terms to normal form.

4.7 Unification

4.8 Elaboration

Bidirectional type-checking algorithm is used [11, 25].

4.9 Main

[7]

4.10 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] Zena M. Ariola et al. "The Call-by-Need Lambda Calculus". In: *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23–25, 1995*. Ed. by Ron K. Cytron and Peter Lee. ACM Press, 1995, pp. 233–246. DOI: 10.1145/199448.199507.

[3] 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.

[4] 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.

[5] Hendrik Pieter Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Perspectives in Logic. Cambridge University Press, 2013. ISBN: 978-0-521-76614-2. URL: http://www.cambridge.org/de/academic/subjects/mathematics/logic-categories-and-sets/lambda-calculus-types.

[6] 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.

[7] Paolo Capriotti and Huw Campbell. *Optparse-applicative*. Version 0.16.1.0. 2020. URL: http://hackage.haskell.org/package/optparse-applicative-0.16.1.0.

[8] Alonzo Church. "A Formulation of the Simple Theory of Types". In: *J. Symb. Log.* 5.2 (1940), pp. 56–68. DOI: 10.2307/2266170.

[9] Alonzo Church. *The Calculi of Lambda Conversion* (*AM-6*). Princeton University Press, 1985. ISBN: 9781400881932. DOI: 10.1515/9781400881932.

[10] Alonzo Church and J. Barkley Rosser. "Some Properties of Conversion". In: *Transactions of the American Mathematical Society* 39.3 (1936), pp. 472–472. DOI: 10.1090/s0002-9947-1936-1501858-0.

[11] Thierry Coquand. "An Algorithm for Type-Checking Dependent Types". In: *Sci. Comput. Program.* 26.1–3 (1996), pp. 167–177. DOI: 10.1016/0167-6423(95)00021-6.

[12] Haskell B. Curry. *Combinatory Logic*. Studies in Logic and the Foundations of Mathematics. Amsterdam: North-Holland Pub. Co, 1958. ISBN: 9780720422085.

[13] GHC Team. *GHC User's Guide*. Version 8.10.4. 2021. URL: https://downloads.haskell.org/ghc/8.10.4/docs/html/users_guide/index.html.

[14] Adam Gundry et al. *Optics*. Version 0.4. Well-Typed LLP. 2021. URL: http://hackage.haskell.org/package/optics-0.4.

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

[16] 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.

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

[18] 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.

[19] Graham Hutton and Erik Meijer. "Monadic Parsing in Haskell". In: *J. Funct. Program.* 8.4 (1998), pp. 437–444. URL: http://journals.cambridge.org/action/displayAbstract?aid=44175.

[20] Jan Martin Jansen. "Programming in the λ-Calculus: From Church to Scott and Back". In: *The Beauty of Functional Code – Essays Dedicated to Rinus Plasmeijer on the Occasion of His 61st Birthday*. Ed. by Peter Achten and Pieter W. M. Koopman. Vol. 8106. Lecture Notes in Computer Science. Springer, 2013, pp. 168–180. DOI: 10.1007/978-3-642-40355-2_12.

[21] Pieter W. M. Koopman, Rinus Plasmeijer, and Jan Martin Jansen. "Church Encoding of Data Types Considered Harmful for Implementations: Functional Pearl". In: *Proceedings of the 26th 2014 International Symposium on Implementation and Application of Functional Languages, IFL '14, Boston, MA, USA, October 1–3, 2014*. Ed. by Sam Tobin-Hochstadt. ACM, 2014, 4:1–4:12. DOI: 10.1145/2746325.2746330.

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

[23] Dmitrii Kovanikov et al. *Relude*. Version 1.0.0.1. Kowainik. 2021. URL: http://hackage.haskell.org/package/relude-1.0.0.1.

[24] Daan Leijen. "Extensible Records with Scoped Labels". In: *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005, Tallinn, Estonia, 23–24 September 2005*. Ed. by Marko C. J. D. van Eekelen. Vol. 6. Trends in Functional Programming. Intellect, 2005, pp. 179–194.

[25] Andres Löh, Conor McBride, and Wouter Swierstra. "A Tutorial Implementation of a Dependently Typed Lambda Calculus". In: *Fundam. Informaticae* 102.2 (2010), pp. 177–207. DOI: 10.3233/FI-2010-304.

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

[27] Per Martin-Löf. *Intuitionistic Type Theory*. Vol. 1. Studies in Proof Theory. Bibliopolis, 1984. ISBN: 978-88-7088-228-5.

[28] Megaparsec contributors, Paolo Martini, and Daan Leijen. *Megaparsec*. Version 9.0.1. 2020. URL: http://hackage.haskell.org/package/megaparsec-9.0.1.

[29] 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.

[30] Neil Mitchell. "Leaking Space". In: *Commun. ACM* 56.11 (2013), pp. 44–52. DOI: 10.1145/2524713.2524722.

[31] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999. ISBN: 978-0-521-66350-2.

[32] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. ISBN: 978-0-262-16209-8.

[33] 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.

[34] Peter Sestoft. "Demonstrating Lambda Calculus Reduction". In: *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones* [*on occasion of his 60th birthday*]. Ed. by Torben Æ. Mogensen, David A. Schmidt, and Ivan Hal Sudborough. Vol. 2566. Lecture Notes in Computer Science. Springer, 2002, pp. 420–435. DOI: 10.1007/3-540-36377-7_19.

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

[36] Alan M. Turing. "Computability and λ-Definability". In: *J. Symb. Log.* 2.4 (1937), pp. 153–163. DOI: 10.2307/2268280.

[37] Philip Wadler et al. *Prettyprinter*. Version 1.7.0. 2020. URL: http://hackage.haskell.org/package/prettyprinter-1.7.0.

[38] Love Waern. *In-other-words*. Version 0.2.0.0. 2021. URL: http://hackage.haskell.org/package/in-other-words-0.2.0.0.

List of Figures

List of Tables

List of Listings