
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 details 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 [35, 37].

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 [2, 35].

If one defines a function that takes other functions as arguments, it is knwon as 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 [35]. 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 known as *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 [5, 21, 35]. 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 [5, 35, 37]. 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 [35]. 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 [39, 59]. With an optimizing compiler, recursive functions can be just as performant as imperative loops [2].

## 2.2 Lambda calculus

*Lambda calculus* is a model of computation based on mathematical functions, it is the basis of functional programming languages [35]. 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 known as *free variables* [5, 16, 34]. Bound variables can be renamed without changing the behavior, it is known as *α-equivalence* [64, 70, 72]. 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 $x$s were bound to different binders, which is why they can become both $a$ and $c$.

Lambda calculus has a single rule for computation known as *β-reduction*, the rule is as follows [34, 35, 64]:

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

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

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

Figure 2.1: The syntax of untyped lambda calculus

capture of free variables [6, 35]. 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 *η-equivalence*, which says the following [6, 35]:

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

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 known as *reducible expression* (*redex*), terms which do not have a redex are said to be in *normal form* [5, 34, 64]. 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 *confluence* [17, 34, 64].

Some lambda terms do not have normal forms at all, for example [34, 35, 70]:

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

General recursion can also be represented, for example with the *Y combinator* [34, 35, 64]:

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

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 [70, 79].

### 2.2.1 Evaluation strategies

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 [2, 70, 74].

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 [2, 74].

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 [4, 35], thought it can lead to space leaks when arguments are unnecessarily stored in unevaluated thunks [63].

### 2.2.2 Church encoding

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 [42, 47].

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 [16, 70, 72]. 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 $O(n)$ [70, 72].

*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 [21, 42, 47]. The two encodings coincide for non-recursive data types.

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

Figure 2.2: The syntax of STLC types

### 2.2.3 De Bruijn indices and de Bruijn levels

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 [11].

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.

## 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) [15]. 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 [7, 70]. 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 [7, 70]. Unfortunately STLC is too restrictive, as it does not allow parametric polymorphic functions, the function in eq. (2.12) needs to be repeated for every type one plans to use it on.

A more advanced version of typed lambda calculus with parametric polymorphism is

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

Figure 2.3: The typing rules of STLC

$$\frac{\Gamma,\ X : \mathsf{U} \vdash t : A}{\Gamma \vdash \Lambda X.\ t : \forall X.\ A} \qquad \frac{\Gamma \vdash t : \forall X.\ A \quad \Gamma \vdash B : \mathsf{U}}{\Gamma \vdash t\ B : A[X \mapsto B]}$$

Figure 2.4: The extra typing rules of System F

*System F* [28, 64, 70]. It allows terms to depend on types by introducing abstraction and application of types by terms (see fig. 2.4).

One can encode data types with Boehm–Berducci encoding [10], the typed version of Church encoding. For example, the following type represents natural numbers:

$$\forall X.\ (X \to X) \to (X \to X) \tag{2.13}$$

Then the number 5 can be represented as follows:

$$\Lambda X.\ \lambda(f : X \to X).\ \lambda(x : X).\ f(f(f(f(f\,x)))) \tag{2.14}$$

There is a subset of System F known as Hindley–Milner [60], where all the types can be omitted and inferred automatically, the ML family of languages are based on it [56, 61].

If one allows types to depend on terms and types (see fig. 2.5), then one arrives at full dependent type theory. This erases the distinction between terms and types, so term evaluation is necessary during type-checking as well, but correctly typed terms still terminate [20, 57, 64]. These previously mentioned systems are part of the lambda cube, where you can get different type systems depending on whether types and terms can depend on each other [8, 64].

Due to Curry–Howard correspondence, types can be seen as propositions in intuitionistic logic and terms can be seen as proofs of those propositions [75, 82]. For example,

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

Figure 2.5: The new typing rule for dependent types

the type

$$(A \to (B \to C)) \to ((A \to B) \to (A \to C)) \tag{2.15}$$

can be seen as a proposition where the function arrows are implications, and the proof is the term

$$\lambda(f : A \to (B \to C)).\ \lambda(g : A \to B).\ \lambda(x : A).\ (f\,x)\ (g\,x) \tag{2.16}$$

# Chapter 3

# User documentation

Pirec is a functional language with dependent functions, dependent pairs, 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

The program can be installed as follows:

1. Install *Stack* [76].
2. Clone or download the source code repository.
3. Enter the repository and run the following command: `stack install`

(During the compilation of `Paths_pirec` module, there will be warning about a prepositive qualified module. This is due to a bug in *Cabal*'s [12] automatic file generation in combination with the warning which is only fixed in a recent pull request to the development version of Cabal [9].)

## 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 #{  }
```

One can give command line options to the program to modify what to output, for example `--elab-only` can be given so that the program does not evaluate to normal form, instead it only elaborates and infers the type. Run `pirec --help` to see all options.

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 definition can also be split across multiple lines, as long as the following lines have more indentation than the beginning of the definition

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{}
```

Running `pirec id3.pirec --elab-only --metas` it outputs the following:

```
let{ id3 = λ {A} a → a; id3 {?1} rec{  } }
  : Rec #{  }
Metavariables:
  ?0 = Type
  ?1 = Rec #{  }
```

Here it shows that the implicit argument is elaborated to a metavariable which is solved to the correct type.

Data types can be defined with their Church encoding (see section 2.2.2), for example the natural numbers:

```
Nat  : Type = ∀ {A} → (A → A) → A → A
zero : Nat  = λ _ z → z
suc  : Nat → Nat = λ x s z → s (x s z)
```

One can then define operators on them like addition and multiplication:

```
add : Nat → Nat → Nat = λ x y s z → x s (y s z)
mul : Nat → Nat → Nat = λ x y s z → x (y s) z
```

Then one can evaluate simple arithmetic expressions such as $1 + 2 * 3$:

```
add (suc zero) (mul (suc (suc zero)) (suc (suc (suc zero))))
```

Running this will get you the correct Church encoded number:

```
λ {A} s z → s (s (s (s (s (s z)))))
  : ∀ {A : Type} → (A → A) → A → A
```

Records can be created in Pirec, for example:

```
r : Rec #{ x : Nat; y : Nat } = rec{ x = suc zero; y = suc (suc zero) }
```

Record types are denoted with `Rec`, then follows a row, and `#` is the start of a row literal, following that are the two fields surrounded by braces which are both `Nat` in this case. Record literals are denoted with the keyword `rec`, following that are the two fields which are given values. Records can be extended and projected from, and restricted from:

```
r2 : Rec #{ x : Nat; z : Nat } = rec{ z = suc r.y | r.-y }
```

Here there is a new z field which takes the same successor of the y field, while the original y field is removed. Record extension is like a literal, but with a pipe and the original record right of it. Record projection and restriction are done with a period or .- followed by the field respectively. There is syntactic sugar for modifying a field, which is the same as restricting then extending with the same field:

```
r3 : Rec #{ x : Nat; y : Nat } = rec{ x := zero | r.-x }
r4 : Rec #{ x : Nat; y : Nat } = rec{ x := zero | r }
```

The above two records are the same, both have the x field become zero.

Row polymorphism can be used to manipulate records with unknown fields. For example

```
f1 : ∀ {R} → Rec #{ x : Nat | R } → Rec #{ x : Nat | R }
   = λ r → rec{ suc r.x | r }
```

is a function that takes any record with an x field and modifies it to have the field have its successor. The implicit argument R is an arbitrary row, and pipe is used for row extension similarly to record extension.

The extensible record is based on [51], so there multiple fields can have the same name in the same row and record. For example

```
r5 : Rec #{ x : Nat; x : Nat } = rec{ x = suc zero; x = suc (suc zero) }
```

is valid. Note that the order of fields with the matter when the name of the fields are the same, in this case, projection and restriction only work on the left most field of that name. To get a field which is not the left most, one can first restrict then project, like so:

```
r5.-x.x
```

One can use dependent products and records to create a sort of list type. It is defined as follows:

```
List = λ A → ∃ (x : Nat) ×
               let z = #{}
                   s = λ (R : Row Type) → #{ e : A | R }
                   x s z
```

| Unicode | ASCII |
|---|---|
| → | -> |
| ∀ | forall |
| λ | \ |
| × | & |
| ∃ | exists |

Table 3.1: ASCII alternatives to Unicode syntax

It is a type for dependent pairs with a natural number, and a record with that amount of fields all named `e`. The dependent product type is denoted similarly to the dependent function type, except that `∃` and `×` are used instead. Right of the times sign, a let expression is used, local definitions can be given with it with the last line as the return value. The definitions and the return value must align, but there is an alternative syntax, and can be like so:

```
List = λ A → ∃ (x : Nat) ×
               let{ z = #{}
                  ; s = λ (R : Row Type) → #{ e : A | R }
                  ; x s z }
```

One might notice that the syntax of braces and semicolons is similar rows and records. In fact, rows and records can also be defined without them. For example here is an element of the `List` type:

```
l : List Nat = suc (suc (suc zero)) ,
               rec e = zero
                   e = suc zero
                   e = suc (suc zero)
```

The constructor of dependent pair is denoted with a comma. `Row` is the type of rows which can take any type, but to use it with records it can only be `Type`.

Every Unicode character shown has an ASCII alternative which can be used instead, the correspondences are shown in table 3.1.

```
examples/thousand.pirec:2:12:
  |
2 | y : Type = something x
  |            ^^^^^^^^^
variable something out of scope
```

Figure 3.1: An example error

## 3.3 Errors

Pirec can report many kinds of errors when given a program, it will also report the position of the error in a visual way (see fig. 3.1). The following is a list of possible errors:

- `pirec: ...: openFile: does not exist (No such file or directory)`

  The file given in the command line argument does not exist.

- `unexpected ...`

  A parse error when the parser was expecting some other token.

- `incorrect indentation ...`

  A parse error when some token is at an incorrect indentation level. Due to the way parsing is implemented, sometimes this error occurs when an expression suddenly ends when it should not have.

- `variable ... out of scope`

  A variable is referenced when there is no such variable in the current scope.

- `got ... application when ... application was expected`

  An implicit argument was given to a function with explicit parameter or the other way around, and the program could not insert an implicit argument.

- ```
  expected type:
    ...
  but got inferred type:
    ...
  ```
  ```

Could not match expected type and inferred type when unifying two types.

- `variable ... not in scope when solving ...`

  A metavariable could not be solved due to the solution requiring a variable outside of the scope of the metavariable.

- `occurs check failed when solving ...`

  A metavariable could not be solved due to the solution depending on that same metavariable.

- `ambiguous hole due to multiple instances of variable ... in the context`

  A metavariable could not be solved due to it depending on the same variable more than once, so there might not be a unique solution.

- `got nonvariable in the context: ...`

  A metavariable could not be solved due to it depending on something which is not a variable.

- `got non-invertable spine`

  A metavariable could not be solved due to it being in a spine which is not made up of solely function applications.

- `pirec: bug`

  A bug occurred in the program which is not supposed to happen, an issue can be opened in the GitHub repository.

## 3.4 Lexical structure

EBNF as specified by ISO [41] is used to describe the lexical structure in this section.

Pirec is an indentation sensitive language utilising the off-side rule, like Haskell [56], Agda [66], and Python [71]. The control characters `\t`, `\r`, `\f`, `\v` and all Unicode space characters are considered white space characters and are ignored except for delimiting tokens and counting towards indentation levels. Each space character counts as one

indentation level except for tab (`\t`) which counts as 8. The newline character (`\n`) is also considered white space, but it resets the indentation level to zero.

Line comments start with `--` and ends at a newline, block comments are surrounded by `{-` and `-}`, they are ignored by the parser. Block comments can be nested and does not need to be closed if it ends at the end of the file.

The syntax contains nested *blocks*, each block contains *lines*, the keywords `let`, `#`, and `rec` start a new block. The block can be denoted by indendation, or alternatively it can be surrounded by braces after the block starting keyword and lines can be separated by semicolons to have the particular block be indentation insensitive. To denote a block with indentation, each line inside the block needs to be more indented than the enclosing block and be at the same indentation as each other, the end of the block happens before decreased indentation.

```
block_begin = ? either "{" or increased indentation
                matching with the corresponding block tokens ? ;
block_end   = ? either "}" with optional ";" before it
                or decreased indentation matching
                with the corresponding block tokens ? ;
block_sep   = ? either ";" or the same indentation
                matching with the corresponding block tokens ? ;
```

A line in a block can be split across multiple lines and still be parsed as the same thing, to do this, each token after the first one in the line needs to be more indented than the enclosing block, and the token at the end of a line break cannot be a block starting keyword.

Identifiers can consist of any character in the Letter, Mark, Number, Punctuation, or Symbol category of Unicode, except for some characters in ASCII: parentheses, braces, semicolons, backslashes, periods, and double quotes.

```
special_char = "(" | ")" | "{" | "}" | ";" | "\" | "." | '"' ;
ident_char   = ? any character not in the Separator
                 or Other category of Unicode ? - special_char ;
```

Identifiers must not be keywords, binders can have underscore in place of identifiers, string literals can be used to denote field labels when it cannot be written as an identifier.

```
keyword      = "_" | "let" | ":=" | ":" | "Type"
             | "→" | "->" | "∀" | "forall" | "λ"
             | "×" | "&" | "∃" | "exists" | "," | "proj1" | "proj2"
             | "Row" | "#" | "|" | "Rec" | "rec" | ":=" ;
ident        = { ident_char } - keyword ;
binder_ident = ident | "_" ;
field_label  = ident | ? string literal enclosed with " ? ;
```

Each Unicode token has an ASCII alternative which can be used instead (see table 3.1).

```
arrow  = "→" | "->" ;
forall = "∀" | "forall" ;
lambda = "λ" | "\" ;
times  = "×" | "&" ;
exists = "∃" | "exists" ;
```

## 3.5 Syntax

The program as a whole is a let block without the starting `let` token, which will be describe further below.

```
program = let_block ;
```

Expressions can consist of many operators with differing precendence and fixity (see table 3.2), function application can be considered as an operator denoted by juxtaposition, the argument can be denoted as implicit with braces. The keywords `proj1`, `proj2`, `Row`, and `Rec` act like functions that must be applied to an argument.

```
expr       = times_expr | times_expr , arrow , expr ;
times_expr = comma_expr | comma_expr , times , times_expr ;
comma_expr = app_expr | app_expr , "," , comma_expr ;
app_expr   = proj_expr | app_expr , proj_expr | app_expr , "{" proj_expr "}"
           | ( "proj1" | "proj2" | "Row" | "Rec" ) , proj_expr ;
proj_expr  = atom | proj_expr , ( "." | ".-" ) , field_label ;
```

The atomic expressions starting with `let`, `∀`, `λ`, `∃`, `#`, or `rec` when they do not have a delimiter at the right side can lead to ambiguous grammar, hence there is an extra rule

| Operator | Fixity |
|---|---|
| `.` and `.-` | Left |
| function application | Left |
| `,` | Right |
| `×` | Right |
| `→` | Right |

Table 3.2: Pirec operators and their fixity ordered by their precedence (from highest to lowest precedence)

which says that those expressions extend as far to the right as possible. The same method is used in GHC's BlockArguments language extension [3, 26].

```
atom = "(" , expr , ")" | ident | "_" | let_expr | "Type"
     | pi_expr | lam_expr | sigma_expr | row_expr | rec_expr ;
```

Let expressions start with a `let` keyword and it starts a *let block*. Each line of the block is a definition with optional types, the last line is a single term which is the result.

```
let_expr  = "let" , let_block ;
let_block = block_begin , { let_def , block_sep } , expr , block_end ;
let_def   = binder_ident , [ ":" , expr ] , "=" , expr ;
```

The syntax of dependent function types and lambda abstractions are similar, the only difference is the starting keyword. They can have multiple parameter binders, each one is an identifier, or it can be surrounded with parentheses then given a type, or braces to denote it as an implict argument.

```
pi_expr      = forall , fun_binder , { fun_binder } , arrow , expr ;
lam_expr     = lambda , fun_binder , { fun_binder } , arrow , expr ;
fun_binder   = binder_ident | "(" , binder_ident , [ ":" , expr ] , ")"
                            | "{" , binder_ident , [ ":" , expr ] , "}" ;
```

Dependent pairs are similar in syntax to dependent functions, except for using `∃` and `×`, and not allowing implicit binders.

```
sigma_expr = exists , sigma_binder , { sigma_binder } , times , times_expr ;
sigma_binder = binder_ident | "(" , binder_ident , [ ":" , expr ] , ")" ;
```

Rows literals start with a hash character, and starts a block. The block can be empty, or it can contain fields, which can be an extension to another row denoted by a pipe character.

```
row_expr   = "#" , block_begin , [ row_fields ] , block_end ;
row_fields = { row_field , block_sep } , row_field , [ row_ext ] ;
row_field  = field_label , ":" , expr ;
row_ext    = [ block_sep ] , "|" , expr ;
```

Record literals have a similar structure to row literals, the extension is the same. However, record fields are defined with the equals sign, or with `:=` for updating the content of a field, they can also be annotated with optional types. Besides these, a record field can just be a single identifier, which is syntactic sugar for defining it to be a variable with the same name.

```
rec_expr   = "rec" , block_begin , [ rec_fields ] , block_end ;
rec_fields = { rec_field , block_sep } , rec_field , [ row_ext ] ;
rec_field  = ident | field_label , [ ":" , expr ] , ( "=" | ":=" ) , expr ;
```

Some of the syntax is syntactic sugar, which will be desugared after parsing to get a simpler AST (see figs. 3.2 and 3.3, optional typing and implicit arguments are not shown but they work analogously to their counterparts).

## 3.6 Type system and Semantics

The typing rules of Pirec are presented in fig. 3.4, some equalities in for typing are shown in fig. 3.5, holes and unannotated binders will have metavariables inserted, then the program will try to solve them.

The row polymorphism and extensible record implementation is based on [51], so rows and records can contain multiple fields, and the most recently extended field “shadows” but does not remove fields with the same name, shadowed fields are still accessible by removing the fields “on top” of them. Fields with different labels are still unordered relative to one another.

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

Figure 3.2: Syntactic sugar in Pirec

The evaluation requires the definition of neutral terms, which are terms with a variable and actions on that variable that cannot proceed due to it being a variable. It is shown in fig. 3.6. The big-step operational semantics of evaluating Pirec terms to normal form is defined in Figure 3.7.

| | |
|---|---|
| $t, u, v, w, r, A, B, R ::= x$ | 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 |
| $\mid \exists(x\ :\ A) \times B$ | dependent pair |
| $\mid t, u$ | pair constructor |
| $\mid \mathsf{proj1}\ t$ | first projection |
| $\mid \mathsf{proj2}\ t$ | second projection |
| $\mid \mathsf{Row}\ A$ | 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 3.3: The desugared terms of Pirec

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

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

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

$$\frac{\Gamma \vdash A\ :\ \mathsf{U} \qquad \Gamma,\ x\ :\ A \vdash B\ :\ \mathsf{U}}{\Gamma \vdash \exists(x\ :\ A) \times B\ :\ \mathsf{U}} \qquad \frac{\Gamma \vdash t\ :\ A \qquad \Gamma \vdash u\ :\ B}{\Gamma \vdash t, u\ :\ \exists(x\ :\ A) \times B}$$

$$\frac{\Gamma \vdash A\ :\ \mathsf{U} \qquad \Gamma,\ x\ :\ A \vdash B\ :\ \mathsf{U}}{\Gamma \vdash \exists(x\ :\ A) \times B\ :\ \mathsf{U}} \qquad \frac{\Gamma \vdash t\ :\ A \qquad \Gamma \vdash u\ :\ B[x \coloneqq t]}{\Gamma \vdash t, u\ :\ \exists(x\ :\ A) \times B}$$

$$\frac{\Gamma \vdash t\ :\ \exists(x\ :\ A) \times B}{\Gamma \vdash \mathsf{proj1}\,t\ :\ A} \qquad \frac{\Gamma \vdash t\ :\ \exists(x\ :\ A) \times B}{\Gamma \vdash \mathsf{proj2}\,t\ :\ B[x \coloneqq \mathsf{proj1}\,t]}$$

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

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

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

Figure 3.4: The typing rules of Pirec

$$\frac{}{(\lambda x \to t)\,u \equiv t[x \coloneqq u]} \qquad \frac{}{(\lambda x \to t\,x) \equiv t}$$

$$\frac{}{\mathsf{proj1}(t, u) \equiv t} \qquad \frac{}{\mathsf{proj2}(t, u) \equiv u} \qquad \frac{}{(\mathsf{proj1}\,t, \mathsf{proj2}\,t) \equiv t}$$

$$\frac{l \neq l'}{\#\{l = t \mid \#\{l' = u \mid r\}\} \equiv \#\{l' = u \mid \#\{l = t \mid r\}\}}$$

$$\frac{l \neq l'}{\mathsf{rec}\{l = t \mid \mathsf{rec}\{l' = u \mid v\}\} \equiv \mathsf{rec}\{l' = u \mid \mathsf{rec}\{l = t \mid v\}\}}$$

$$\frac{}{\mathsf{rec}\{l = t \mid u\}.l \equiv t} \qquad \frac{l \neq l'}{\mathsf{rec}\{l' = t \mid u\}.l \equiv u.l} \qquad \frac{}{\mathsf{rec}\{l_1 = t.l_1; \ldots; l_n = t.l_n\} \equiv t}$$

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

Figure 3.5: Non-trivial equality rules in Pirec

$$
\begin{array}{rr}
n ::= x & \text{variable} \\
\mid n\,t & \text{function application} \\
\mid \mathsf{proj1}\,n & \text{first projection} \\
\mid \mathsf{proj2}\,n & \text{second projection} \\
\mid \#\{l \,:\, t \mid n\} & \text{row extension} \\
\mid \mathsf{rec}\{l = t \mid n\} & \text{record extension} \\
\mid n.l & \text{record projection} \\
\mid n.{-}l & \text{record restriction}
\end{array}
$$

Figure 3.6: The neutral terms of Pirec

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

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

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

$$
\frac{A \Downarrow A' \qquad B \Downarrow B'}{\exists(x \,:\, A) \times B \Downarrow \exists(x \,:\, A') \times B'} \qquad \frac{t \Downarrow t' \qquad u \Downarrow u'}{t, u \Downarrow t', u'}
$$

$$
\frac{t \Downarrow u, v}{\mathsf{proj1}\,t \Downarrow u} \qquad \frac{t \Downarrow n}{\mathsf{proj1}\,t \Downarrow \mathsf{proj1}\,n} \qquad \frac{t \Downarrow u, v}{\mathsf{proj2}\,t \Downarrow v} \qquad \frac{t \Downarrow n}{\mathsf{proj2}\,t \Downarrow \mathsf{proj2}\,n}
$$

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

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

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

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

Figure 3.7: The big-step operational semantics of Pirec

# Chapter 4

# Developer documentation

## 4.1 Project structure

The project is implemented in *Haskell* [56], using *Glasgow Haskell Compiler* (GHC) version 8.10.4 [26] with most warnings and extensions enabled by default, notable extensions include:

**NoImplicitPrelude** To not have the default standard library imported implicitly in every module, instead using Relude (see below).

**Derive*** To derive many type classes automatically, for example every type has automatically derived `Show` instance where possible for debugging purposes.

**TemplateHaskell** To enable automatic optics generation using metaprogramming (see section 4.1.1).

**OverloadedLabels** To use the automatically generated optics with shorter syntax, a hash character followed by the record field name.

**StrictData** To have data type definitions be strict to improve performance and avoid space leaks [63].

the project and its dependencies are managed by *Stack* [76], with package description written in the *Hpack* format [32]. Continuous integration is setup using *GitHub Actions* [29] with actions provided from [30] to setup Stack.

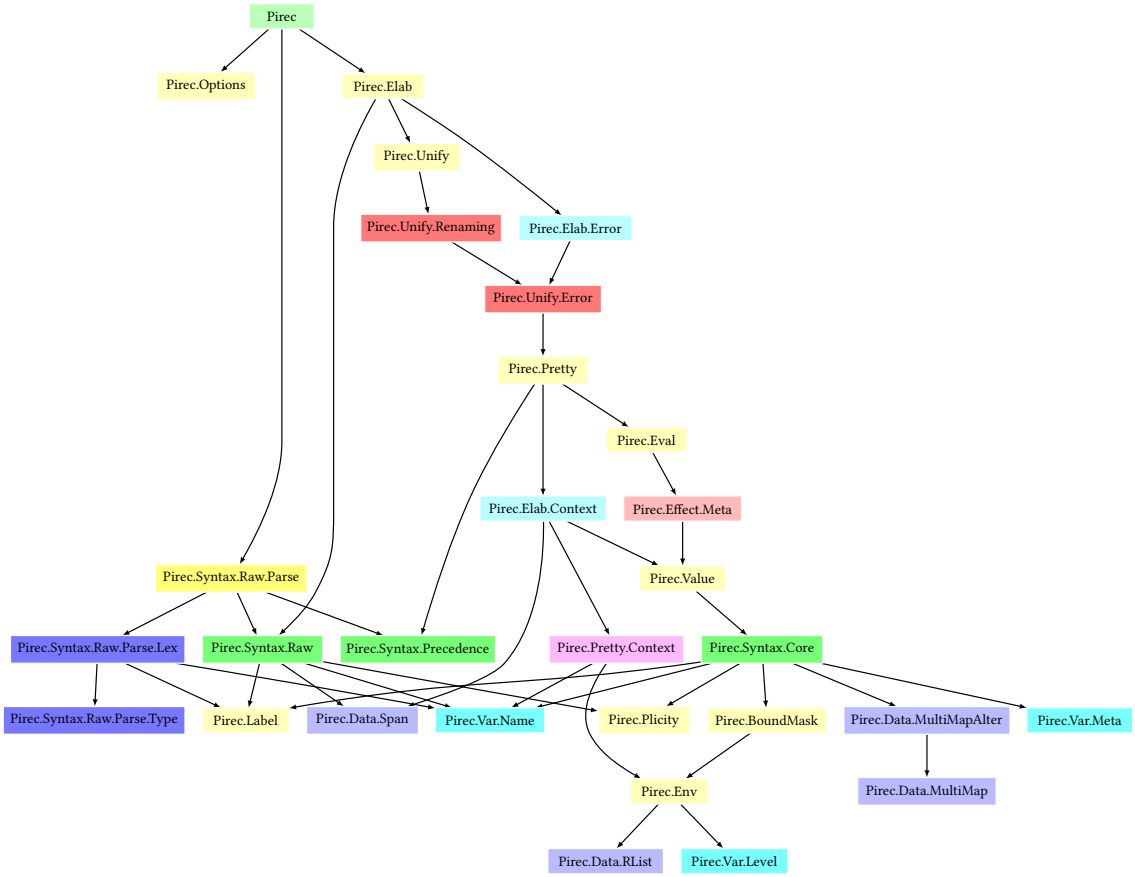


Figure 4.1: Transitively reduced dependency graph of the Haskell modules

An alternate Haskell standard library ("prelude") called *Relude* [49] is used, and it is imported in every module. The source code is checked with *Hlint* [62] with rules provided by Relude and formatted with *Fourmolu* [68] with indentation size of two spaces.

Many ideas about the project and the algorithms are from [48]. The module dependencies of the project are shown in fig. 4.1.

To build the project, run `stack build` inside the repository, to run the tests, run `stack test`, automatically generated documentation with the types of exported functions can be generated with `stack haddock`

### 4.1.1 Optics

The project makes active use of lenses and optics using the *Optics* library [31] in Haskell.

*Lenses* or *functional references* were originally an abstraction to help update nested data types [25], then an encoding using functors was discovered and other variations of lenses were found [46, 50], these collectively are known as *optics*. The Optics library is based

on a more principled profunctor encoding [69], it has an abstract interface, and it can automatically generate lenses using TemplateHaskell which can be used with overloaded labels [31].

### 4.1.2 Effect system

For functions with effects, the project makes use of an higher-order effect system library called *in-other-words* [85].

Monads are a way to manage and reason about computational effects, however defining a different monad for each combination of effects is not modular. Monad transformers are a way to combine effects together in a modular way by stacking types [53], and type classes can be used to have fine-grained control over the effects and to have multiple ways of interpreting effects, it is implemented by the *MTL* library [27]. However, it has some limitation such as there cannot be multiple variants of the same effect, or that defining a new transformer requires writing $O(n^2)$ amount of instances where $n$ is the amount of existing transformers [24].

*Extensible effects* are an alternative to monad transformers without the limitations [45], Haskell libraries usually use a version of free monads. Usually one can only define first-order effects [22], but there are higher-order effect systems [55, 87] using the methods in [86]. Unfortunately this approach has some semantic issues [84], which is what the in-other-words library aims to solve [85].

The performance of using an effect system is suboptimal, though there are efforts to improve on that [43, 44]

## 4.2 Raw Syntax

The raw syntax is the unprocessed AST after desugaring, it is represented as a recursive ADT with many constructors defined at `Pirec.Syntax.Raw` (see listing 1).

The `Span` constructor denotes the range of characters the term spans, the type `Span` stores the starting and ending offset in the source code, it is defined at `Pirec.Data.Span`. Every other contructor correspond to a syntactic construct in fig. 3.3.

The type `Name` represents a variable name, it is a newtype over the the type `ShortText` defined at `Pirec.Var.Name` which is a compact UTF-8 string [73]. Wildcard binders are

```
data Term
  = Span Span Term
  | Var Name                                -- x
  | Hole                                    -- _
  | Let Name (Maybe Term) Term Term         -- let{ x : a = t; u }
  | Univ                                    -- Type
  | Pi Plicity Name (Maybe Term) Term       -- ∀ (x : A) → B
  | Lam Plicity Name (Maybe Term) Term      -- λ (x : A) → t
  | App Plicity Term Term                   -- t u
  | Sigma Name (Maybe Term) Term            -- ∃ (x : A) × B
  | Pair Term Term                          -- t , u
  | Proj1 Term                              -- proj1 t
  | Proj2 Term                              -- proj2 t
  | RowType Term                            -- Row t
  | RowEmpty                                -- #{}
  | RowExt Label Term Term                  -- #{ l : t; r }
  | RecordType Term                         -- Rec R
  | RecordEmpty                             -- rec{}
  | RecordExt Label (Maybe Term) Term Term  -- rec{ l : a = t; u }
  | RecordProj Label Term                   -- t.l
  | RecordRestr Label Term                  -- t.-l
```

Listing 1: Raw syntax ADT

represented with the name of a single underscore, a pattern synonym is defined for it as `Wildcard`. `Label` represents a row or record field label, it has the same representation as `Name`, but it is a different type for type safety, it is defined at `Pirec.Label`.

The constructors related to dependent functions have a field with type `Plicity` defined at `Pirec.Plicity`, it denotes whether the argument is implict, it has two possible values, `Explicit` and `Implicit`. The fields with type `Maybe Term` are optional annotations of the types.

## 4.3 Core Syntax

The core syntax (listing 2) is an AST similar to the raw syntax, it is gotten after the variables and types are resolved, so it must be well-typed. It is defined at `Pired.Syntax.Core`.

Instead of names, the `Var` constructor here takes a de Bruijn level instead (see section 2.2.3), where `Level` is a newtype over `Int` defined at `Pirec.Var.Level`. For constructors which bind variables, names are not needed with de Bruijn levels, however it is retained in the syntax for prettyprinting purposes.

The `Meta` constructor represents metavariables, which are unsolved types and terms that are assigned to unique variables. The `Meta` type is a newtype over `Int` defined at `Pirec.Var.Meta`. The constructor also takes an optional `BoundMask`, which is a mask over the environment which selects all the bound variables (in contrast to variables defined in let expressions), it is defined at `Pirec.BoundMask`, it is used when inserting metavariables, so when it is evaluated, the metavariable is applied to all the bound variables in scope [48].

### 4.3.1 Multimaps

The rows and records are normalized to multimaps so that they can be compared easily without taking order of keys into account. `RowLit` stands for a row with some fields, and `RowExt` represents extension of some fields. Similarly, `RecordLit` is a record with some fields, but `RecordAlter` is an alteration of a record, meaning both the extension and the restriction of some fields. `RowExt` and `RecordAlter` have the invariant that the multimap cannot be empty.

`MultiMap` is a type defined at `Pirec.Data.MultiMap`, it is a newtype over unordered map of ordered sequences. It has the invariant that each sequence is non-empty. The

```
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
  | Sigma Name Term Term
  | Pair Term Term
  | Proj1 Term
  | Proj2 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

keys and values are generic type variables to provide type safety for defining general multimap manipulation functions.

Several typeclasses are automatically derived for `MultiMap`, such as `Eq` and `Traversable`. `Semigroup` and `Monoid` instances are defined for merging multimaps. An instance for the `Ix` typeclass from Optics (see section 4.1.1) is also defined an affine traversal for indexing multimaps with a key and a sequence index.

Operations defined include `match`, which returns a new multimap if the two input multimaps have the same keys and the same amount of values at each key, the new values are calculated with a given input function.

Another operation is `difference`, which calculates the difference of two multimaps. If both multimaps have amount of values at a key, then as many are dropped from the first multimap as there are in the second multimap.

### 4.3.2 MultiMapAlter

Another related type is `MultiMapAlter`, which represents alteration to a multimap (insertion and deletion). It is defined at `Pirec.Data.MultiMapAlter`.

It is a newtype over `ElemAlter`, which represent the insertion and deletion of at a single key. `ElemAlter` is defined to be the product of an integer and a sequence. The integer is the amount to remove at a key, and the sequence is the values to insert at a key.

`MultiMapAlter` has the invariant that `ElemAlter` cannot be an alteration that does nothing. This is checked with `elemIsId`, which checks that the integer is not zero and the sequence is not empty. The function `elemDrop` compares an integer and the length of a sequence, then subtracts from the integer as much as the length of the sequence or drops from the sequence as much as the integer, depending on whichever is larger, finally it embeds the result into `ElemAlter`.

The `Semigroup` instance of `ElemAlter` (listing 3) is the composition of alterations. Removal from the left cancels the insertion from the right alteration, while the others are added to the remains.

`MultiMapAlter` also has a `Semigroup` instance which is the composition. Each element is composed with `ElemAlter`'s `Semigroup` instance, if an element's result is the identity alteration, then the key gets removed from the `MultiMapAlter`.

The function `apply` applies `MultiMapAlter` to a multimap. It applies elementwise and

```
instance Semigroup (ElemAlter a) where
  ElemAlter i xs <> ElemAlter j ys = ElemAlter (i' + j) (xs <> ys')
   where
    ElemAlter i' ys' = elemDrop i ys
```

Listing 3: The `Semigroup` instance of `ElemAlter`

```
data Value
  = Neut Var Spine
  | Univ
  | Pi Plicity Name ~Value Closure
  | Lam Plicity Name Closure
  | Sigma Name Value Closure
  | Pair Value ~Value
  | RowType Value
  | RowLit (MultiMap Label Value)
  | RecordType Value
  | RecordLit (MultiMap Label Value)
```

Listing 4: Value ADT

the key is deleted when the resulting sequence at a key is empty. The function `lookup` takes a key and an index, then returns the value if it exists at that key and index, otherwise it returns another index, but subtracing the length of the sequence and adding the number of removals to the original index. This is to ensure that the indexing an inner record will correct after the alterations.

The `match` function is similar to the `match` function of `MultiMap`, but here the integer also needs to match, in addition to the sequence. There is an indexed fold optic, which folds over the elements insertion for each key, but gives a `Nothing` for each element removal.

## 4.4 Values

Values (listing 4) are what terms from core syntax are evaluated to, it is defined at `Pirec.Value`.

```
data Spine
  = Nil
  | App Plicity Spine ~Value
  | Proj1 Spine
  | Proj2 Spine
  | RowExt (MultiMap Label Value) Spine
  | RecordProj Label Int Spine
  | RecordAlter (MultiMapAlter Label Value) Spine
```

Listing 5: Spine ADT

First, there is the `Neut` constructor, which represents neutral values (see fig. 3.6). It takes a variable and a spine. The variable is either a rigid debruijn level (`Rigid`), or a flex metavariable (`Flex`). The spine is the AST of the neutral value, where `Nil` is where the variable would go. But instead it is an argument of the `Neut` so that one does not have to traverse through the spine to check whether the variable is rigid or flex.

Let expression is not represented in values, since it is not in normal form. All other constructors are similar to the structure of the core syntax, except that `Pi`, `Lam`, and `Sigma` have closures in their body, they all bind a variable. `Closure` is a product of an environment of values (see section 4.4.1) and a term, it represents a delayed evaluation of the term where one can insert a value to the environment to substitute the bound variable.

The function argument of `App` and the second component of `Pair` are lazy to improve performance during unification. The type of `Pi` is lazy because it might not be needed to be evaluated when typechecking.

Several smart constructors are defined for operators. The functions `proj1` and `proj2` take their respective element from a pair if the argument is a pair, otherwise the respective constructor is added to the spine. The `rowExt` function (listing 6) extends a row literal if the argument is such, and it also extends a row extension in the spine if the spine does have an outer row extension. The `recordProj` function can select the field from a record literal, or it could lookup from a record alteration, in this case, it uses `MultiMapAlter`'s lookup, and if the element is not found, then the new spine has the record projection with the new index. The `recordAlter` function can alter a record literal, or it could compose

```
rowExt :: MultiMap Label Value -> Value -> Value
rowExt ts = \case
  Neut x spine -> Neut x case spine of
    RowExt us spine -> RowExt (ts <> us) spine
    _ -> RowExt ts spine
  RowLit us -> RowLit (ts <> us)
  _ -> error "bug"
```

Listing 6: The `rowExt` function

with another `RecordAlter` in the spine, but in this case, if the composition is the identity, then the `RecordAlter` disappears.

### 4.4.1 Environment

The environment to be used with de Bruijn levels should be a type that can be extended from the end and indexed with a level. For asymptotic performance, the `Env` type defined at `Pirec.Env` is implemented as a random access list based on [67].

The random access list is defined as `RList` at `Pirec.Data.RList`. The elements are lazy to support evaluating values lazily. Operations such as `cons` and `uncons` are defined. Indexing is defined as an instance of the Optics library's `Ix` typeclass, it is defined with the van Laarhoven based optics encoding [50], so that getting and updating at an index do not need to be both defined. GHC specialization pragmas are used so that the defining that way does not harm performance. A `TraversableWithIndex` instance is also defined similary, however the `Env` which is a newtype over `RList` must reverse it and its indices.

## 4.5 Parsing

A monadic parser combinator [40] library called *Megaparsec* [58] is used to do both lexing and parsing at the same time.

*Parser combinators* can be implemented as an embedded domain specific language in a library. They provides primitive parsers and combinators which can be used to combine parsers into more complicated parsers. Ultimately it is a top-down recursive descent parser, so it cannot directly parse left recursive grammar. Having a monadic interface

```
type Parser = ReaderT ParseContext (StateT ParseState (Parsec Void Text))

newtype ParseContext = ParseContext
  { blockIndent :: Int
  }

data ParseState = ParseState
  { lexemeEnd :: Int
  , lineStart :: Bool
  }
```

Listing 7: The Pirec parser type definitions

means that the grammar they can parse is no longer restricted to being context-free and it allows the freedom to reformulate left recursion into right recursion which gets transformed afterwards [40, 78].

The Megaparsec library was originally forked from *Parsec* [52]. It has an interface compatible with MTL [27], built-in error reporting and customizable errors, and combinators to work with indentation [58].

### 4.5.1 Parser type

The type definition for the Pirec parser defined at `Pirec.Syntax.Raw.Parse.Type` is show in listing 7. Reader and state monad tranformer from MTL [27] are used for some extra parsing context and state, which are records, and their fields have overloaded label lenses (see section 4.1.1) that are automatically generated.

The `lexemeEnd` field in the state is used to track the offset of the end of the last token parsed, so that it can be stored in a source code span for error reporting without including the following white space.

The `blockIndent` and the `lineStart` field are both for indentation sensitivity. The `blockIndent` field is for tracking the indentation of the current block, it is in a reader context so that `local` can be used to modify it only in an environment. The indentation level for `blockIndent` at the start is 0. The `lineStart` field is to signify whether the next token is at the start of the line, depending on it, a token's indentation level either should

```
lexeme :: Parser a -> Parser a
lexeme p = do
  lineStart <- use #lineStart
  _ <- indentGuard (if lineStart then EQ else GT)
  x <- p
  assign #lexemeEnd =<< getOffset
  assign #lineStart False
  x <$ space
```

Listing 8: The `lexeme` function

be equal to the block's indentation level, or greater than it.

### 4.5.2 Parsing tokens

The parsers for the tokens are at `Pirec.Syntax.Raw.Parse.Lex`. The parser for white space (`space`) uses Megaparsec library's utility function, it also parses comments based on user supplied comment parsers, all of which are ignored by the parser. The line comment parser uses the function supplied by Megaparsec, the block comment parser is defined so that closing the comment is optional, and to allows nesting.

For parsing tokens, the `lexeme` function is defined (see listing 8), every token parser will be wrapped with it. It takes a parser as input and returns a new parser that takes indentation into account and ignores the white space following the token. First it checks whether the current token is at the start of a block's line, if it is, then check if it is at the same indentation as the current block, otherwise check if it has greater indentation. Afterwards use the given parser then get the position and set `lineStart` to false and parse the following white space.

The function `indentGuard` wraps Megaparsec's function with the same name so that it uses `blockIndent` from the reader context, which can be 0, not just a positive number.

To parse a token which cannot be an identifier, the parser `symbol` should be used, otherwise it is a keyword and the parser `keyword` should be used instead. The difference is that `keyword` checks that the character following the token is not an identifier character. All the special symbols and keywords are defined using `symbol` or `keyword`

The function `isIdentChar` is used to check if a character is an identifier character

```
indentBlock :: (Parser () -> Parser a) -> Parser a
indentBlock f =
  choice
    [ braces $ f (void semicolon) <* optional semicolon
    , do
        blockIndent <- indentGuard GT
        local (#blockIndent .~ unPos blockIndent) $
          assign #lineStart True *> f (assign #lineStart True)
    ]
    <?> "indented block"
```

Listing 9: The `indentBlock` function

(see section 3.4), it checks using Unicode categories, and it cannot be a special character declared in `specialChars`. Identifiers are parsed as a sequence of one or more identifier characters, and it is checked that it is not a keyword.

The parser `ident` is for identifier, `binderIdent` is for identifiers in binders as it allows wildcards. The parser `fieldLabelOrName` parses a field label, but besides the label, it also returns a `Name` if it is a valid identifier. Since labels can be string literals, it is not a valid identifier in that case.

The function `indentBlock` (listing 9) is for parsing blocks, which can be denoted with both braces and indentation (see section 3.4). It takes a function which takes a line separator and returns a parser for parsing the inside of the block. The parser first tries to parse the block surrounded by braces, with semicolon as line separator and with an optional trailing semicolon. If it fails, the parser checks if the current indentation level is greater than the outer block's, then sets the indentation level for the current block to the current level in the context of parsing the inside of the block, which is parsed by having `lineStart` be true at the start and for the separator.

### 4.5.3 Parsing syntax

The function for parsing a whole program is defined at `Pirec.Syntax.Raw.Parse` called `parseRaw`. It first parses some leading white space, then parses a let block (see section 3.5), and at the end checks if it is at the end of the file.

Expression parsers are written with precedences in mind (see table 3.2). There is the `Precedence` enumeration defined at `Pirec.Syntax.Precedence`. It is used so that one can add a new precedence between existing ones without modifying the parsers of others, as one can automatically derive the `Enum` typeclass for Haskell enumerations, which contains the `succ` function for getting the next element in the enumeration.

The `term` parser parses an expression at the lowest precedence. The `termPrec` function takes a precedence and returns a parser that parses the constructs at that precedence.

All subexpressions are wrapped with `withSpan`, which is a function that takes a term parser and wraps it in the `Span` constructor (see section 4.2) with the starting offset and the offset in the `lexemeEnd` field.

Arrows, times signs, and commas are all right associative, so they use the `infixRight` function, which takes a precedence and parses a right associative operator, with one higher precedence as subexpressions.

Function application and record projection/restriction are both left associative, they use the `suffixes` function, which takes a starting parser, and parse suffixes one by one from other parameter. For function application, the function arguments can be considered as suffixes which can be surrounded by braces if they are implicit. The function application also considers `proj1`, `proj2`, `Row`, and `Rec` applied to arguments as functions.

Atoms are at the highest precedence, expressions starting with `let`, `∀`, `λ`, `∃`, `#`, or `rec` can be considered atomic as the parser automatically parses as much as possible (see section 3.5).

Let blocks are parsed using `indentBlock` (listing 10). Inside the block it parses zero or more definitions terminated with the separator, at the end it parses an extra term as the return value of the let expression. At the start of each definition, the parser first tries to parse a binder identifier followed by a colon or an equals sign. It needs to be able to backtrack using `try` here, since the identifier might be a part of the final term. The optional type could have been part of the argument of `try` which would have been simpler, but instead that line returns an extra Boolean to denote whether it expects a type, this way it would not backtrack after parsing a colon since it cannot be the final term anymore.

Dependent function types, lambda abstraction, and dependent pair types all have similar structure, particularly the binders. The binders of dependent functions types

```
termLetBlock :: Parser R.Term
termLetBlock = indentBlock \sep -> do
  fs <- many do
    (x, hasType) <-
      try $ (,) <$> binderIdent <*> (True <$ colon <|> False <$ equals)
    a <- if hasType then Just <$> term <* equals else pure Nothing
    R.Let x a <$> term <* sep
  foldr (.) id fs <$> term
```

Listing 10: The parser for let blocks

and lambda abstraction are parsed with `binderImpl`, it can parse bare binder variables, or variables surrounded by parentheses or braces with optional type after the variable. Dependent pair types are parsed with a different `binder`, which does not allow binders.

Row blocks and record blocks are similarly structured, as both extend the same way (see section 3.5), so an extra `rowBlock` function is defined. It takes an empty row or record, and a parser to parse each field, then returns a parser which parses a block. The inside of the block has the option to be empty, or it can be one or more fields separated by the separator. At the end, there can be an optional pipe, which a term follows. There can be an extra separator before the pipe.

Row literals are parsed straightforwardly using `rowBlock`, where each field is followed by a colon and a type. Record literal fields however can have optional types besides a term, they can have record modification syntax, and they can have a variable by itself which is both a label and a name, so it is parsed in a more complicated way (see listing 11).

To parse a record field, first, a label is parsed using `fieldLabelOrName`, the possibility of being a name will be used later. Then it tries to parse record extension or record modification without type annotation, so the next token would be an equals sign or colon equals, the record modification is directly desugared to record restriction then extension. If the token does not match equals sign or colon equals, then it will try parsing a colon then a type, then extension or modification with the type given. In the end, if all fails, check if the original label is compatible with a variable name, if so then desugar this to extending with that variable name.

```
termRecordLit :: Parser R.Term
termRecordLit =
  recordLit *> rowBlock R.RecordEmpty do
    (lbl, name) <- fieldLabelOrName
    let extend a = R.RecordExt lbl a <$ equals <*> term
        modify a = do
          t <- coloneq *> term
          pure $ R.RecordExt lbl a t . R.RecordRestr lbl
    choice
      [ extend Nothing <|> modify Nothing
      , do
          a <- colon *> term
          extend (Just a) <|> modify (Just a)
      , case name of
          Nothing -> empty
          Just name -> pure $ R.RecordExt lbl Nothing (R.Var name)
      ]
```

Listing 11: The parser for record literals

```
data MetaLookup :: Effect where
  MetaLookup :: MetaLookup m (Meta -> Maybe Value)

data MetaState :: Effect where
  FreshMeta :: MetaState m Meta
  SolveMeta :: Meta -> Value -> MetaState m ()

type MetaCtx = Bundle [MetaLookup, MetaState]
```

Listing 12: The definition of metacontext effects

## 4.6 Metacontext

The metacontext is a context of metavariables which are can be solved with definitions during elaboration. In the project, the library in-other-words is used to track metacontext as an effect. The effect definition is shown in listing 12.

The command `metaLookup` is used for returning a function that looks up a metavariable from the metacontext. It is written like this instead of having the command take a metavariable is so that evaluation can remain lazy through recursion. This command is separated into its own effect because a lot of functions only need read access to the metacontext.

The other two commands that make up the `metaCtx` effect are `freshMeta`, which returns a fresh new metavariable, and `solveMeta`, which solves a metavariable with a value and stores it in the metacontext.

The effect interpreter `runMetaLookup` only interprets `MetaLookup`, it takes a pure function, and uses it for the command purely.

The interpreter `metaCtxToIO` uses an integer reference and a hashtable to implement metacontext, the integer reference refers to the next fresh metavariable. For interpreting `metaLookup`, a `unsafeDupablePerformIO` is used, so that the lookup is only resolved when needed, this way, the lookup is pushed as late as possible, when the metavariable might have become solved. The interpreter returns all the metavariables with their possibly solved values at the end.

```
eval :: Eff MetaLookup m => Env Value -> Term -> m Value
eval env t = do
  mlookup <- metaLookup
  let goEnv !env = go
        where
         go = \case
           Var lx -> env & Env.index lx ?: error "bug"
           Let _ t u -> goEnv (env & Env.extend (go t)) u
           Lam pl x t -> V.Lam pl x (V.Closure env t)
           App pl t u -> appValuePure mlookup pl (go t) (go u)
           ...
  pure $ goEnv env t
```

Listing 13: A fragment of the `eval` function

## 4.7 Evaluation

*Normalization-by-evaluation* (NBE) with explicit closures is implemented to evaluate lambda terms to normal form [1, 48].

The `eval` function evaluates a term into a value in an environment (see listing 13). It has a recursive inner function which is in scope of the metavariabe lookup function. Variables index the environment, let expressions extend the environment and recurse on the body, lambda abstractions replace the body with a closure, and applications evaluate the function and applies it lazily over the evaluated argument. The other constructors are done in a straightforward way, with closures, or with the correspoinding smart value constructors.

There are many utility functions defined together, such as the smart constructor for applying a value. It must use metavariable lookup, since the function might be a lambda abstraction, then the body must be substituted using `eval`. Applying closures and applying spines are also defined.

The `forceValue` function (listing 14) takes a value and if it is a spine, it tries to solve using the newest metavariable, until it cannot be solved. This function should be used before pattern matching on a value.

The `quote` function turns a value back into a term. It is defined with the `quoteWith`

```
forceValue :: Eff MetaLookup m => Value -> m Value
forceValue t = case t of
  V.Neut (V.Flex mx) spine ->
    metaLookup ?? mx >>= \case
      Just t -> appSpine t spine >>= forceValue
      Nothing -> pure t
  _ -> pure t
```

Listing 14: The `forceValue` function

function to generalize over the variable handling and the accumulator because the `rename` function in unification has the same structure.

The `openClosure` and `closeValue` functions convert closures and values to each other. The function `normal` computes the normal form of a term by first evaluating it to a value, then quoting it back.

## 4.8 Unification

Untyped pattern unification is implemented similarly as in [48] at `Pirec.Unify`.

First, a `Renaming` type is defined at `Pirec.Unify.Renaming`, which contains a level and the renaming of each level below that level as a map. An `invert` functions is defined (see listing 15), which inverts a spine to a renaming. The inversion only works with a metavariable applied to unique rigid variables, otherwise error is thrown.

The `rename` function renames value according to a renaming, and quotes it at the same time, so it uses `quoteWith`. It checks if the variable is in scope for the renaming and checks whether the to be solved metavariable does not appear in the value, otherwise it throws errors.

The `solve` function (listing 16) tries to solve a metavariable given its spine and the value to match. It first inverts the spine to a renaming, then uses it to rename the value, then lambda abstractions are added based on implicitness of the spine with a recursive `abstract` function, and in the end the solution is set in the metacontext.

The `unify` function matches the inferred and the expected type. It recurses both types, and matches both to check that they have the same structure. Exceptions are the η-rules of functions, pairs, and records, row extensions and record alterations also needed to

```
invert :: Effs [MetaLookup, Throw UnifyError] m => Spine -> m Renaming
invert = go
 where
  go = \case
    V.Nil -> pure $ Renaming 0 mempty
    V.App _ spine t -> do
      Renaming lvl m <- go spine
      forceValue t >>= \case
        V.Neut (V.Rigid lx@(Level n)) V.Nil
          | Just _ <- m ^. at n -> throw $ Nonlinear lx
          | otherwise -> pure $ Renaming (lvl + 1) (m & at n ?~ lvl)
        t -> throw $ Nonvariable t
    _ -> throw NonInvertable
```

Listing 15: The `invert` function

```
solve ::
  Effs [MetaCtx, Throw UnifyError] m =>
  Level -> Meta -> Spine -> Value -> m ()
solve lvl meta spine t = do
  renaming <- Renaming.invert spine
  t <- rename meta renaming lvl t
  solution <- eval Env.empty $ abstract spine t
  solveMeta meta solution
```

Listing 16: The `solve` function

have some complex combination of patterns to unify as desired.

The error type of unification is defined at `Pirec.Unify.Error`, containing six of the errors in section 3.3. Prettyprinting is also defined there.

## 4.9 Elaboration

Bidirectional type-checking algorithm [19, 54] is used for type checking and elaboration. It is defined at `Pirec.Elab`

The elaboration is computed in a context which is defined at `Pirec.Elab.Context`. It contains the span of the term to be elaborated, the environment for evaluation, a map from names to levels and values, a context for prettyprinting, and a `boundMask`.

A reader effect containing the elaboration context is used for convenience and readability, several common operations are redefined to use the reader effect. Two major functions are defined, `check` and `infer`. They are defined together in a tuple with mutually recursive local functions for convenience.

The error type of elaboration is defined at `Pirec.Elab.Error`, it subsumes the unification errors and contains an additional two errors. Prettyprinting is defined through the interface given by Megaparsec, as it shows error position the source code visually.

## 4.10 Prettyprinting

Prettyprinting is defined at `Pirec.Pretty` with the help of the Prettyprinter [83] library with line breaking based on line length and text alignment, it is based on the work of [81].

Prettyprinting takes a context defined at `Pirec.Pretty.Context`, it contains an environment of names, and a set of names in use. The `prettyTermWith` takes the context, a precedence (see section 4.5.3), and a term. The term is recursively turned into a pretty document. For some special constructors, extra function is written to group the same constructors together, so that they can align.

## 4.11 Main

The rough process of executing the main program is show in fig. 4.2.

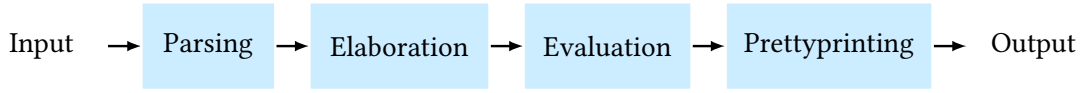


Figure 4.2: The steps of the main program

```
data Options = Options
  { input :: Input
  , eval :: Bool
  , showTerm :: Bool
  , showType :: Bool
  , showMetas :: Bool
  }

data Input
  = File FilePath
  | Stdin
  deriving stock (Show)
```

Listing 17: The type for the command line options

For parsing command line arguments, the library *optparse-applicative* [13] is used, it is based on the free applicative function to be able to introspect and automatically generate a suitable help text [14].

A type is defined for the command line options at `Pirec.Options` (see listing 17). The function `getOpts` parses a command line argument which is the file path, or it could parse `--stdin` which means that the input will come from the standard input. There are flags to control what to show, and whether to evaluate the term.

the `main` function is wrapped with `withUtf8` from the library *with-utf8* [23] to make `stdin`, `stdout`, and `stderr` work with UTF-8 regardless of system and locale. Then it parses the command line options and reads the input file. It parses and elaborates the input, then returns the elaborated term and its type, the metavariables, and the metavariable lookup function. In the end, depending on whether evaluation options is set to true, it normalizes the term, and prettyprints the things based on the options.

```
specify "cons" $ hedgehog do
  x <- forAll int
  xs <- forAll $ list 1000 int
  validToList (RList.cons x (fromList xs)) === Just (x : xs)
```

Listing 18: Testing `RList`'s `cons` function

## 4.12 Testing

The *Hspec* testing framework is used for testing [33], the package *Hspec-discover* is used for automatic test discovery in the `test` folder.

Data types and their operations defined in the project have property based random tests [18, 36] using Hedgehog [77], where each test case is checked many times with random input. The random data generators of Hedgehog have integrated shrinking over the applicative functor operations (shrinking is the process of searching for simpler counterexamples) [80].

Every operation of the `RList` type has property based test case to check that the type invariant is maintained and comparing its behavior with Haskell list's. As an example, the definition of the test for `cons` is shown in listing 18. It takes a random integer and a random list with length of maximum 1000, then converts the list to an `RList` and uses `cons` on it. The `validToList` function checks if the `RList` is valid, if so, it is converted to a normal list. In the end, it is compared to the consing of Haskell list.

Both `MultiMap` and `MultiMapAlter` also have property based test cases for every non-autmatically derived operation. The test cases check the invariants and properties such as the associativity of the semigroup operation.

The program is tested on a lot of examples in the `examples` folder, each one is type-checked and evaluated without error. There is an extra program at `examples/loop/hurkens.agda` translated from [65] based on [38], which shows that the typing is unsound since it has type in type. It is tested that it typechecks, and it does not terminate.

# Bibliography


[1] Andreas Abel, Klaus Aehlig, and Peter Dybjer. "Normalization by Evaluation for Martin-Löf Type Theory with One Universe". In: *Proceedings of the 23rd Conference on the Mathematical Foundations of Programming Semantics, MFPS 2007, New Orleans, LA, USA, April 11-14, 2007*. Ed. by Marcelo Fiore. Vol. 173. Electronic Notes in Theoretical Computer Science. Elsevier, 2007, pp. 17–39. DOI: 10.1016/j.entcs.2007.02.025.

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

[3] Takano Akio. *BlockArguments Extension*. 2017. URL: https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0090-block-arguments.rst (visited on 2021-05-29).

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] Henk Barendregt. "Introduction to Generalized Type Systems". In: *J. Funct. Program.* 1.2 (1991), pp. 125–154.

[9] Matthew Bauer et al. *Set -Wno-prepositive-qualified-module in Paths_*.hs*. URL: https://github.com/haskell/cabal/pull/7352 (visited on 2021-05-29).

[10] Corrado Böhm and Alessandro Berarducci. "Automatic Synthesis of Typed Lambda-Programs on Term Algebras". In: *Theor. Comput. Sci.* 39 (1985), pp. 135–154. DOI: 10.1016/0304-3975(85)90135-5.

[11] 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.

[12] Cabal Team. *Cabal 3.4.0.0 User's Guide*. Version 3.4. URL: https://cabal.readthedocs.io/en/3.4/ (visited on 2021-05-29).

[13] Paolo Capriotti and Huw Campbell. *Optparse-applicative: Utilities and Combinators for Parsing Command Line Options*. Version 0.15.1.0. 2020. URL: http://hackage.haskell.org/package/optparse-applicative-0.15.1.0 (visited on 2021-05-29).

[14] Paolo Capriotti and Ambrus Kaposi. "Free Applicative Functors". In: *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014*. Ed. by Paul Levy and Neel Krishnaswami. Vol. 153. EPTCS. 2014, pp. 2–30. DOI: 10.4204/EPTCS.153.2.

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

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

[17] 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.

[18] Koen Claessen and John Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18–21, 2000*. Ed. by Martin Odersky and Philip Wadler. ACM, 2000, pp. 268–279. DOI: 10.1145/351240.351266.

[19] 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.

[20] Thierry Coquand and Gérard P. Huet. "The Calculus of Constructions". In: *Inf. Comput.* 76.2/3 (1988), pp. 95–120. DOI: 10.1016/0890-5401(88)90005-3.

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

[22] Allele Dev, Ixcom Core Team, Alexis King, et al. *Freer-simple: Implementation of a Friendly Effect System for Haskell*. Version 1.2.1.1. 2021. URL: http://hackage.haskell.org/package/freer-simple-1.2.1.1 (visited on 2021-05-29).

[23] Kirill Elagin. "Haskell with UTF-8". Version 1.0.2.2. In: (2020). URL: https://serokell.io/blog/haskell-with-utf8 (visited on 2021-05-29).

[24] Ismael Figueroa, Nicolas Tabareau, and Éric Tanter. "Effect Capabilities for Haskell: Taming Effect Interference in Monadic Programming". In: *Sci. Comput. Program.* 119 (2016), pp. 3–30. DOI: 10.1016/j.scico.2015.11.010.

[25] J. Nathan Foster et al. "Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem". In: *ACM Trans. Program. Lang. Syst.* 29.3 (2007), p. 17. DOI: 10.1145/1232420.1232424.

[26] GHC Team. *Glasgow Haskell Compiler 8.10.4 User's Guide*. Version 8.10.4. 2021. URL: https://downloads.haskell.org/ghc/8.10.4/docs/html/users_guide/index.html (visited on 2021-05-29).

[27] Andy Gill et al. *MTL: Monad Classes, Using Functional Dependencies*. Version 2.2.2. 2018. URL: http://hackage.haskell.org/package/mtl-2.2.2 (visited on 2021-05-29).

[28] Jean-Yves Girard. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science 7. Cambridge University Press, 1989. ISBN: 9780521371810.

[29] GitHub, Inc. *GitHub Actions Documentation*. 2021. URL: https://docs.github.com/en/actions (visited on 2021-05-29).

[30] GitHub, Inc. et al. *Haskell Github Actions*. Version 1. 2020. URL: https://github.com/haskell/actions/tree/v1 (visited on 2021-05-29).

[31] Adam Gundry et al. *Optics: Optics as an Abstract Interface*. Version 0.4. Well-Typed LLP. 2021. URL: http://hackage.haskell.org/package/optics-0.4 (visited on 2021-05-29).

[32] Simon Hengel. *Hpack: A Modern Format for Haskell Packages*. Version 0.34.3. 2020. URL: https://github.com/sol/hpack/tree/0.34.3 (visited on 2021-05-29).

[33] Simon Hengel, Trystan Spangler, and Greg Weber. *Hspec: A Testing Framework for Haskell*. Version 2.7.8. 2021. URL: http://hspec.github.io (visited on 2021-05-29).

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

[35] 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.

[36] John Hughes. "QuickCheck Testing for Fun and Profit". In: *Practical Aspects of Declarative Languages, 9th International Symposium, PADL 2007, Nice, France, January 14-15, 2007*. Ed. by Michael Hanus. Vol. 4354. Lecture Notes in Computer Science. Springer, 2007, pp. 1–32. DOI: 10.1007/978-3-540-69611-7_1.

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

[38] Antonius J. C. Hurkens. "A Simplification of Girard's Paradox". In: *Typed Lambda Calculi and Applications, Second International Conference on Typed Lambda Calculi and Applications, TLCA '95, Edinburgh, UK, April 10–12, 1995, Proceedings*. Ed. by Mariangiola Dezani-Ciancaglini and Gordon D. Plotkin. Vol. 902. Lecture Notes in Computer Science. Springer, 1995, pp. 266–278. DOI: 10.1007/BFb0014058.

[39] 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.

[40] 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.

[41] International Organization for Standardization. *ISO/IEC 14977:1996 – Information technology – Syntactic metalanguage – Extended BNF*. 1996. URL: https://www.iso.org/standard/26153.html (visited on 2021-05-29).

[42] 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.

[43] Alexis King. *Delimited Continuation Primops*. 2021. URL: https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0313-delimited-continuation-primops.rst (visited on 2021-05-29).

[44] Alexis King et al. *Eff – Screaming Fast Extensible Effects for Less*. Hasura. 2020. URL: https://github.com/hasura/eff/tree/7d7f9ab77f3c7f473d52457e41e9b6e1870d72f6 (visited on 2021-05-29).

[45] Oleg Kiselyov, Amr Sabry, and Cameron Swords. "Extensible Effects: An Alternative to Monad Transformers". In: *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23–24, 2013*. Ed. by Chung-chieh Shan. ACM, 2013, pp. 59–70. DOI: 10.1145/2503778.2503791.

[46] Edward Kmett. *Lens: Lenses, Folds, and Traversals*. Version 5.0.1. 2021. URL: http://hackage.haskell.org/package/lens-5.0.1 (visited on 2021-05-29).

[47] 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.

[48] András Kovács. *Elaboration-zoo*. 2021. URL: https://github.com/AndrasKovacs/elaboration-zoo/tree/dc5904b092657267ef9eda22c56c8d907ff811db (visited on 2021-05-29).

[49] Dmitrii Kovanikov et al. *Relude: Safe, Performant, User-Friendly and Lightweight Haskell Standard Library*. Version 1.0.0.1. Kowainik. 2021. URL: http://hackage.haskell.org/package/relude-1.0.0.1 (visited on 2021-05-29).

[50] Twan van Laarhoven. "CPS Based Functional References". In: (2009). URL: https://www.twanvl.nl/blog/haskell/cps-functional-references (visited on 2021-05-29).

[51] 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.

[52] Daan Leijen and Erik Meijer. *Parsec: Direct Style Monadic Parser Combinators for the Real World*. Tech. rep. UU-CS-2001-27. 2001. URL: https://www.microsoft.com/en-us/research/publication/parsec-direct-style-monadic-parser-combinators-for-the-real-world/.

[53] Sheng Liang, Paul Hudak, and Mark P. Jones. "Monad Transformers and Modular Interpreters". 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. 333–343. DOI: 10.1145/199448.199528.

[54] 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.

[55] Sandy Maguire. *Polysemy: Higher-Order, Low-Boilerplate Free Monads*. Version 1.5. 2021. URL: http://hackage.haskell.org/package/polysemy-1.5.0.0 (visited on 2021-05-29).

[56] Simon Marlow, ed. *Haskell 2010 Language Report*. 2010. URL: https://www.haskell.org/onlinereport/haskell2010/ (visited on 2021-05-29).

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

[58] Megaparsec contributors, Paolo Martini, and Daan Leijen. *Megaparsec: Monadic Parser Combinators*. Version 9.0.1. 2020. URL: http://hackage.haskell.org/package/megaparsec-9.0.1 (visited on 2021-05-29).

[59] 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.

[60] Robin Milner. "A Theory of Type Polymorphism in Programming". In: *J. Comput. Syst. Sci.* 17.3 (1978), pp. 348–375. DOI: 10.1016/0022-0000(78)90014-4.

[61] Robin Milner, Mads Tofte, and Robert Harper. *Definition of Standard ML*. MIT Press, 1990. ISBN: 978-0-262-63132-7.

[62] Neil Mitchell. *HLint: Source Code Suggestions*. Version 3.2.7. 2021. URL: https://hackage.haskell.org/package/hlint-3.2.7 (visited on 2021-05-29).

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

[64] Rob Nederpelt and Herman Geuvers. *Type Theory and Formal Proof: An Introduction*. Cambridge University Press, 2014. ISBN: 9781107036505.

[65] Ulf Norell. [*Agda*] *A Plea For Set:Set*. URL: https://lists.chalmers.se/pipermail/agda/2008/000316.html (visited on 2021-05-29).

[66] Ulf Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. Doktorsavhandlingar vid Chalmers Tekniska Högskola N. S., 2677. Chalmers Univ. of Technology, 2007. ISBN: 9789172919969.

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

[68] Matt Parsons et al. *Fourmolu: A Formatter for Haskell Source Code*. Version 0.3.0.0. 2021. URL: http://hackage.haskell.org/package/fourmolu-0.3.0.0 (visited on 2021-05-29).

[69] Matthew Pickering, Jeremy Gibbons, and Nicolas Wu. "Profunctor Optics: Modular Data Accessors". In: *Art Sci. Eng. Program.* 1.2 (2017), p. 7. DOI: 10.22152/programming-journal.org/2017/1/7.

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

[71] Python Software Foundation. *The Python Language Reference*. Version 3.8.10. 2021. URL: https://docs.python.org/3.8/reference/index.html (visited on 2021-05-29).

[72] 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.

[73] Herbert Valerio Riedel. *Text-short: Memory-Efficient Representation of Unicode Text Strings*. Version 0.1.3. 2021. URL: http://hackage.haskell.org/package/text-short-0.1.3 (visited on 2021-05-29).

[74] 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.

[75] Morten Heine Sørensen and Pawel Urzyczyin. *Lectures on the Curry–Howard Isomorphism*. Vol. 149. Studies in Logic and the Foundations of Mathematics. Elsevier, 2006. ISBN: 9780444520777. DOI: 10.1016/S0049-237X(06)X8001-1.

[76] Stack contributors. *The Haskell Tool Stack*. Version 2.7.1. Commercial Haskell SIG. 2021. URL: https://docs.haskellstack.org/en/v2.7.1/ (visited on 2021-05-29).

[77] Jacob Stanley. *Hedgehog: Release With Confidence*. Version 1.0.5. 2021. URL: http://hackage.haskell.org/package/hedgehog-1.0.5 (visited on 2021-05-29).

[78] S. Doaitse Swierstra. "Combinator Parsing: A Short Tutorial". In: *Language Engineering and Rigorous Software Development, International LerNet ALFA Summer School 2008, Piriapolis, Uruguay, February 24 – March 1, 2008, Revised Tutorial Lectures*. Ed. by Ana Bove et al. Vol. 5520. Lecture Notes in Computer Science. Springer, 2008, pp. 252–300. DOI: 10.1007/978-3-642-03153-3_6.

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

[80] Edsko de Vries. "Integrated versus Manual Shrinking". In: (2019). URL: https://www.well-typed.com/blog/2019/05/integrated-shrinking/ (visited on 2021-05-29).

[81] Philip Wadler. "A Orettier Printer". In: *The Fun of Programming*. Macmillan Education UK, 2003, pp. 223–243. DOI: 10.1007/978-1-349-91518-7_11.

[82] Philip Wadler. "Propositions as Types". In: *Commun. ACM* 58.12 (2015), pp. 75–84. DOI: 10.1145/2699407.

[83] Philip Wadler et al. *Prettyprinter: A Modern, Easy to Use, Well-Documented, Extensible Pretty-Printer*. Version 1.7.0. 2020. URL: http://hackage.haskell.org/package/prettyprinter-1.7.0 (visited on 2021-05-29).

[84] Love Waern. *Choice of Functorial State for runNonDet*. 2019. URL: https://github.com/polysemy-research/polysemy/issues/246 (visited on 2021-05-29).

[85] Love Waern. *In-other-words: A Higher-Order Effect System Where the Sky's the Limit*. Version 0.2.0.0. 2021. URL: http://hackage.haskell.org/package/in-other-words-0.2.0.0 (visited on 2021-05-29).

[86] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. "Effect handlers in scope". In: *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4–5, 2014*. Ed. by Wouter Swierstra. ACM, 2014, pp. 1–12. DOI: 10.1145/2633357.2633358.

[87] Nicolas Wu et al. *Fused-effects: A Fast, Flexible, Fused Effect System*. Version 1.1.1.1. 2021. URL: http://hackage.haskell.org/package/fused-effects-1.1.1.1 (visited on 2021-05-29).

# List of Figures

# List of Tables

# List of Listings