University of Copenhagen
Computer Science Department

The Worlds Best Futhark Formatter

Authors: Therese Lyngby & William Henrich Due
Advisor: Troels Henriksen
Submitted: November 3, 2024

Abstract

This report details the extension of a rudimentary formatter for the programming language Futhark. The formatter is described, implemented and its correctness asserted. The formatter is judged to be correct, if it it does not destroy the original syntax tree of a program, formatting a program a second time does not change the output, and if all comments in the program have the same order before and after formatting.

Furthermore, the formatter should retain the original layout of the program, that is if a term spans a single line it should continue to span a single line, and if a term spans multiple lines, it should continue to span multiple lines. Finally after formatting a program comments should be placed close to terms they were next to in the original program.

1 Introduction

This project implements a code formatter for the programming language Futhark [2]. The formatter should be able to take a Futhark program and make it "pretty". However, as prettiness is highly subjective, we shall define some properties the formatter must have in order to produce, in our opinion, nicely formatted programs. Some of these properties can be tested and guarantee the correctness of the formatter. Other properties are used as heuristics for inserting comments in such way as to preserve their context in the original program.

The formatter should be similar to the Haskell formatter Ormolu[1], in the sense that it should be nonconfigurable, and allow the user some control of layout by not forcing some piece of code to be spread over multiple lines or force some code to fit on a single line. Instead, if the piece of code that was written on a single line stays it must stay on a single line, and the same for a piece of code written across multiple lines.

The design of the formatters interface is based on the Haskell library PrettyPrinter [1] and the formatter from the article "A Prettier Printer" by Philip Wadler [3]. The Wadler article was a great inspiration on how to handle multi-line and single-line formats. Another inspiration was looking at how Ormolus monad for formatting is defined to see if something similar to this could be used.

The implementation can be found in the Futhark repository[2] at the path `src/Futhark/Fmt/`. The formatter can be used by using the newest nightly build or by compiling the newest version of the master branch. The command to use the formatter is `$ futhark fmt` and the formatter works correctly in accordance to the CI tests we have made.

2 Theory

To design and implement a formatter: A frame of theory behind the programs and formatters is needed.

2.1 Programs

First a definition of what a program and what a term is in the context of formatting.

[1] https://github.com/tweag/ormolu
[2] https://github.com/diku-dk/futhark

Definition 2.1 (Set of Programs)**.** The set of Futhark programs P is a set of byte sequences that can be parsed and expressed as a syntax tree.

Definition 2.2 (Term)**.** A term t is a sub-tree in the syntax tree resulting from parsing a program p corresponding to some syntactical construct in the Futhark grammar. Every term t has a corresponding span of bytes in p.

Another definition that is needed is the notion of a layout of a term in a program. This is needed to format specific parts of a program in a style that only fits on a single line or spans over multiple lines.

Definition 2.3 (Layout)**.** A term in a syntax tree of a program $p \in P$ has either a *single-line* or *multi-line* layout.

- A term spanning over a single line is *singlel-ine*.
- A term spanning over two or more lines is *multi-line*.

A property of this notion of layout in regards to nested terms in a syntax tree can then be realized.

Proposition 2.1. If t is a term in the syntax tree of a program $p \in P$ then:

- If t is single-line then any sub-term of t is single-line.
- If t is multi-line and is a sub-term of t', then t' is multi-line.

Proof. If t is single-line then any sub-term must also be single-line since you can not fit two or more lines in a single line. And if t is multi-line and is a sub-term of a term t', then t' is multi-line since you can not fit two or more lines into a single line. □

This property is useful because for any given term with a layout then if it is single-line then it only has to play nicely with other single-line layouts in it. While something like multi-line formatting has to account for sing-line and multi-line but it only has to work well inside a multi-line layout. It also helps knowing that nested terms does not change the layout of a current term.

Programs may also contain comments which can not be represented in the syntax tree of a Futhark program.

Definition 2.4 (Comment)**.** A comment c_i is a piece of documentation in a program $p \in P$ which do not effect the syntax tree.

- A comment has a single line number it belongs to $i \in \mathbb{N}$.
- Every comment is placed at the end of a line.

- Every comment has a corresponding span of bytes in p.

Because comments are always a the end of a single-line then they cannot appear inside a single-line. And they will only appear some where before a line. These comments as aforementioned does not exists in the syntax tree and only exists as a sequence that somehow has to be inserted in a formatted document.

These comments also has to be placed correctly in relation to the original code. The definition of location is as follows:

Definition 2.5 (Location Relation)**.** A comment is either located before or after a term t in the syntax tree of a program p:

- c is located before t if the span of bytes corresponding to c comes before the span of bytes corresponding to t.
- c is located after t if the span of bytes corresponding to c comes before the span of bytes corresponding to t.

2.2 Formatter Properties

When asserting the correctness of a formatter then it should always produce correct programs. That is any formatted program should produce the same output for any input as the original program. Furthermore, this formatter should produce syntactical equivalent programs to the original program. Such a definition 2.6 is weaker, allows the formatting of ill-formed programs, and makes the property computable.

Definition 2.6 (Nondestructive)**.** A formatter $f : P \to P$ is said to be nondestructive if the syntax tree of p is equivalent to the syntax tree of $f(p)$ regardless of the layout of each term in respect to the program.

The formatter must not change already formatted code. This definition is partly needed to assert if something has already been formatted. Such that if the original program is equivalent to the formatted program then the program must already be formatted. This property is known as idempotence 2.7 and we wish to find a formatter f such that it is idempotent under function composition.

Definition 2.7 (Idempotence)**.** A formatter $f : P \to P$ is said to be idempotent if for any $p \in P$ it holds that $f(p) = f(f(p)) = (f \circ f)(p)$.

The formatter must also retain the structure of comments. The comments are not included in the syntax so it is not easily defined what correctly placed comments are. At least the formatter should not change the ordering of comments 2.8 given in the original program.

Definition 2.8 (Comment Order)**.** The formatter $f : P \to P$ is said to retain comment order if for any $p \in P$ the sequence of comments $(c_n)_{n \in \mathbf{N}}$ in p is element wise equivalent to the sequence of comments $(c'_n)_{n \in \mathbf{N}}$ in $f(p)$.

As comment explains the surrounding code, they should ideally be inserted into the syntax tree close to the terms, they were placed next to in the original program. We distinguish between two different types of comment placement:

Definition 2.9 (Prepended Comments)**.** A prepended comment c in program p, must be placed on its own line without any code, and typically give context to the code immediately below the comment.

Definition 2.10 (Trailing Comments)**.** A trailing comment c in program p, must be placed at the end of a line, which also contain bytes corresponding to a term t in p, and typically give context to aforementioned code.

Because prepended comments give context to code after the prepended comment, and trailing comments give context to code before the trailing comment in order to optimally retain the context of comments, we also want our formatter to obey the following property:

Definition 2.11 (Type of Comment Placement)**.** A formatter $f : P \to P$ retain Type of Comment Placement, if for every comment c in program p:

- If c is a prepended comment in p, then c is also a prepended comment in $f(p)$
- If c is a trailing comment in p, then c is also a trailing comment in $f(p)$

To simplify the decision of when to add a comment, and to further preserve the context of it, we added the following restriction on the placement of comments:

Definition 2.12 (Restricted Comment Order)**.** A formatter $f : P \to P$ is said to retain Restricted Comment Order, if it retains Comment Order and for any $p \in P$ the following also holds: For every comment c in p and every term t in p

- If c is located after t in p, then c is also located after t in $f(p)$

- If c is located before t in p, then c is also located before t in $f(p)$

A nondestructive formatter allows for the possibility to compare the layout of each term. The formatter must not intrude on the layout choices of the user. And the formatter must be able to use the layouts of terms to format them correctly. So if you have a sequence of terms with certain layouts then the formatter should be able to use this information to construct correctly formatted programs.

Definition 2.13 (Layout Invariant)**.** Given a program $p \in P$, a nondestructive formatter $f : P \to P$, a term t in the syntax tree of p and its corresponding term t' in the syntax tree of $f(p)$. If the layout of any t is equivalent to t' then the layout invariant is fulfilled.

3 Design

In this section we will discuss how our design uses and intents on fulfilling specific properties.

3.1 Formatting

Haskell has a library PrettyPrinter [1] which will be used for formatting the actual code. It can be used to print tree like structures like Futhark programs. This library supports things such as being able to format by specific document width. We will not be using this feature since this formatter should retain the same layout as the user has used. Therefore we would not want to split something that is single-line into multi-line or vice versa.

The library also has helpful functions for aligning, indenting, and nesting text which will be used. We use this library even thought it is overkill because it can efficiently format the text, and it is already part of the Futhark repository.

3.2 Formatting Monad

To format Futhark programs a monad is needed to make the code that does the actual formatting comprehensible. The goal is to make some of the formatting process implicit such that when writing a formatter for a specific term it is easy to read. The goal is the user should not have to handle small details.

- Implicit comment insertion when needed.

- Allow for copying from the original file.
- Adaptive formatting.

This monad is a combination of a reader monad and a state monad which is created using monad transformers from the library `mtl`. The name of this monad is `FmtM` and when its computed value is a `Doc` string from PrettyPrinter then we call the synonym type for `Fmt`. `Fmt` can be seen as a monadic string which can be formatted using PrettyPrinter-like functions. This monadic string should then take care of the aforementioned concerns.

3.2.1 Reader Monad

The environment of the reader monad part only consists of the `Layout` which describes what formatting strategy should be used.

```
data Layout = SingleLine | MultiLine
```

This is part of the environment since the layout can only be changed in future computations. This is due to Proposition 2.1 then the layout of subterms can only change to single-line as the formatter traverses downwards the syntax tree.

3.2.2 State Monad

The state monad should contain:

1. `[Comment]`: A list of comments that is popped from as comments are inserted during formatting.
2. `Maybe Comment`: A pending comment which can be inserted as a trailing comment.
3. `Maybe LastOutput`: The last thing which was printed which is either:
4. `ByteString`: The byte string containing the unformatted program.

```
data Last =  Line | Space | Text
```

The goal with state is to when ever a location of a term is known then it is possible to add comments before the term is printed. Or use the location of the term to determine wether a comment should be added as being a trailing comment to the term. And the last output can then be used to determine if a comment should be printed with a line before it. Lastly the byte string is used to copy from the original code specific substrings. All of

this information should be hidden from the user and should be handled by the exposed primitives. Ormolu also has things like pending comments and a symbol for the last output printed.

3.3 Primitives

We have a bunch of primitive functions for formatting which mirrors how PrettyPrinter works but monadic variances. For instance something like:

```
sep :: [Fmt] → Fmt → Fmt
```

The first argument is concatenated (in the order they are given) with the second argument separating each element.

The moment this gets interesting is in relation to the `Layout` environment modifies some of these primitive functions. There are two special kind of primitive functions which are:

```
line :: Fmt
indent :: Int → Fmt → Fmt
```

In something like the PrettyPrinter `line` would insert a line character and `indent` would indent a expression by some amount of spaces. But since `Fmt` is monadic and contains information about the layout two different strategies can be used for formatting `line` and `indent` depending on what the layout is. We choose here that `line` should become a `space` and `indent` should not indent at all in a `SingleLine` layout. Meaning that in an ideal world you would only have to define your `MultiLine` format and depending on the `Layout` in the environment then the transformation of removing `indent` and turning `line` into spaces should do the rest. This ends up being very natural in some cases, if you consider something like Futharks sum types. A multi-line layout could be defined as followed:

```
sep (line <> "|" <> space) (map fmt tes)
```

Here `<>` is concatenation, `space` is just a space and `map fmt tes` is a list of formatted type constructors in the sum type. If the constructors are `#foo`, `#bar`, and `#foobar`, then in Figure 1 it can be seen how this definition will be formatted in relation to a specific layout.

```
#foo | #bar | #foobar
```

(a) Formatted using single-line Layout.

```
#foo
| #bar
| #foobar
```

(b) Formatted using multi-line Layout.

Figure 1: The formatting of sum types in different layouts.

This case works out great since the single-line layout looks great while only having to think of the multi-line layout. There are more complicated cases where a single-line and multi-line formats are not easily unifiable. Such a case happens when formatting tuple literals in figure 2.

```
(a, b, c)
```

(a) Formatted using Singleline Layout.

```
(a
,b
,c)
```

(b) Formatted using Multiline Layout.

Figure 2: The formatting of tuple literals in different layouts.

If one wanted to naturally just define the multi-line layout from sub-figure 2b then the resulting single-line version would be `(a,b,c)` which is wrong in relation to what we wanted. The solution here is to introduce the binary operation:

```
( <|> ) :: Fmt → Fmt → Fmt
```

Where `s <|> m` means if in a single-line layout choose `s` and if in a multi-line layout choose `m`. This operation is defined with a lower precedence than concatenation `<>`. Now using this, we can then define the tuple like so.

Listing 1: Tuple formats with different separator.

```
"(" <>
    sep ("," <> space <|> line <> ",") (map fmt tes)
<> ")"
```

Since the tuples only differ in their separator then `"," <> space <|> line <> ","` can just be used for the separator. It might also be the case that it is difficult to realize if some format designed for multi-line printing overlaps somehow with a single-line design. In some cases it might be unclear how they differ and where `<|>` should be inserted. Luckily, there is always a trivial solution to have two completely different designs. Since `<|>` always either chooses between single-line and multi-line, then it can be used as the outermost operation to just choose between two different formats depending on layout.

Listing 2: Tuple formats defined with no overlap.

```
("(" <> sep ("," <> space) (map fmt tes) <> ")")
<|>
("(" <> sep (line <> ",") (map fmt tes) <> ")")
```

The idea is this should give the user the ability to get the best of both worlds in regards to if a given multi-line to single-line transformation feels natural or not. We can also realize some algebraic rules for our `<|>` operator in relation to `indent` and `line` which allows for simplifications of `Fmt`.

```
a <|> a = a
space <|> line = line
a <|> indent i a = indent i a
```

There are also distributive laws such that `<|>` can be factored inside sub-expressions.

```
a <> b <|> a <> b' = a <> (b <|> b')
a' <> b <|> a' <> b = (a <|> a') <> b
sep s as <|> sep s' as = sep (s <|> s') as
align a <|> align a' = align (a <|> a')
nest i a <|> nest i a' = nest i (a <|> a')
indent i a <|> indent i a' = indent i (a <|> a')
```

Meaning it would be possible to mechanically go from the tuple format which completely separates the single-line and multi-line formats in Listing 2 to the original format where only the separator was the difference in Listing 1.

Also note that you can distribute over each element in the list `as` in `sep s as`. Since `sep` is just a shorthand form for:

```
a1 <> s <> a2 <> s <> a3 <> ... <> s <> aN
```

For this design to work then the user who is writing the formatter of some term should then use something like **`local`** to update the environment to match the layout of the current term.

3.4 Comments

The formatter uses the Futhark parser to build the syntax tree of a program, which is when turned back into formatted program text. Since comments are not part of the syntax tree, we get these separately as a list with `parseFutharkWithComments`. The list of comments, sorted in order of appearance, is passed as part of the initial state when running the `FmtM` monad. After formatting the syntax tree, the comments need to be inserted into the formatted program. As discussed in sub-section 2.2, the formatter must do this in way, that retain Restricted Comment Order and Type of Comment Placement.

Our initial idea for comment insertion was based on a heuristic for whitespace in parsers: After parsing a token all trailing whitespace must be consumed. This lead us to the heuristic, that every time a term is formatted, all prepended comments must be consumed. We defined the function:

```
addComments :: Located a ⇒ a → Fmt → Fmt
```

which given the located term `a` and its `Fmt`, selects a local layout according to the original layout of `a`, pops all comments located before `a` from the list of comments in the `FmtM` state, formats them, and prepends them to the format of `a`.

3.4.1 Trailing Comments

This simple approach retain Restricted Comment Order, but not Type of Comment Placement, as all comments are treated as prepending. In order to also treat trailing comments, we got inspiration from Ormolu, which defines a similar function, `located`[3]. This lead us to define the function

```
setTrailingComment :: (Located a) ⇒ a → FmtM ()
```

Which checks whether the top comment `c` in the list of comments in the `FmtM` state is located on the same line as `a`. If so, the formatter cannot immediately append `c` behind the format of `a`, as `a` may be a single-line sub-term in a larger term, which continues afterwards on the same line. Instead the formatter stores `c` as a pending comment in the state of `FmtM`, and waits until next time the `hardline` primitive is called.

As `line` becomes an actual line, once it is called, the formatter is guaranteed that no more code may follow on the same line. Therefore it can safely prepend any pending comment to the format of newline. Note that since there may be only one comment on a line, there can never be more than one pending comment.

3.4.2 Prepended Comments

Now that the formatter handles trailing comments, we discover another flaw in our design: Since the formatter also call `addComments`, while formatting sub-terms, the formatter is not guaranteed that prepended comments start on a new line. As a result some comments, which where prepending in p become trailing comments in $f(p)$. This is bad because the formatter now treats trailing comments differently, meaning it may format the program differently, breaking Idempotence, if applied a second time.

Since the formatter must retain Restricted Comment Order, the formatter must add the prepending comments on a line by themselves before formatting the term `a`. However the formatter should also not add unnecessary newlines. Therefore we define the data type

[3] https://hackage.haskell.org/package/ormolu-0.7.7.0/docs/Ormolu-Printer-Combinators.html

```
data LastOutput = Line | Space | Text | Comm
```

which indicates, which primitive was last used to output format. Each time any of the primitive functions are called, `lastOutput` is updated in the `FmtM` state. In the state `lastOutput` is of type `Maybe LastOutput`, where `Nothing` indicates that nothing has been output yet. As part of formatting the prepended comments, the formatter adds a newline, if `lastOutput` is not `Nothing` or `Just Line`.

3.4.3 Drawback of Restricted Comment Order

The Restricted Comment Order property does unfortunately have the drawback, that comments placed between the sub-terms of a term t may break the formatting of t as shown in Figure 3.

```
...
let x = foo (
  bar (
...
```

(a) Before formatting without comment.

```
...
let x =
  foo (bar (
...
```

(b) After formatting without comment.

```
...
let x = foo (
  -- example comment
  bar (
...
```

(c) Before formatting with comment.

```
...
let x =
  foo (
    -- example comment
    bar (
...
```

(d) After formatting with comment.

Figure 3: Formatting broken by comment

We have decided that this is acceptable in the interest of simplifying comment reinsertion while keeping the original context of the comment, since the format still looks decent.

3.5 Layout Invariant

One of the properties we want our formatter to fulfill is the Layout Invariant 2.13. Here something like adaptive primitive functions helps fulfill the layout invariant. It makes it so that lines can not be inserted into a term that should be formatted in a single-line style. It also does not allow for indents since this is not something that is often not wanted:

```
line <|> space = space
indent i a <|> a = a
```

This does not help to guarantee that multi-line layouts always become multi-line. So the layout invariant should be fulfilled by the user who writes the formatter.

As aforementioned this property allows for a design such that the user can format code as single- or multi-line depending on what the user has written. It also has another reason since fulfilling this invariant currently helps doing better formatting. A example of this is if you want to format declarations. Consider if you have a bunch of declaration. Then something you may want to is you would want to separate single-line terms by a single line and multi-line terms by two lines like in Figure 4.

```
def a =
  1

def b = 2

def c = 3
```

(a) Before formatting.

```
def a =
  1

def b = 2
def c = 3
```

(b) After formatting.

Figure 4: Formatting declarations.

Consider if you did not follow the layout invariant and wanted to format `def a = 1` as a single-line instead of a multi-line. Well then when formatting each declaration you would have to inspect wether it actually ends up being single-line or multi-line. Meaning you would have to inspect the structure of `def a = 1` and propagate the result of if the formatted codes layout is multi-line or single-line. We sadly do not have such an mechanic in our formatter. Therefore, we simply inspect what the layout of the original program is and choose the formatting by this. This results in if `def a = 1` is formatted as single-line when it is multi-line then this would break idempotence 2.7. Therefore, not fulfilling the layout invariant will result in the code in figure 5.

```
def a = 1

def b = 2
def c = 3
```

(a) After formatting the code from sub-figure 4b where functions become single-line.

```
def a = 1
def b = 2
def c = 3
```

(b) After formatting sub-figure 5b.

Figure 5: Formatting declarations to single-line can break idempotence.

This problem could probably had been solved by having a `Fmt` type that also contain information about if it was single-line or multi-line by propagating if a line was used in some `Fmt`. But instead we just utilize the layout invariants since that should ideally always be fulfilled.

4 Testing

This section describes how the correctness of the formatter is asserted.

4.1 Methodology

The formatter implementation should fulfill the three properties described in section 2.2: Idempotence, Nondestructive, and Comment Order. Only these properties are tested since they are important for the correctness of the formatter. While the other properties are deemed to not be as important for the correctness or might be hard to test for. These properties are tested using property-based testing. Instead of using a generator to generate Futhark programs, we use the collection of, at the time of writing, 2305 test programs in the Futhark repository to assert, that the formatter fulfills the properties. This gives us a higher degree of confidence in the correctness of our formatter than a handful of unstructured unit tests would. Since we here assert that the properties of the formatter in regards to arbitrary programs is fulfilled with greater confidence.

We believe this testing method is not ideal since the formatting of each term must play nicely with every other term and each term can be single- or multi-line. Meaning we end up with a lot of cases where possibly different combinations of terms and layouts can go wrong. Furthermore, testing for our specified properties does not help us determine the aesthetic quality of our formatter, which we test separately on a small scale. We believe all these tests suffices to establish, that our formatter have our desired properties and

roughly formats according to our desired aesthetic. Therefore we are decently confident in the correctness of our formatter.

4.2 Nondestructive

This test is done by using `futhark hash`, which takes a program and calculate its hash. The hash is calculated using the Futhark pretty printer, which prints the syntax tree as a sting and hashes it.

In the tests we simply compare the computed hash of our program p and the computed hash of the formatted program $f(p)$. The test may falsely pass since two different futhark programs may have the same hash. As this is highly unlikely, we judge this as acceptable. It would had been nice, if the syntax tree generated by the parser also somehow contained comments, similar to how it contains documentation. This test would then also assert that comments are placed correctly in relation to specific terms.

4.2.1 Token Stream

Initially, we sought to test the nondestructive property of the formatter by comparing the token stream produced by `futhark tokens` before and after applying the formatter to a program. However, during testing we discovered, that the Futhark parser removes certain information while building the syntax tree, which results in changes to the tokens produced before and after formatting.

An example is chained let bindings. Here the futhark syntax optionally allow the keyword **`in`** before every chained let binding. The optional **`in`** is not visible in the syntax tree produced by the futhark parser, and as a result our formatter removes all optional uses of **`in`**, leaving only the **`in`** in the innermost let binding as shown in figure 6.

```
let a = 1 in
let b = a * a
in b
```

(a) Before formatting.

```
let a = 1
let b = a * a
in b
```

(b) After formatting.

Figure 6: Formatting removes certain **`in`** tokens.

As the **`in`** tokens are optional, their removal only changes the token stream, not the syntax tree produced by the parser. The change is therefore nondestructive. We switched to comparing program hashes, which are calculated

based on the syntax tree produced by the parser. Thus **in** tokens can be added or removed without causing the test to fail.

4.2.2 Simplifying Literals

The nondestructive test also accounts for cases where literals are simplified, such as when hexadecimal integers are turned into decimal integers. The literals are turned back into their proper format then printed by the pretty printer. It is highly likely, that expressing an integer in two different formats will yield different hashes. As a result the test checks, that our formatter does not simplify literals, but retain the format specified by the user.

4.3 Idempotence

To test that our formatter is idempotent we simply format a Futhark program once $f(p)$ and asserts that it is unchanged if we format it once more $f(p) = f(f(p))$. We do this with our formatter and `cmp(1)`. As we compare the two produced files byte by byte, there is no leeway in whitespace or hidden characters.

4.4 Comment Order

We use `futhark tokens` to extract every token from the original program and the formatted program. Using `grep(1)` we get only the tokens representing comments. Again with `cmp(1)`, we compare the comment tokens before and after formatting the program. This test sadly do not help assert, that comments are placed correctly in regards to specific terms.

4.5 Aesthetic

In order to assert that the formats produced by the formatter actually match our desired aesthetics, we wrote a few short test programs p, stored in the `tests_fmt` directory. In the subdirectory `expected` we wrote equivalent test programs p' following our desired format. We compare the formatted test programs $f(p)$ to their expected output p' to check that $f(p) = p'$. This is not the case for `extra_equals.fut`, which we will discuss later.

Overall this test could benefit from longer test programs following more outlandish formats. Currently there are only a limited number of short test programs, all written in a style close to our preferred format. Therefore the test is not that interesting, as it hardly poses any actual challenge to the formatter. Two more interesting programs are `trailingComments1` and

`trailingComments2`, which check the placement of comments after formatting. The other test programs primarily focus on terms, we discovered had information loss in the parser as described in subsection 4.2.

4.5.1 Redundant Information

In the case of chained let bindings, programs without the optional `in` keywords were neater, and therefore we find programs formatted without them to be the more aesthetic choice. However there are cases, where the formatter, due to information loss, ends up adding redundant information to formatted programs, which we think is undesirable. An example of this is pattern matching on a record.

Futhark record expressions consists of comma separated field expressions, which may take two different forms in the Futhark grammar. One form, `f`, is a simplification of the other, `f = e`, for cases where the expression `e` is a variable defined in scope with the same name as the field `f`[4]. When parsing record patterns both forms are represented with the same type of node in the syntax tree. As a result our formatter always follow the `f = e` form turning record patterns with field expressions of the `f` form into the form `f = f` like in figure 7

```
let {a} = {a = 1}
in a
```

(a) Before formatting.

```
let {a = a} = {a = 1}
in a
```

(b) After formatting.

Figure 7: Formatting adds redundant information

Test programs like `tests_fmt/extra_equals` already more or less follow our desired format, but contain terms with parser information loss. They are meant to assert, that no redundant information is added to the program by the formatter. However, since we believe the best solution to the parser information loss is to extend the parser by for example making the form of the field expressions of a record pattern clear in the syntax tree, our formatter currently adds redundant information to programs like `extra_equals`, and as a result the test fail.

[4] https://futhark.readthedocs.io/en/stable/language-reference.html#f1-f2-fn?

4.6 Result

All of our property-based tests passes, and most of our test programs formats into our desired style. A known exception is pattern matches on records, which we believe is most effectively solved by an update of the Futhark parser. We also have other known issue like sum type constructors not having a location for comment placement and missing tokens in let binding but these are also Futhark related. Furthermore, looking through a subset of the files produced during the property-based testing, we see that their formats also end up looking decent. We wish we had more diverse Futhark programs, which would allow us to cover more edge cases, but with time the Futhark repository will likely get more programs for testing. We also do not know for certain if we cover every edge cases but it would be nice to have a greater confidence.

5 Discussion

In this section we will discuss the formatter in relation to comments and our adaptive function.

5.1 Adaptive Functions

We are quite happy with how the adaptive functions which switch from single-line multi-line. It ended up working quite nicely to write some multi-line design and it turn into a single-line design which looked good. There have been brought up cases where a single-line and multi-line design does not work that great together. The current tuple design (by the advisor of this project) has defined the single-line format:

```
("(" <> sep ",␣" xs <> ")")
<|>
(align $ "(" <+> sep (line <> "," <> space) xs </> ")")
```

Here rewriting this as a combined format becomes a bit much to comprehend.

```
    align $ "(" <> ("" <|> space) <>
        sep (("" <|> line) <> "," <> space) xs
    <> ("" <|> line) <>")"
```

You could define some helper definition to make things easier on the eyes:

```
    a <:/> b = a <> ("" <|> line) <> b
    a <:+> b = a <> ("" <|> space) <> b
    nline = "" <|> line
```

Which gives us the definition:

```
align $ "(" <:+> sep (nline <> "," <> space) xs <:/> ")"
```

This might just also be annoying for the reader that they have to understand a bunch of small helper functions. Another annoyance is there are multiple ways of using `<|>` to create a format design which creates multi-line and single-line designs. The authors of this paper believe this is fine because `<|>` gives you the option to choose how you wanna write the way to format something. We believe that `<|>` should allow to help with redundant single-line and multi-line formats. And we do admit that this design does not always show its best side.

5.2 Comments

We do not believe that comment formatting is that great, a huge problem is the user of the formatter is the one who decides where the comments are placed. Ideally the formatter should place the comments itself and the user should not have to think about it. A problem where the user has to place comments is when something like `sep` is used. A problem that can happen is with the following code:

```
sep ("," <> line) (map fmt as)
```

If `map fmt as` add comments before each term then the formatting could result in something like.

```
a
, -- Comment
b
```

Because a comment to the term `b` meaning it will be added after the separator. The solution to this is having special function called `sepComments` which takes a located list of elements and then adds the comments before the separator. This is not ideal but it seems like this is one of the only cases where we have odd comment placement.

6 Conclusion

To the authors knowledge the formatter works correctly in the sense that it does not produce incorrect code and fulfills the properties given. The code is also deemed "pretty" by the authors in most cases. There are still some formatting problems and we are unsatisfied with with how the comment insertion works. If we had more time we would had wanted to improve the code for inserting comments such that the user had to think less of them.

Overall we see this project as a success but there are still problems which will have to get improved or solved with time.

References

[1] Michael Gersh. *prettyprinter*. https://hackage.haskell.org/package/prettyprinter. Version 1.7.1. 2021.

[2] The Futhark Hackers. *Futhark*. URL: https://github.com/diku-dk/futhark.

[3] Philip Wadler and Joyce Kilmer. "A prettier printer". In: 2002. URL: https://api.semanticscholar.org/CorpusID:63096807.