ARL (A Reversible Programming Language)

Jacob Herbst

January 19, 2021

1 Introduction

Reversible computation dates backs quite some time, but had its first milestone in 1961 when Landauer's principle were proposed**??**. Then in 1973 Bennett**??** formalised a model for reversible turing machines (RTM). Any RTM can only compute injective functions as any input must map to a exactly one unique output, otherwise reversiblility would be ambigious. Despite this constriction compared to a classical turing machine, there has been some significant research on the topic and its usecases. Obviously a lot of the work has been done in relation to heat dissipation of reversible vs irreversible languages, but have also shown usecases in the fields of quantum computing, cryptography and checkpointing of simulations**??**. Already in 1982 the time reversible language Janus, and later formalised by Yokoyama and Glück. Janus might be the most prevelant reversible programming language, however the rise of reversible functional languages have been noticeable, probably because of the restriction or property of injectivity. Multiple functional languages have been formalised over the years. The most significant of these might be RFUN, which was Yokoyama, Axelsen and Glück**??**. RFUN uses a heap manager build on the principle of linearity. Mogensen then presented RCFUN which uses another approach for the heap manager, namely simple sharing using reference counting. Mogensen then later presented RIL, which is an intermediate language, where the heap manager uses maximal sharing. The reasoning for a maximally shared heap manager is to abstract away or eliminate memory management completely, opposed to the linear and reference counting models. For the linear approach one can only use a variable once and must use that variable exactly once. In reference counting, this constriction is eliminated, as we may use a variable multiple times, however reference counting limits the way we can construct patterns. RIL with its maximal sharing system resolves this. The constriction for what languages can use RIL is however limited to pure functional languages as RIL uses cons-hashing to improve searches in the heap. And since RIL is an intermediate language and is tedious and error-prone to write by hand, RIL poses a great choice for a heap manager for a reversible function language. This report concerns itself with an informalisation and implementation of such language.

2 The Functional Reversible Language ARL

The Functional Reversible language ARL (A Reversible Programming-language) is an implementation of the simple suggested language presented by Mogensen. ARL implements the core concepts of the language, meaning ARL in its current state is a simple type-free language, with Pairs as its only construct, in the ML-style family. The

syntax for ARL has undergone a few modifications from the original syntax to make the language more manageble and easier to work with as a programmer, adhering to the same philosophy constituting the heap management in RIL. An example showing the syntax of ARL can be seen in fig **??** which inverts a tree. One thing to note from this example is that flip is invertible meaning it satisfies $a \circ a^{\dagger} \equiv a^{\dagger} \circ a$. That is, it doesnt matter whether we call the function or uncalls it, it will have the same semantics. Thus the semantics of the supplied program is actually the same as the identity of input as we invert the program 2 times. This property however does not have to hold for all functions. They do however, have to satisfy $a \circ a^{\dagger}a = a$. We uses the dagger from category theory, as this carries nicely over in the syntax of ARL, and has the same meaning as the functions being injective. Therefore it is still important for the programmer to know when a function is partially or fully invertible.

```
fun flip (l,r) = let fl = flip l in
                 let fr = flip r in
                 (fr, lr)
    | x     = x

fun main =
    !flip
    flip
```

Program	::=	*Main Function*$^{+}$
Main	::=	fun **main** = *FunctionCall*$^{+}$
FunctionCall	::=	!**fname** | **fname**
Function	::=	fun **fname** *Rules*
Rules	::=	*Pattern* = *Def** *Pattern*
	|	*Rules* | *Rules*
Pattern	::=	**vname**
	|	**constant**
	|	**vname** <> *Pattern*
	|	(*Pattern*::*Pattern*)
	|	(*Pattern*,*Pattern*)
	|	**vname** as (*Pattern*,*Pattern*)
Def	::=	let *Pattern* = **fname** *Pattern* in
	|	let *Pattern* = !**fname** *Pattern* in
	|	let *Pattern* = loop **fname** *Pattern* in
	|	let *Pattern* = !loop **fname** *Pattern* in

Table 1: Syntax of ARL

As shown in 1 and from the example code previously presented, there is some changes to the original grammar presented by Mogensen, which has been carefully selected to make the syntax cleaner and somewhat easier and more relatable to programmers who are not comfortable with reversible programming languages. These are as follows

- **Introduction of a main**
  the introduction of a main functions, is most likely the biggest adjustment. The reasoning behind for a main function stems from the fact that the original presentation of the language provides no interface for IO. Futhermore, ARL is a pure functional language exactly because of RIL's implementation of a maximally shared heap manager is implemented using a con-hashing, resulting in no obvious way to do IO. The main function will hereby serve as an entrypoint and be a pipeline over the function-calls envoked inside main. That is the input from stdin will be the argument for the first function, and since the compilation ensures that inputs and outputs will be placed in the same register or rather variable in the case of RIL, the input for the next function call will be the result of the previous one. This might not be the optimal solution, and might thus change as the need decides, but for now this simple interface will work. Furthermore there is no current way for the program to output anything, this has to be decided in the compilation. One solution could be to provide the compiler with a flag stating whether or not the user wants any form of output.

- **Two ways of constructing a pair**
  It might seems unimportant to have multiple ways to construct pairs, and at first hand it is, as they have the same semantic meaning. However the decision to do so, is to give programmers an easier time. A pure mathematical function can only take a single argument, where this argument might be an argument-vector. This is unsuprisingly also the case for ARL. The difference between ARL and other programming languages is that ARL litterally only takes one argument where most others take arbitrary many, either by currying as in Haskell or as a vector like in C-style languages. However, with the distinction between cons as (::) representing the cons from ML-style languages and cons as (,) representing a dotted pair from LISP. Letting (::) having a higher level of precedence, following `(x::xs,y)` will construct `Pair (Pair (Var ''x'') (Var ''xs'')) (Var ''y'')` . One can hereby interpret it as a dotted pair with `car` being a list with a head `x` and a tail `xs`, and a `cdr` of any construct. This might make it easier for the programmer despite them beeing equivalent. Furthermore it is also allowed to introduce arbitrary many cons operators as this will get folded the same as the example above.

- **More readable let declarations**
  The let declarations have likewise been modifed in the same philosophy as the rest of the modifications, to make it more approachable by using familiar or close to familiar syntax to ML. Thus instead of having function call on the rhs of the assignment and function uncall on the lhs, we consistently delimit calls and uncalls to the rhs, denoting a difference with a prefix `!`, since this is the symbol closest resembling a dagger. the same concept holds for loops.

- **Change of != to <>**
  This is simply a minor syntactical change, changing the denotation of != to <> as the inequality operator. This has been reasoned to having a more relatable ML-style syntax.

3 Semantics

We will go over how ARL is translated into RIL, both using the formal specifications presented by Mogensen **??**, and with the flip function as an example. Firstly, RIL as a different value representatition than ARL. RIL is as mentioned a intermediate language, with a syntax of very simple instructions. It thus uses specific patterns of machine words for different values.

- 0 in RIL is simply null.
- ARL's pairs is in RIL represented as a pointer to a 3-word block memory, where the first word is the reference count, the second and third word is the first and second part of the pair respectively. the RIL pointer is always represented as a multiple of 4. A special instance of this is nilnil (`[[]]`), which simply is a pair of two empty lists, and has the value 4.
- Constants n in ARL will be translated to $2n + 1$ in RIL, since constants in RIL has to be an odd number. This ensure that constants and pairs dont get mixed up.
- The last type of word in RIL is even numbers, whose value is not a multiple of 4. In its current state only one symbol (`[]`) is present, which is represented as 2.

3.1 Functions

Translation Scheme

$F[\![f\ r_1|\cdots|r_n]\!] =$
```
    begin f
    skip
    --> f1
    f'1 <--
    skip
    end f
```
$R[\![r_1]\!]$
$\vdots$
$R[\![r_n]\!]$
```
    fn+1 <--
    assert A != A
    --> f'n+1
```

where $R[\![r_i]\!]$ is the translation of the Rules and f_i and f_i' represent entrypoint and exitpoints respectively. Essentially a function will evalute each rule sequentially, if a rule evalutes correctly it will terminate and if no rules is matched it will assert a false statement, thus exiting with a failure.

Example flip

$F[\![flip\ r_1|\cdots|r_n]\!] =$
```
    begin flip
    skip
    --> flip1
    flip'1 <--
    skip
    end flip
```
$R[\![r_1]\!]$
$\vdots$
$R[\![r_n]\!]$
```
    flipn+1 <--
    assert A != A
    --> flip'n+1
```

As the structure of a function is quite generic we only change f to flip in the example.

3.2 Rules

Translation Scheme

$R[\![p_i = d_i^1 \cdots d_i^m o_i]\!] =$
```
f_i <--
P[[p_i]]A
A != 0 --> f_i + 1
D[[d_i^1]]
⋮
D[[d_i^m]]
f'_{i+1} <-- A != 0;
P[[o_i]]A
--> f'_i
```

A rule will first evaluate $P[\![p_i]\!]A$. If this evaulates correctly, meaning $A = 0$ (we have moved the value of A into p_i), the body of the function will be evaluated. the result o_i will be evaluted under $\overline{P[\![o_i]\!]A}$ as we needs to reconstruct a pattern and $P[\![x]\!]$ corresponds to a deconstruction of a pattern whereas $\overline{P[\![x]\!]}$ corresponds to a construction.

Example flip

$R[\![(l, r) = fl_{d_1^1}\ fr_{d_1^2}\ (fr, fl)]\!] =$
```
flip_1 <--
P[[(l, r)]]A
A != 0 --> flip_2
D[[fl]]
D[[fr]]
flip'_2 <-- A != 0;
P[[(fr, fl)]]A
--> flip'_1
```
$R[\![x = x]\!] =$
```
flip_2 <--
P[[x]]A
A != 0 --> flip_3;
flip'_3 <-- A != 0;
P[[x]]A
--> flip'_2
```

Rules, like Functions, are quite generic in nature, so only small changes have to be made.

3.3 Patterns

Translation Scheme (variables)

$P[\![x]\!]v =$
```
x <-> v
```

where x is a variable. This is the most basic rule and will be valid whenever x first occurs, if x however is not the first occurence we will need to use the copy subroutine.

$P[\![x]\!]v =$
```
v != x --> l_1;
v <-> copyQ;
x <-> copyP;
uncall copy;
l_1 <-- v != 0;
```

What we first need to do is to check whether or not x and v is identical. This is a prerequisite for the copy subroutine to work. we then switch the values into the variables that is used in the routine. we switch v into copyQ as this is the value that will be consumed. x will be switched into copyP as this is the value that will be saved.

Example flip

$P[\![x]\!]A =$
```
x <-> A
```

This evaluation happens, a few times in the translation of flip. One of these is when we want to destruct our input of the second rule. Essentially we bind the input A into x and clears A.
every variable only occurs once in the flip example.

Translation Scheme (pairs)

$P[\![(p_1, p_2)]\!]v =$

```
v & 3 --> l1;
v <-> consP;
uncall cons;
P[[p1]]consA;
consA != 0 --> l2;
P[[p2]]consD;
consD == 0 --> l3;
l2 <-- consA != 0;
P[[p1]]consA^-1;
call cons;
v <-> consP;
l3 <-- v == 0;
l1 <-- v & 3;
```

When translating a pattern to RIL, we first start by checking whether or not v is a pointer to a pair, since we can safely skip unfolding the pattern if it is not. we will then move the v into consP as this is the variable used by the cons sub-routine. The results of these are stored in consA and consD. We therefore recursively check if p_1 can be correctly unfolded, meaning the value of consA should be 0. If this is not the case we jump to l_2 reconstructing the p_1. Afterwards p_2 is checked in the same way and is the result of consD 0, then we have safely deconstructed the pattern.

Example flip

$P[\![(p_1, p_2)]\!]A =$

```
A & 3 --> l1;
A <-> consP;
uncall cons;
l <-> consA;
consA != 0 --> l2;
r <-> consD;
consD == 0 --> l3;
l2 <-- consA != 0;
l <-> consA;
call cons;
v <-> consP;
l3 <-- v == 0;
l1 <-- v & 3;
```

This is used both as a deconstruction of pair in the argument of the first rule of flip and its inverse in constructing the result (fr, fl) in the same rule. Every case of translation of a subterm i.e. $P[\![p_1]\!]consA$ translates into the simple case `l <-> consA`.
Every variable only occurs once in the flip example.

3.3.1 stuff that is not included in flip

Translation Scheme (constant)

$P[\![k]\!]v =$

```
v != k --> l1;
v -= k;
l1 <-- v != 0;
```

Constants is quite simple. firstly the constant needs be equivalent to v for the pattern to match. Is this the case, we subtract k from v, essentially setting v = 0.

Example flip

Translation Scheme (not equal)

$P[\![x \neq p]\!]v =$
```
    assert x == 0;
    P[[p]]v
    x += v;
    ~P[[p]]v
    v -= x
```

When evaluating a not equal pattern, we first needs to assume x is 0 otherwise our two updates, first to x then to v, would compromise the integrity of v. For instance in the case of flip the rule `| x = x` could be written as `| x <> (l,r) = x <> (fr,fl)`. In such a case p would not be a pointer (v !& 3), thus we skip the entire evaluation of p. we would then subtract, v from x, do nothing once again, and then subtract a value larger than v from v, which is nonsensical. Therefore x must be 0 before the evaluation.

Translation Scheme (As pattern)

$P[\![xas(p_1, p_2)]\!]v =$
```
    v & 3 --> l1;
    x <-> fieldsP;
    call fields
    P[[p1]]fieldsA;
    consA != 0 --> l2;
    P[[p2]]fieldsD;
    consD == 0 --> l3;
    l2 <-- consA != 0;
    ~P[[p1]]consA;
    uncall fields;
    x <-> fieldsP;
    l3 <-- v == 0;
    l1 <-- v & 3;
```

An as pattern is almost identical from the pairs, the only difference is that we want to keep the integrity of x, which is done by using the fields sub-routine.

3.3.2 Let definitions

3.3.3 TODO describe the let defs

4 Parsing

The compiler for ARL has been written in haskell using Megaparsec as parsing library. This was chosen over lexer/parser tools such as Alex/Happy, because of familiarity and because ARL as a language is quite small, thus making it pretty easy to implement. Megaparsec was chosen over other parsing libraries such as Parsec for 2 main reasons. First ARL is an indentation sensitive language, chosen to have quite strict rules, which

we will se later on. Second Megaparsec makes position handling extremely easy giving exact position of when parsing failed with out having to bundle the AST with positions.

4.1 AST

The implementation of the abstract syntax tree, is almost true to the Grammar presented in 1. There are however three minor changes. Instead rules looking like

```
data Rules = P Pattern [Def] Pattern
           | R Rules Rules
```

it simply will be a sum type of only the single constructor P, and then the Func sumtype will take a list of rules, as such:

```
data Func = Func ID [Rules]

data Rules = P Pattern [Def] Pattern
```

This changes is mainly because it is easier to parse and evaluate. the meaning should not change. For the same reason we introduce another pattern namely a NilNil, essentially this is a constant value of 4 or even more explicit a pair of nils (2's). NilNil as a legal value in RIL however depends on the build procedure that will create it. This distinction just makes it easier. Lastly, we earlier described the usefulness of having two ways of creating pairs, in the AST, we however only have one constructor for these as we can use some build-in functionality of megaparsec to enforce precedence without rewriting our grammar.

4.2 Parsers

the basics. Comments is based of f#. line-comments is the same as in // and block-comments is (* *). identifers can be any string starting with a lower character followed by any alphanumerical character, a dash or an underscore.

4.2.1 Functions

As described ARL has been chosen to have some strict rules for indentation. This is forced to make the code readable. We must thus enforce the specific rules in the parser. Firstly we ensure that a function is always declared in column 0. This makes for a fine structure, but might need to be changed in the future if we allow for nested function declarations. we will then consume the unecessary garbage. We will then either have a main function er a pure function, if we encouter a main we will then parse the function calls. Here we enforce another indentation rule, a function call, must reside directly under the fname of main, like solution in the flip code-example. In the parsing we do not enforce a single main function, instead we handle this in the pre-processing. If we however encounter a non-main function (from here just function) we will parse its rules. Like in the main function we ensure that a rule (other than the first, which must be on the same line) resides under under the fname. that is the guard | must be placed here. Other than this indentation handling the parser is simply a sequence of parsers and combinators.

```
funP :: Parser (Either Main Func)
funP = L.nonIndented scn $ L.lineFold scn p
```

```
  where
    p sc'    = do rword "fun"
                  ind <- L.indentLevel
                  id <- identifier
                  case id of
                    "main" -> Left <$> mainP
                    _      -> Right <$> rest id ind
    rest id ind  = do r <- ruleP ind;
                      rs <- many $ try rules
                      mapM_ (\(_,x) -> when (x /= mkPos 5)
                          (L.incorrectIndent EQ (mkPos 5) x)) rs
                      return $ Func id $ r:map fst rs
    rules  = do scn; ind <- L.indentLevel; symbol "|"; r <- ruleP ind; return (r,ind)
    mainP  = do symbol "="; some $ try funC
```

4.2.2 Rules

The parsing for the Rule sum type is actually quite simple as most of the of the indetation is handled in the function parser. Although the rule parser also will have to do some indentation enforcement, it will pass on its indentation level for the let-declarations parser, to make certain that let definitions is deeper indented than the rule, along with forcing let declarations to be lined up with the resulting pattern. Again this is simply used to establish structure for the body of a particular function pattern, also called rule.

4.2.3 Let declarations

Unsurprsingly the let declaration follow a similar structure as the the other parsers. overall we can reduce a let declaration to either of two, it is a function call/uncall or it is a loop. These are very similar in structure so we will only go over the simple case for function calls. again we ensure the indentation is correct, throwing a parse error otherwise. we then uses the same strategy as we did for function calls in main to distinguise between a call and an uncall using the observing function. depending on whether the symbol ! is present before the function identifier, we get a `Left` or a `Right` value which we then converts to the appropriate type.

```
defP :: Pos -> Parser Def
defP ind = try call <|> try loop <?> "Let def"
  where
      call   = do L.indentGuard scn EQ ind;
                  rword "let"
                  lhs <- patternP
                  symbol "="
                  uncall <- observing $ symbol "!"
                  fname <- identifier
                  rhs <- patternP
                  rword "in"
                  scn
                  case uncall of
                    Left _ -> return $ Call lhs fname rhs
                    Right _ -> return $ Uncall lhs fname rhs
```

This function has a lot of duplicate code, which potentially could be eliminated.

4.2.4 Patterns

Patterns is the most atomic part of the grammar, as its only non-terminal symbol is that of Pattern. It is thus also the easiest to parse. We construct a parser for each terminal and combine these using the parser combinators. We can see that whenever we encounter a `[[]]` we have a `NilNil` constructor. for constants we simply wrap the constant value in the Const constructor, we however ommit to changing the value to its internal representation in RIL which would be 2n+1. The reason for this is that we want to distingus between the sytactical and sematical meaning of the program. It is further noticeable that we also wrap `nil` as a Const with a value of 1. A variable is simply the indentifier wrapped. A not equal pattern is likewise simply the identifier and a recursive pattern call. The same holds for the `as` constructor, however the second part of an `as` can only be a pair. For pairs we can see it makes use of the MakeExprParser which specifies associativity and precedence for the two ways of constructing pairs. Lastly we also want to allow to wrap any `Pattern` in parenthesis.

```
patternP :: Parser Pattern
patternP = try as <|> try neq <|> try nilnil <|> var <|> const' <|> try pair <|> parLE <?>
  where
    nilnil = rword "[[]]" >> return NilNil
    const' = (integer <|> nils) <&> Const
    nils   = rword "[]" >> return 1
    var    = identifier <&> Var
    neq    = do ident <- identifier; rword "<>"; Neq ident <$> patternP
    as     = do ident <- identifier; rword "as"; As ident <$> pair --patternP
    pair   = parens pairP
    pairP  = makeExprParser patternP
             [
                [InfixR $ Pair <$ symbol "::"],
                [InfixR $ Pair <$ symbol ","]
             ]
    parLE  = parens patternP
```

5 Evaluation

5.0.1 TODO describe how the compiler is implemented.

6 Results

6.0.1 TODO describe tests

6.0.2 TODO describe how well the project has come along

7 How to use - code structure

7.0.1 TODO describe the code structure and how to run the program.

8 Conclusion

8.0.1 TODO

9 References

9.0.1 TODO fill, mostly with references in the introduction.