ARL (A Reversible Programming Language)

Jacob Herbst

January 25, 2021

1 Introduction

Reversible computation dates back quite some time but had its first milestone in 1961 when Landauer's principle was proposed[1]. Then in 1973 Bennett[2] formalized a model for Reversible Turing Machines (RTM). Any RTM can only compute injective functions as any input must map to exactly one unique output. Otherwise, reversibility would be ambiguous. Despite this constriction, compared to a classical Turing Machine, there has been some significant research on the topic and its use cases. A lot of the work has been done concerning heat dissipation of reversible vs irreversible languages. But reversibility of programs have also shown uses in the fields of quantum computing, cryptography, and checkpointing of simulations[3]. Already in 1982 the time-reversible language Janus was presented, and later formalized by Yokoyama and Glück[4]. Janus might be the most prevalent reversible programming language, however, the rise of reversible functional languages have been noticeable, probably because of the injectivity restriction. Multiple functional languages have been formalized over the years. The most significant of these might be RFUN, presented Yokoyama, Axelsen, and Glück[5]. RFUN uses a heap manager built on the principle of linearity. Mogensen then presented RCFUN[6] 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[7]. The reasoning for a maximally shared heap manager is to abstract away or eliminate memory management completely from the high-level language, 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 purely functional languages as RIL uses cons-hashing to improve lookups 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 a formal description and implementation of such language.

2 The Functional Reversible Language ARL

The Functional Reversible language ARL (A Reversible Language) is an implementation of the simple language presented by Mogensen[7]. 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 manageable 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 figure **??** which inverts a tree. One thing to note from this example is that flip is invertible meaning it satisfies $a \circ a^{\dagger} \equiv a^{\dagger} \circ a$ where a is a function. That is, it doesn't matter whether we call the function or uncalls it, it will have the same semantics. Thus the semantics of the supplied program is equivalent to the identity of input as we in the main function calls flip 2, thus invert the tree 2 times. This property however does not have to hold for all functions. They do, however, have to satisfy $a \circ a^{\dagger} \circ a = a$. We use the dagger from category theory, as this carries nicely over in the syntax of ARL, and has the same mathematical meaning as the functions being injective in the context. We programming one therefore might have to keep track of 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
```

Figure 1: function to flip a tree in ARL

As shown in Figure2 and from the `flip` function in Figure**??**, there are some changes to the original grammar presented by Mogensen. These changes have 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 function, is most likely the biggest adjustment. The need for a Main function stems from the fact that the original presentation of the language provides no interface for IO. Since the heap manager in RIL is maximally shared and uses a con-hashing algorithm, ARL cannot allow any updating of variables and no way of doing side-effects, there is no obvious way to do IO. The main function will contextually serve as an entry point and be a pipeline over the function-calls invoked 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 ARL evolves and the needs change. 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.

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

Figure 2: Syntax of ARL

- **Two ways of constructing a pair**
  It might seem unimportant to have multiple ways to construct pairs, and at first hand, it is, as they have the same semantical 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 unsurprisingly also the case for ARL. The difference between ARL and other programming languages is that ARL only takes a single argument where most other popular languages take arbitrary many, either by currying as in Haskell or vector-like in C-style languages. However, these are pure abstractions. And in the same manner, we can abstract away any single argument restriction in ARL. With the distinction between cons as (::) representing the cons from ML-style languages and cons as (,) representing a dotted pair from LISP. One can interpret any (::) as a list with a head and tail and (,) as vector/tuple abstractly giving a C-like parameter list. Letting (::) having a higher level of precedence than (,), following `(x::xs,y)` will construct

  `Pair (Pair (Var ‘‘x’’) (Var ‘‘xs’’)) (Var ‘‘y’’)` .

  One can see this as a dotted pair with `car` being a list with a head `x` and a tail `xs`, and a `cdr` of any construct `y`. This abstraction might make it easier for the programmer despite them being 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 modified 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 Parsing

The compiler for ARL has been written in Haskell using Megaparsec as the 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 see later on. Second Megaparsec makes position handling extremely easy giving the exact position of when parsing failed without having to bundle the AST with positions.

3.1 AST

The implementation of the abstract syntax tree is almost true to the Grammar presented in figure2. There are however three minor changes. Instead of rules looking like

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

it simply will be a product type of the constructor Rule, and then the Func sum type will take a list of rules, as such:

```
data Func = Func ID [Rules]

data Rule = Rule { args :: Pattern,
                   body :: [Def],
                   output :: Pattern }
```

This change is mainly reasoned by being 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, however, NilNil as a legal value in RIL depends on the build procedure that will create it. We, therefore, want it to have its own constructor as this simply makes implementation 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.

3.2 Parsers

3.2.1 Basics.

Comments are based on f#. line-comments is the same as in // and block-comments is (* *). Identifiers can be any string starting with a lower character followed by any alphanumerical character, a dash, or an underscore.[1]

3.2.2 Functions

As described ARL has been chosen to have some strict indentation rules. 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 unnecessary garbage. A function will then either be a Main function or a pure function, if we encounter a main we will then parse the function calls. Here we enforce another indentation rule. A function call, must reside directly under the function name of main (which is "main"), like in the example code in Figure**??**. 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 the function name. that is the guard | must be placed here. Other than this indentation handling, the parser is simply a sequence of parsers and combinators.

3.2.3 Rules

The parsing for the Rule sum type is in itself quite simple as most of the indentation 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 a structure for the body of a particular function pattern, also called a rule.

3.2.4 Let declarations

Unsurprisingly the let declaration follows a similar structure as 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 use the same strategy as we did for function calls in main to distinguish 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 convert to the appropriate type. This function has a lot of duplicate code, as the loop/unloop construct is very similar. This could potentially be eliminated.

3.2.5 Patterns

Patterns are 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, once again this constructor is useful as the construct of nilnil in RIL depends on a subprocedure that construct a list with an empty list inside it, which then will have the varaible name `nilnil`. for integer

[1] any code described the following subsections can be found in appendix ?? or in the file Parser.hs

constants we simply wrap the constant value in the Const constructor, we, however, omit to change the value to its internal representation in RIL which would be 2n+1. The reason for this is that we want to distinguish between the syntactical and semantical meaning of the program. It is further noticeable that we also wrap `nil` as a Const with a value of 2. A variable is simply the identifier wrapped in our `Var` constructor. 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 in figure**??** 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.

4 Semantics

4.1 RIL

Before we explain the semantics of ARL, we will shortly go over RILs syntax and semantics[7]. At its core, RIL is a set of blocks consisting of an Entry, a Body, and an Exit. Entries have one of the following forms[2]:

- Unconditional jump: `-> l`
- Conditional jump: `c -> l`
- SubRoutine exit: `end l`

These should be fairly similar to the reader, as these works very similarly as regular jumps. the unconditional jump will always jump to label l, the conditional jump will jump to label if the condition c is true, if not it will proceed to execute the next operation. the `end` label will simply denote the ending of a subroutine. Exits is just opposite of entries, meaning RIL has the following types of entries:

- Unconditional entry: `l <-`
- Conditional entry: `l <- c`
- SubRoutine entry: `begin l`

The unconditional works as a classic label and is the duality to the unconditional jump. called in reverse the unconditional entry works as an uncoditional jump and vice versa. Likewise the conditional entry is the inverse of a conditional jump. A subroutine entry is simply the beginning of a subroutine. We jump to this label whenever a subroutine is called. In this context it is worth noting, that since RIL has such a basic structure it is a parameterless language, meaning subroutines will use specific variables for their computation, which requires a program to have some extra conditional exit and entry points. any conditional has the form $L \bowtie R$, where L is a variable or memory location and R is either an L or an integer constant in the range -2^{31} to $2^{31} - 1$. the $\bowtie$ operator can be any of the following operators in the first column of table1 where each operator carries the same semantics as they do in C, except for `!&` which is a binary NAND operator. The body of a block consists of statements or subroutine calls. As RIL is reversible, the statements in a body are quite limited and can be expressed on the form $L_1 \oplus = R_1 \odot R_2$, or L_1 <-> L_2. where L, again is a variable or a memory location

[2] we describe the shorthand syntax as this is the one generated by ARL

and R is either an L or an integer constant in the range -1^{31}-2^{31}-1. And once again with the operators having same semantics as in C. There is further some restrictions to which L value R can take, that ensures reversibility. We wont go over these as these are not really relevant for the semantics of ARL. When we in the later sections describe the semantics of ARL, we willactually not use any of the ⊙ operators as these is actually not needed to compile ARL to RIL. The swap function <-> likewise swaps the value of the two L's, ensuring no loss of information. Table{invop} shows how each RIL instruction inverts, which will be useful as a reference for evaluating patterns in reverse.

$\bowtie$	$\oplus$	$\odot$
==	^=	+
<	+ =	−
>	− =	^
! =		&
<=		$\vert$
>=		<
&		>
!&		

Table 1: RIL operators

l <–	inverts to	–> l
l <– c;	inverts to	c –> l;
begin l	inverts to	end l
L += R;	inverts to	L -= R;
L_1 <-> L_2;	inverts to	L_1 <-> L_2;
call l;	inverts to	uncall l;
$s_1 \cdots s_n$	inverts to	$\overline{s_n} \cdots \overline{s_1}$

Table 2: Inversion of RIL operations

4.1.1 Value representation

RIL furthermore has a different value representation than ARL. RIL is as mentioned an intermediate language, with a syntax of very simple instructions. It thus uses specific patterns of machine words for different values.

- 0 represents the absence of a value.
- ARL's pairs are 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. An instance of this is nilnil (`[[]]`), which simply is a pair of two empty lists, represented by 2, and is constructed by an initialize procedure looking as such:

```
begin initialise
consA += 2;
consD += 2;
call cons;
nilnil <-> consP
end initialise
```

- Integer constants n in ARL will be translated to $2n + 1$ in RIL since the heap-manager uses odd numbers to represent integers. This ensures that constants and pairs don't 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 ([]/nil) is present, which is represented as the value 2.

4.1.2 Subroutines

RIL also has 3 subroutines, which are used by the heap manager to manage the reference counts of nodes along with ensuring maximal sharing. These are used for some of the more complicated patterns in ARL. copy - is used to copy values, which is what allows us to use the same variable multiple times. fields - is used in the “as” pattern. cons - is used for pairs.

4.1.3 Copy

the copy subroutine uses the variables copyP and copyQ. Copy assumes copyP to be bigger than 0 and copyQ to be equal to 0. This makes sense, since 0 is the absence of a value, and thus copyP cannot be 0 as there would be no value to copy and it must be a positive integer as it is an index in memory. CopyQ likewise needs to be 0 as would not be a true copy of copyP if it weren’t. If copyP is a pointer the reference count is increased and copyQ is set to the same value as copyP using `copyQ += copyP`. Called in Reverse copy assumes the two variables to be equivalent, as this is the only way to “destroy” a variable without loss of information. This happens by subtracting copyP from copyQ. thus copyQ will be 0. again if copyP is a pointer the reference counter is decreased.

4.1.4 Fields

Fields have 3 variables, fieldsA, fieldsD, and fieldsP. We have previously described how an as pattern is an identifier and a pair. the identifier will be the pointer to this pair and will be located in the fieldsP variable. The other two variables must be 0, to ensure correctness. It will then set the fieldsA and fieldsD to the second and third word of the pair respectively, which corresponds to the car and cdr of the pair. In reverse, the fieldsA and fieldsD will be cleared.

4.1.5 Cons

The cons subroutine is quite a bit more complex than the other two. This is because it also has to allocate and deallocate nodes and it is implemented using hashing to make lookup more efficient. We will not go over the specifics, but only the general functionality. Cons take two arguments consA and consD, which must be values (not 0). These values will be cleared, or possibly more intuitive they will be placed as second and third word of the pair, if the pair doesn’t already exist on the heap, otherwise the reference count will be increased, while consA and consD are cleared. The pointer to the pair (consA, consD) will be in consP. Called in reverse a pair is deconstructed, deallocating the pointer if the reference count reaches 0 and increasing the reference count for the consA and consD fields.

4.2 Functions and Rules

Mogensen have developed a schematic to compile pattern-matching for functional programming languages such as ARL[8]. The semantic of ARL, which will be covered in section4.2-4.4. We however will present these using valid RIL syntax. Figure3 show how we evaluate functions and rules, where $R[\![r_i]\!]$ is the translation of the Rules and f_i and f'_i represent entry points and exit points respectively. Essentially a function will have its entry point, and jump immediately to the entry of the first rule. It will

$F[\![f\ r_1|\cdots|r_n]\!] =$

```
begin f
skip
--> f_1
f'_1 <--
skip
end f
```

$R[\![r_1]\!]$

$\vdots$

$R[\![r_n]\!]$

```
f_{n+1} <--
assert A != A
--> f'_{n+1}
```

$R[\![p_i = d_i^1 \cdots d_i^n o_i]\!] =$

$\mathtt{f}_i$ `<--`

$P[\![p_i]\!]A$;

`A != 0 -->` $\mathtt{f}_{i+1}$;

$D[\![d_i^1]\!]$;

$\vdots$

$D[\![d_i^n]\!]$;

$\mathtt{f}'_{i+1}$ `<-- A != 0;`

$P[\![o_i]\!]A$;

`-->` $\mathtt{f}'_i$

Figure 3: Semantics of functions and rules

evaluate each rule sequentially until one is evaluated correctly, that is, its exit point has been reached it will then terminate the function/subroutine (this is a simplification). If no rules are matched the function will assert a false statement, thus exiting with a failure, which essentially means a function cannot be called on any construct only those matching the rules.

Rules are introduced by their entry point f_i. From here p_i, which is the parameter pattern of the rule will be evaluated. Essentially what we do when we evaluate a pattern in the forward direction we try to move it out of A, which is the variable chosen for input and output as RIL as stated is parameterless. If A is correctly distributed to p_i, the value of A will be 0 and we can thus ignore the conditional jump. and proceed to evaluate the body of the rule. Is A however not equal to 0, it means that the pattern was not correctly matched and thus we want to make the jump, which leads us to the next rule. If the jump is not taken the body can safely be evaluated. We can then see there is an exit point for f'_{i+1}. The reason for this is we have to evaluate the result of each previous rule to make sure the output is disjoint, meaning the function is injective. This concept is visualised in figure4. It is also worth noting that when evaluating the result o_i, we evaluate it inversely. This can be seen as a construction of A based on o_i, whereas the $P[\![p_i]\!]A$ could be the deconstruction of A into p_i. Lastly, we will take an unconditional jump to right before the result in the previous rule, to do the disjoint checking as described.

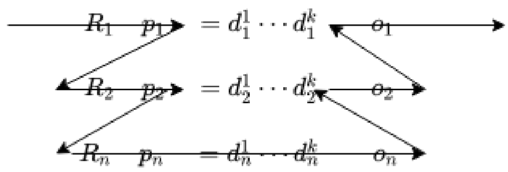

Figure 4: Flowgraph rule evaluations

4.3 Patterns

4.3.1 Variables

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

when x is first occurence

$P[\![x]\!]v =$
```
v != x --> l1;
v <-> copyQ;
x <-> copyP;
uncall copy;
x <-> copyP;
l1 <-- v != 0;
```

$\overline{P[\![x]\!]v} =$
```
x <-> v;
```

when x is first occurence

$\overline{P[\![x]\!]v} =$
```
v != 0 --> l1;
x <-> copyP;
call copy;
x <-> copyP;
v <-> copyQ;
l1 <-- v != x;
```

Figure 5: Semantics of variables

There are two different ways a variable needs to be compiled. The most basic rule `x <-> v`, with x being the variable, will be valid whenever x first occurs in a pattern. Called in reverse this is simply the same instruction. For every occurrence of x that is not the first occurrence, we will need to use the copy subroutine, described earlier. When evaluating a variable that has occurred previously we first need to check whether or not x and v are identical. This is a prerequisite for the copy subroutine to work as it results in an assertion failure in the copy subroutine otherwise. we then switch the values into the variables that are 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. When evaluated in reverse, we check that v is 0 as this again would result in an assertion failure, we move x into copyP and makes a copy into copyQ and move it back to x and v.

4.3.2 Constants

$P[\![k]\!]v =$
```
v != k --> l1;
v -= k;
l1 <-- v != 0;
```

$\overline{P[\![k]\!]v} =$
```
v != 0 --> l1;
v += k;
l1 <-- v != x;
```

Figure 6: Semantics of constants

Constants are quite simple. firstly the constant need to be equivalent to v for the pattern to match. Once again this we want to extract the constant k from v, getting v to equal 0 if the pattern matches. This can only be the case when they are equivalent. In the case they are, we simply subtract k from v, and since k is a constant and will never change we cannot and there is no need to do anything to k. In reverse we do the opposite we check if v is 0 if it is we can set it to the value of k.

4.3.3 Pairs

$$
\begin{array}{l}
P[\![(p_1, p_2)]\!]v = \\
\quad \texttt{v \& 3 -{}-> l}_1\texttt{;} \\
\quad \texttt{v <-> consP;} \\
\quad \texttt{uncall cons;} \\
\quad t_1\ \texttt{<-> consA;} \\
\quad t_2\ \texttt{<-> consD;} \\
\quad P[\![p_1]\!]t_1\texttt{;} \\
\quad \texttt{t}_1\ \texttt{!= 0 -{}-> l}_2\texttt{;} \\
\quad P[\![p_2]\!]t_2\texttt{;} \\
\quad \texttt{t}_2\ \texttt{== 0 -{}-> l}_3\texttt{;} \\
\quad \texttt{l}_2\ \texttt{<-{}- t}_1\ \texttt{!= 0;} \\
\quad \overline{P[\![p_1]\!]t_1}\texttt{;} \\
\quad t_1\ \texttt{<-> consA;} \\
\quad t_2\ \texttt{<-> consD;} \\
\quad \texttt{call cons;} \\
\quad \texttt{v <-> consP;} \\
\quad \texttt{l}_3\ \texttt{<-{}- v == 0;} \\
\quad \texttt{l}_1\ \texttt{<-{}- v \& 3;}
\end{array}
$$

$$
\begin{array}{l}
\overline{P[\![(p_1, p_2)]\!]v} = \\
\quad \texttt{v \& 3 -{}-> l}_1\texttt{;} \\
\quad \texttt{v == 0 -> l}_3\texttt{;} \\
\quad \texttt{v <-> consP;} \\
\quad \texttt{uncall cons;} \\
\quad t_1\ \texttt{<-> consA;} \\
\quad t_2\ \texttt{<-> consD;} \\
\quad P[\![p_1]\!]t_1\texttt{;} \\
\quad \texttt{t}_1\ \texttt{!= 0 -{}-> l}_2\texttt{;} \\
\quad \texttt{l}_3\ \texttt{<-{}- t}_2\ \texttt{== 0;} \\
\quad \overline{P[\![p_2]\!]t_2}\texttt{;} \\
\quad \texttt{l}_2\ \texttt{<-{}- t}_1\ \texttt{!= 0;} \\
\quad \overline{P[\![p_1]\!]t_1}\texttt{;} \\
\quad t_1\ \texttt{<-> consA;} \\
\quad t_2\ \texttt{<-> consD;} \\
\quad \texttt{call cons;} \\
\quad \texttt{v <-> consP;} \\
\quad \texttt{l}_1\ \texttt{<-{}- v \& 3;}
\end{array}
$$

Figure 7: Semantics of pairs

When translating a pair to RIL, we first start by checking whether or not v is a pointer to a pair. This can be done by checking `v & 3`, as pointers always will have 11 in their 2 least significant bits. If v simply is not a pair, we can skip the entire unfolding of v, jumping straight to the bottom. is v however a pair, we move v into consP, as we need to deconstruct by uncalling cons. the car and cdr will then be in consA and consD. we however have to move these to two newly created variables t_1 and t_2. this might seems unnecessary at first but whenever we have nested patterns, not moving consA and consD out to new variables will make the program fail as these will not be 0 in the uncall to cons in the nested pair. when moved accordingly, we can then evaluate p_1 under t_1. After this evaluation we need to check if `v` was correctly cleared. If t_1 is 0 we can move on to evaluate p_2 under t_2. Is it the case that t_1 is not 0 we jump to entry l_3 and reconstruct v_1. Once again this is to ensure we don't lose any information while evaluating a pattern we will then proceed to reconstruct v by doing the inverse sequence of operations as when we deconstructed the pair. Do we on the other hand evaluate p_2 correctly we can jump to entry l_4. When evaluated inversely we start by checking whether v is a pointer, skipping the entire thing if it isn't. we then check wether v is 0. if it is we jump to entry l_4, and proceed to construct v by evaluating t_2 and t_1 inversely, calling cons and moving into v. If v is not 0 we have to deconstruct it even further, by uncalling cons and make evaluate t_1. Overall the procedure will deconstruct a pair in forward direction and create a pair in the inverse direction.

4.3.4 As pattern

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

$$
\begin{array}{l}
P[\![x\,as(p_1,p_2)]\!]v = \\
\quad \texttt{v \& 3 --> } \texttt{l}_1\texttt{;} \\
\quad \texttt{v <-> fieldsP;} \\
\quad \texttt{call fields;} \\
\quad \texttt{x <-> fieldsP;} \\
\quad t_1 \texttt{ <-> fieldsA;} \\
\quad t_2 \texttt{ <-> fieldsD;} \\
\quad P[\![p_1]\!]t_1\texttt{;} \\
\quad \texttt{t}_1 \texttt{ != 0 --> l}_2\texttt{;} \\
\quad P[\![p_2]\!]t_2\texttt{;} \\
\quad \texttt{t}_2 \texttt{ == 0 --> l}_3\texttt{;} \\
\quad \texttt{l}_2 \texttt{ <-- t}_1 \texttt{ != 0;} \\
\quad \overline{P[\![p_1]\!]consA}\texttt{;} \\
\quad \texttt{x <-> fieldsP;} \\
\quad t_1 \texttt{ <-> fieldsA;} \\
\quad t_2 \texttt{ <-> fieldsD;} \\
\quad \texttt{uncall fields;} \\
\quad \texttt{v <-> fieldsP;} \\
\quad \texttt{l}_3 \texttt{ <-- v == 0;} \\
\quad \texttt{l}_1 \texttt{ <-- v \& 3;}
\end{array}
\qquad
\begin{array}{l}
\overline{P[\![x\,as(p_1,p_2)]\!]v} = \\
\quad \texttt{v \& 3 --> } \texttt{l}_1\texttt{;} \\
\quad \texttt{v == 0 -> l}_3\texttt{;} \\
\quad \texttt{v <-> fieldsP;} \\
\quad \texttt{call fields;} \\
\quad \texttt{x <-> fieldsP;} \\
\quad t_1 \texttt{ <-> fieldsA;} \\
\quad t_2 \texttt{ <-> fieldsD;} \\
\quad P[\![p_1]\!]t_1\texttt{;} \\
\quad \texttt{t}_1 \texttt{ != 0 --> l}_2\texttt{;} \\
\quad \texttt{l}_3 \texttt{ <-- t}_2 \texttt{ == 0;} \\
\quad \overline{P[\![p_2]\!]t_2}\texttt{;} \\
\quad \texttt{l}_2 \texttt{ <-- t}_1 \texttt{ != 0;} \\
\quad \overline{P[\![p_1]\!]t_1}\texttt{;} \\
\quad \texttt{x <-> fieldsP;} \\
\quad t_1 \texttt{ <-> fieldsA;} \\
\quad t_2 \texttt{ <-> fieldsD;} \\
\quad \texttt{uncall fields;} \\
\quad \texttt{v <-> fieldsP;} \\
\quad \texttt{l}_1 \texttt{ <-- v \& 3;}
\end{array}
$$

Figure 8: Semantics of As pattern

pair, we check if v is in fact a pair. we will then move v into fieldsP, calling `fields` and then distributing the pointer to x, fieldsA to t_1 and fieldD to t2. Again t_1 and t_2 needs to be unique newly created variables, such that we encounter any trouble with nested patterns. The rests of the evaluation of an `as` pattern is the same as for pairs, since the only difference between an `as` pattern and a `pair` pattern is that we in the `as` pattern want to keep a reference to the pair. In reverse the same principles also holds.

4.3.5 Not equal (<>)

$$
\begin{array}{l}
P[\![x \neq p]\!]v = \\
\quad \texttt{assert x == 0;} \\
\quad P[\![p]\!]v\texttt{;} \\
\quad \texttt{x += v;} \\
\quad \overline{P[\![p]\!]v}\texttt{;} \\
\quad \texttt{v -= x;}
\end{array}
\qquad
\begin{array}{l}
\overline{P[\![x \neq p]\!]v} = \\
\quad \texttt{v += x;} \\
\quad P[\![p]\!]v\texttt{;} \\
\quad \texttt{x -= v;} \\
\quad \overline{P[\![p]\!]v}\texttt{;} \\
\quad \texttt{assert x == 0;}
\end{array}
$$

Figure 9: Semantics of <> pattern

For a `not equal` pattern, we first need 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 v 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. As explained, after the assertion we want to deconstruct p under v, then update x with `x += v`, setting x to v. here v should have its original value as it should skip moving v into p, else x would be equal to p. we then reconstruct p under v and subtract the value of x from v. In its core this is a simple swap, however, if p matches v, v should be 0 and no update to x is happening.

4.4 Let definitions

$$\begin{array}{l} D[\![\texttt{let}\,p_1 = \texttt{call}\,f\,p_2\,\texttt{in}]\!] = \\ \quad \overline{P[\![p_2]\!]A}\texttt{;} \\ \quad \texttt{call f;} \\ \quad P[\![p_2]\!]A\texttt{;} \\ \quad \texttt{assert A == 0;} \end{array}$$

$$\begin{array}{l} D[\![\texttt{let}\,p_1 = \texttt{loop}\,f\,p_2\,\texttt{in}]\!] = \\ \quad \texttt{l}_1 \texttt{ <-- A != 0;} \\ \quad \overline{P[\![p_2]\!]A}\texttt{;} \\ \quad P[\![p_1]\!]A\texttt{;} \\ \quad \texttt{A == 0 --> l}_2\texttt{;} \\ \quad \texttt{uncall f;} \\ \quad \texttt{--> l}_1 \\ \quad \texttt{l}_2 \texttt{ <--} \\ \quad \texttt{assert x == 0;} \end{array}$$

Figure 10: Semantics of let definitions

4.4.1 function calls

A call consists of 4 parts. First, we want to evaluate p_2 under A in inverse. We want to construct A from p_2. this should prepare A to be the input for `f`. the call to `f` then happens, and the result is always placed in A. we then evaluate p_1 under A, moving the value from A into p_1. lastly, we need to assert that A is 0. this assertion is important, as it ensures us that the result of `f` is in fact a matching pattern to p_1. For instance, if `f` returns 7, we cannot assign 7 to a pair, thus such a construct should fail.

4.4.2 Loops

Loops are useful in situations where tail-recursive functions are needed. but since these are not allowed we can write these as our loop construct. The loop will keep calling `f` until p_1 is matched. we first have an entry l_1. This is where the loop starts. we then construct A from p_2. Then right after we deconstruct A into p_1. if A is 0 it means p_1 was matched correctly and we do not call the function `f` as we jump to entry l_2. if A is not 0 p_1 is not matched and we call the function `f`. We then jump back to l_1, repeating the procedure until p_1 is matched.* Evaluation

5 Compiler implementation

The implementation of the evaluation functions for ARL is built on a stack of monad-transformers. The reason for choosing such a solution is that monads are a well-integrated part of Haskell and it makes it a lot easier to implement the recursive calls to the different functions as we can use do notation to lift our functions into the monad.

Furthermore, we both have an environment we want to pass on to the different `eval`-functions and some states to make it a lot easier to ensure that entries and exits are unique and that variables are correct etc. And probably most importantly, the stack allows for easy extensibility as we can easily add new monad transformers to our stack. The stack looks as follows:

```
type Eval a = ReaderT Env (StateT RilState Identity) a

runEval env st ev = runIdentity
```

Figure 11: The Monad stack for evaluation

As can be seen from Figure11, the stack is fairly simple. The eval type takes an arbitrary type a, we only use `String` as this allows us to write the RIL code directly to a file. our string is then wrapped in an identity monad, this in itself is useless, but works well with other monads. This again is better for extensibility as, we can always substitute for another monad such as IO, which cannot be stacked as a transformer. The identity monad is then wrapped in a state transformer, where the state itself is of the type `RilState`, which is a product type we will go over later in this section. And lastly, we wrap readerT around the State. In the future, it could be useful to add the error monad to the stack to handle failures, which we currently don't do, or the writer monad to add some kind of logging.

5.1 Why reader?

The reader monad is extremely useful in our case as we have an environment we want to pass around to the different function, and it makes it easier to manage if this is not passed around as parameters but is kept isolated in the environment which can then be locally set to the specific function calls. From section~4 it might be clear that we often use `A` as the variable, we evaluate under, however in some cases this change, for instance when evaluating a pair where we need to evaluate t_1 and t_2. Therefore we might want to keep track of this. This at first seems like a state but since it never changes inside of any function we can define it in the environment. The second part of the environment is a map. We use this to keep track of which variables are alive in the program. These should be stored on the stack before a function call. this is fairly simple to do since the control flow of ARL is extremely simple. One solution might be to search the AST from the bottom up, however since the control-flow is as simple as it is, we extract all variable IDs from a Rule into a list of ID lists. we then check if a variable in a list is in any of the following lists. If this is the case the variable must be alive. we can then zip these results with the unique identifier for a let declaration, constructing our map. Thus the Environment looks as follows:

```
type Env = (String, M.Map String [ID])

baseEnv = ("A", M.empty)
```

5.2 RilState

As mentioned, there is some state in RIL that we want to keep track of to make everything easier to grasp. The RilState can be seen in figure12, where one can notice that there is quite a lot of fields for the product type. Firstly there is RuleNo, this simply is a counter on rules, which `rLabel` is simply the string version of `ruleNo`, so we don't have to call `show` whenever we need the rule number. This might be a bit excessive. fnameS will be set at the beginning of the `evalFun`, and is used together with the unique identifiers for patterns and let declarations to ensure that label names do not occur multiple times. We can exploit this since we know, any function name needs to be unique and every rule needs to be unique. `LabNo` and `label` are the same duality as ruleNo and `rLabel` and will number jumps and entries inside the rules. Once again to enforce no duplication of labels. `pVars` is the last field of the state. `pVars` is used to check if a variable has previously occurred in a pattern. Now that we have already gotten over how we use the reader monad, the reader might seem like a good solution for this. It would be if it weren't for how pairs are evaluated. As described in section4 we need to rebuild t_1 if is not correctly matched, which is opposes some problems. Therefore an easier solution is to add a variable to pVars when it is first encountered, otherwise generating duplicate code, and then resetting this map back to empty right before we check $\overline{P[\![p_1]\!]t_1]\!]}$.

```
data RilState = RilState { ruleNo :: Int
                         , rLabel :: String
                         , fnameS :: ID
                         , labNo :: Int
                         , label :: String
                         , pVars :: M.Map ID Int
                         }
```

Figure 12: The state of the RIL program

5.3 Generating RIL code

Just like in the parser, we have an `eval` function for each non-terminal in the AST. We use the do notation to generate the state etc. we need for a specific function, and then we want to wrap the string inside the monad. we construct the strings, by creating a list of strings, where each string is a RIL instruction, which then gets intercalated, with newlines to preserve structure in the RIL file. To make the code easier to read we abstract away the operations. functions with names v(EQ|NEQ)(0|x)(E|J), will be conditional jumps and entries, where v is equal or not to 0 or x. Plus and sub is the updates (+=) (-=) respectively. Swap is (<->). Furthermore, we have defined swap functions for each of the variables used in the 3 subroutines described in section4.1 as these are used quite often, e.g. `consP x` swaps x with consP.

6 Results

*When it comes to the actual ARL compiler it is still in its early stages. First and foremost they are no optimizations implemented. One such optimization could be dead code removal, which would make the actual RIL file less cluttered. Furthermore, there is very little error-handling implemented in the ARL compiler itself. As described in section~**??** MegaParsec does fine error handling on its own and we let any syntax error be handled by the library. We then check that functions are not defined multiple times, but this is where the error-checking stops. The reason for this is the compiler does not do a whole lot of static checks. However, the need for these checks is also very limited, when keeping in mind there are no types that need to be unified, type-checked, etc. One thing that is not ideal however is that any errors that might occur will be RIL-errors, and thus can be hard to identify right away. There are further downsides to the compiler in its current state. First and foremost there is no adequate way to provide any input for the program other than lists. It means one needs to modify the RIL file to construct the desired structure in A, which is highly undesirable. We provide some example programs, that are used to test the evaluation, to show how ARL can be used.

- **flip**: This is the function used in the example. This program is an example of a program that would need some modification in the RIL as it is designed to work on a tree and not a list, despite it working on lists too.
- **flipN**: This function is a duplicate of flip but with the explicit disjoint patterns meaning in the `| x = x` case we specify that x cannot be equal to (l,r) as such ~| x<>(l,r) = x<>(fr,fl). Thus this works as a test for the neq pattern.
- **dup**: Dup is a function that duplicates a structure. This program shows that we correctly can use a variable multiple times (in a simple manner). Called in reverse it will test that the "forward" direction of multiple occurrences works as well.
- **head**: This creates a pair of the head and a list using the as pattern. This program is interesting compared to the other functions we have looked at. The reason for this is, that head is an obviously non-invertible function in its classical sense, we must hereby pair it with some garbage that can identify the original input. This shows some of the major limits with ARL and reversible computing in general. However many "regular" functions can be expressed in ARL if we include some garbage.
- **rev**: this shows that our loop construct works, reverse is a fully invertible function meaning we can use call and uncall interchangeably. We also can do this inside the loop of rev.

It should be clear that these programs show some of the interesting properties of reversible computing. One big advantage is that many functions are injective and for partially invertible functions we can limit ourselves to only define a function once, that can then do multiple things. An example of such would be zip and unzip functions. Clearly, these advantages hold for all reversible languages, and despite ARL's relatively low expressiveness, it has some clear advantages because of the backend. The maximal sharing of the heap-manager makes ARL a bit more manageable compared to RFUN etc. where one has to call a duplicate function every time a variable is used multiple times. On the other hand, ARL is still very limited in its structure. It can be a bit difficult to make any larger programs, since there is no obvious way to store results of function calls, and function calls are structured as a pipeline, which means functions in the same program must fit together. To be able to store variables would be highly valuable. Another place ARL could be improved is in the unexisting type-system. It would be very nice to be able to do arithmetic on integers etc. Lastly, since ARL is a functional language it would make obvious sense to allow higher-order functions, which potentially also could circumvent some of the limitations. Overall ARL show some great promise, and the syntax makes it quite easy to understand compared to other languages but for now, it is limited to do some fairly simple programs.** TODO describe tests

7 How to use - code structure

In its current state, the compiler is still a bit tedious to use, since no good interface have been implemented. The ARL compiler will simply generate a RIL file, which has to be compiled by the RIL compiler, which then in turn needs to be compiled using a C compiler. The Haskell project is build using stack and is thus required (The build details is defined in the packages.yaml file and might thus also work with cabal). The workflow of compiling and running an ARL file looks as follows:

stack build stack exec "ARL-exe filename [options]" here the filename is the .arl file to compile. Notice opposite of the RIL compiler it also needs the .arl suffix. The compiler is able to take some arguments, which looks as follows -i –input: this is the list to provide for the build subroutine in the RIL program. This has to be an integer list. The default value is the list 1..10. -o –output: this option specifies the name of the produced .ril file. The default option is to have the same name as the .arl file. -c –compile: This flag is a little redundant but will specify which file to compile, if the file is not provided. When the RIL code has been generated, the program can be compiled to C code using the comRIL binary. We provide this in the project folder. Now one can compile the c file. the generated file can now be generated. If the output shows 4 lines, which shows some statistics of the program, the program terminated correctly. To make this project easier we provide a Makefile to compile and run the example programs. Simply run make to compile all the examples, or make `filename` to build and run a specific program.

7.1 Code structure

the project consists of the following test-folder: This folder test the parsing both that it works correctly and that it fails correctly. It also tests some of the other functions used in the code. These tests could have been a lot more extensive. Lastly, the test-folder has the examples-folder. This is where all the examples files are stored. In this folder, the comRIL compiler is also provided. This file is not original work of the author but can be credited to Mogensen. src-folder: This folder consists of the source-code to the ARL compiler. And includes the following files: Main: Here we do the IO, read a file call the parser and the evaluation, and writes to a file. Ast: This defines the Abstract syntax tree. Parser: This file includes the parser. Eval: This file defines the evaluation and the monad we do the evaluation under. RilState: Defines the State used in the evaluation of the AST. RilEnv: Defines the Environment to read from in the evaluation. RIL: specifies the general setup for the heap-manager. such as the heap size, subroutines described in section**??** etc. Util: Holdsfunctions to make the eval-functions more manageable. Options: implements the compiler options. fs

8 Conclusion

8.0.1 TODO

References

[1] R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5(3):183–191, 1961.

[2] C. H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17(6):525–532, 1973.

[3] Markus Schordan, Tomas Oppelstrup, Michael Kirkedal Thomsen, and Robert Glück. *Reversible Languages and Incremental State Saving in Optimistic Parallel Discrete Event Simulation*, pages 187–207. Springer International Publishing, Cham, 2020.

[4] Tetsuo Yokoyama and Robert Glück. A reversible programming language and its invertible self-interpreter. In *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM '07, page 144153, New York, NY, USA, 2007. Association for Computing Machinery.

[5] Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. Towards a reversible functional language. In Alexis Vos and Robert Wille, editors, *Reversible Computation*, volume 7165 of *Lecture Notes in Computer Science*, pages 14–29. Springer Berlin Heidelberg, 2012.

[6] Torben Ægidius Mogensen. Reference counting for reversible languages. In *Reversible Computation*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014.

[7] Torben Ægidius Mogensen. Reversible garbage collection for reversible functional languages. *New Generation Computing*, 36:203–232, 2018.

[8] Torben Ægidius Mogensen. Compiling pattern-matching for reversible functional languages. *Unpublished*, 2018.


```
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

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

patternP :: Parser Pattern
patternP = try as <|> try neq <|> try nilnil <|> var <|> const'
           <|> try pair <|> parLE <?> "Pattern"
  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
    pair   = parens pairP
    pairP  = makeExprParser patternP
             [
                [InfixR $ Pair <$ symbol "::"],
                [InfixR $ Pair <$ symbol ","]
             ]
    parLE  = parens patternP
```