Compiling Janus to RSSA

Martin Kutrib[2], Uwe Meyer[1(✉)], Niklas Deworetzki[1], and Marc Schuster[1]

[1] Technische Hochschule Mittelhessen, Wiesenstr. 14, 35390 Giessen, Germany
uwe.meyer@mni.thm.de
[2] Institut für Informatik, Universität Giessen, Arndtstr. 2, 35392 Giessen, Germany
kutrib@informatik.uni-giessen.de

Abstract. Reversible programming languages have been a focus of research for more than the last decade mostly due to the work of Glück, Yokoyama, Mogensen, and many others. In this paper, we report about our recent activities to compile code written in the reversible language Janus to reversible static-single-assignment form RSSA and to three-address-code, both of which can thereafter be compiled to C. In particular, this is – to our knowledge – the first compiler from Janus to RSSA. In addition, we have implemented a novel technique for a reversible compiler by executing the code generator itself in reverse. Our compiler provides the basis for optimizations and further analysis of reversible programs.

Keywords: Reverse computing · Reversible programming languages · Janus · Reversible static-single-assignment

1 Introduction

Reverse computing, although the initial ideas can be traced back to the 1960s [10], has been a major research area over the last decade. With the growing importance of sustainability and reduced energy consumption, reverse computing promises contributions by avoiding the waste of energy through deletion of information [5].

More than twenty years after the first creation of a reversible language called Janus [11], the papers of the Copenhagen group [17] brought new life into the area of reversible languages by formally defining and extending Janus. Interpretation and partial evaluation [12] as well self-interpretation [18] were studied and in [4] Axelsen published his results on the compilation of Janus. In [19] a reversible flowchart language as a foundation for imperative languages is described and its r-Turing-completeness, i.e. their ability to compute exactly all injective computable functions, is proved.

Whilst it was now possible to execute programs forwards and backwards, there seem to be no results about the optimization of Janus programs. Optimization in this regard refers to improving the execution time of programs or their memory consumption [14].

It is well known from the discipline of compiler construction that optimization can most effectively be performed on some intermediate representation of

© Springer Nature Switzerland AG 2021
S. Yamashita and T. Yokoyama (Eds.): RC 2021, LNCS 12805, pp. 64–78, 2021.
https://doi.org/10.1007/978-3-030-79837-6_4

the source rather than the source code itself or its abstract syntax tree. Such intermediate representations include three-address-code [1] and static-single-assignment [16].

In 2015, Mogensen published his work on RSSA, which is a special form of static-single-assignment that can be executed forwards and backwards [13]. We are going to describe the major concepts of RSSA in Sect. 2.2. However, we are not aware of any work to connect the dots between Janus and RSSA.

In this paper, we report on a new Janus compiler, called *rc3* (*reversible computing compiler collection*) with multiple backends including RSSA. The compiler is available at https://git.thm.de/thm-rc3/release. and has been written with two intentions:

- Provide the ability to execute Janus programs forwards and backwards.
- Establish the basis for further research on the optimization of reversible languages.

Based on the Janus definition in [17], we developed a compiler front-end focusing on the semantic analysis required for a reversible language. The compiler allows for pluggable back-ends with currently three of them in place (see Fig. 1 for an overview):

1. Interpreter: Instead of generating code, the interpreter back-end directly allows the execution of Janus programs forwards or backwards.
2. Three-Address Code: This backend will generate intermediate code in form of non-reversible three-address code that can be translated to a "forwards" as well as a "backwards" C-program including all necessary declarations and function definitions to be able to compile the code on any platform.
3. RSSA: This backend will firstly generate RSSA-code, and at the same time also construct building blocks and a program graph (in order to be able to use these for further analysis of control and data flow). To be able to verify the RSSA code, the RSSA code can be translated into C-code, as well. In addition, we have created a virtual machine for RSSA that allows direct execution of RSSA code.

In summary, this paper describes two important new aspects for the compilation of reversible languages:

- Whilst optimization techniques such as dead code elimination, common subexpression elimination, and many more are very well understood for "traditional" languages, no results are known for reversible languages. These optimizations are typically implemented on some low-level intermediate code. Our compiler is – to our knowledge – the first compiler from Janus to reversible static-single-assignment intermediate code and work to implement optimizations is already underway.
- In addition, we have implemented a novel technique to let the code generator itself operate "in reverse". For instance, the Janus language provides stack primitives such as `push` and `pop`. We have implemented only the former, whereas the code for `pop` is automatically created by inverting the code for `push`.

We will start by briefly explaining Janus as well as RSSA in order to provide a basis for the description of our compiler in Sect. 3. This chapter forms the main part of the paper and contains detailed descriptions and examples of our code generation schemes. Lastly, we will provide insights into the "reverse code generator", followed by an outlook on current achievements on optimization of RSSA.

2 Preliminaries – Janus and RSSA

2.1 Janus

The procedural and reversible programming language *Janus* has originally been proposed in 1982 [11]. Since then, Yokoyama and Glück have formalized the language [18], and together with Axelsen, have considerably extended Janus and shown that it is Turing-complete [17]. We specifically use the extended version as defined in [17].

In this version, a Janus program consists of a `main`-procedure followed by a sequence of additional procedure declarations. Procedures represent a parameterized list of statements, which are the smallest reversible building blocks of a Janus program. Statements can change the state or control flow of the execution by inspecting and manipulating variables. All variables are defined locally, either as an integer, an array of integers, or a stack of integers. While the size of an array must be known at compile-time, stacks can grow arbitrarily at runtime.

A statement is either a reversible assignment, one of the reversible control flow operators (conditional, loop), a procedure invocation, one of the stack operations (push, pop), a local variable block, or an empty skip statement. Each of those statements has a well-defined inverse, which is used during backwards execution. To change the execution direction, procedure invocations are used. In addition to the destination of a procedure invocation, the call direction must be specified using either the `call` or `uncall` keyword. If the `uncall` keyword is used, the invoked procedure is executed in reverse execution direction. Since the keywords `call` and `uncall` interchange their meanings in reverse execution direction, it is possible to restore the original execution direction.

For a program to be reversible, all information must be preserved during runtime. To ensure that no information is lost when exiting the scope of a variable, all local variables are restricted to the body of a local variable block. In these blocks, variables are allocated and deallocated in a structured way that preserves information. For the same reason, parameters for procedures are passed by reference so that the information stored in the parameters is preserved when the procedure ends.

2.2 The Reversible Intermediate Code RSSA

Reversible static-single-assignment (RSSA) was firstly introduced by Mogensen [13] as a reversible variant of SSA (Static-Single-Assignment), which in turn is an intermediate language to facilitate data-flow analysis and was proposed by Alpern, Wegman, and Zadek in 1988 [2]. SSA forms an intermediate representation in which each variable has only one definition and new "versions" of the variables are used for each assignment to it. This is often accomplished by inserting Φ-functions to merge potentially different versions occurring due to branches in the control flow.

RSSA uses variables and constants defined as atoms. A memory location can be accessed in the form $M[a]$ where $M[a]$ represents the location to which an atom a points. Atoms and memory accesses can be used in conditions. If a variable is used on both sides of an assignment, a new version of the variable, a so-called fresh variable, must be created on the right side of the expression so that it can not be defined and used simultaneously [13].

In RSSA, a program is a set of basic blocks, each consisting of a sequence of assignments or a call. Each block is enclosed by an entry and an exit point. Entry and exit points use labels, which must be utilized at exactly one entry and exit point. Valid RSSA programs need to entail one entry point *begin main* and a corresponding exit point *end main* [13]. Entry and exit points may occur in a conditional or unconditional form. Conditional entry points are used in the following manner, $L_1(x,...)L_2 \leftarrow C$. Depending on whether it is entered through a jump to L_1 or L_2, the condition C will be evaluated to either true or false. Conditional exit points are used in a similar manner using the form $C \rightarrow L_1(y,...)L_2$. The condition C is evaluated and depending on the evaluation, a jump to either L_1 or L_2 is performed [13]. These conditional entry and exit points are an alternative means to implement the Φ-functions in RSSA [13], and are used because $\Phi - functions$ use two inputs to compute one output, and would thus prohibit reversibility. RSSA defines a set of reversible instructions, that can be used to compose reversible programs. The most important ones – as shown in Table 1 – are assignments, call, and uncall instructions, as well as entry and exit points. Operands for these instructions are separated into *atoms* or *memory locations*.

Table 1. Important RSSA instructions

Assignment:	x := y ⊕ (a ⊙ b)
Call:	(x$_1$, ...) := call p (y$_1$, ...)
Uncall:	(x$_1$, ...) := uncall p (y$_1$, ...)
Uncond. Entry:	L(x$_1$, ...) <-
Cond. Entry:	L1(x$_1$, ...)L2 <- c
Uncond. Exit:	-> L(x$_1$, ...)
Cond. Exit:	c -> L1(y$_1$, ...)L2

Similar to statements in the programming language Janus, every instruction has a well-defined inverse. Therefore a whole program can be inverted by inverting every single instruction in it and the order in which those instructions appear in. Since RSSA is reversible, every subroutine should be runnable in a forwards and backwards manner. Running a subroutine backwards is performed by calling the inverted form of the subroutine [13].

3 Our Compiler

This section briefly explains the structure and implementation of our compiler's front-end and back-ends. Our approach to implement the compiler in Java largely resembles the approach for a classical multi-pass compiler [3]. The backends are pluggable, such that adding new backends is easy. The next version of the compiler includes an additional layer to optimize the RSSA code.

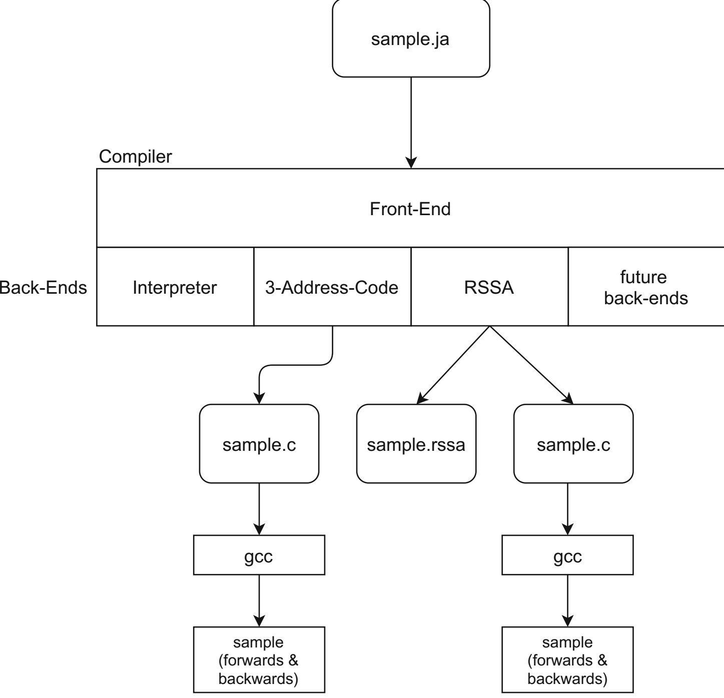

Fig. 1. Overview of our compiler

3.1 Compiler Front-End

The front-end consists of a dedicated scanner and parser, which are generated using the scanner generator *JFlex* [9] and the parser generator *CUP* [7] respectively. The scanner performs the lexical analysis, converting the input characters

into a sequence of tokens. These tokens are then passed to the parser, which performs the syntactic analysis and constructs an abstract syntax tree. After the construction of an abstract syntax tree, it is passed to the semantic analysis.

With the implementation of the semantic analysis, the particularities arising from the properties of a reversible language become clear: As in a conventional language, Janus defines rules for the visibility of identifiers and restrictions on types of variables and expressions. In addition to classical analysis passes, that enter declarations into a symbol table and check the visibility and types of used variables and expressions, another pass has to be defined, the aim of which is to check the reversibility of individual instructions.

Even though Janus specifically defines reversible variants of classical statements, it is possible to construct a statement that cannot be reversed. In total, there are four different variants of assignments, that may not be reversible at runtime, but can be recognized at compile time. In [17] these are mentioned as a "syntactic rule". We have chosen to split them into four cases which are checked during the semantic analysis.

$$x \quad \odot= \quad x + k \tag{1}$$
$$M[e_1] \quad \odot= \quad M[e_2] + k \tag{2}$$
$$x \quad \odot= \quad M[x] + k \tag{3}$$
$$M[M[e_1] + k] \quad \odot= \quad e_2 \tag{4}$$

Fig. 2. Four variants of non-reversible assignments

Figure 2 shows these different variants, where x is an integer variable, M is an array variable, and e_i as well as k are expressions. Variants (1) and (2) are not reversible, since both sides of the assignment can point to the same memory location. In this case, the result of the assignment is the result of modifying a value with itself, which potentially leads to a constant result, causing information to be destroyed. Variants (3) and (4) are not reversible, since the assignment may modify the index, that is used to access an array as part of the assignment. After the index has been modified, it is no longer possible to deterministically identify the values needed to reverse the assignment.

Since we need to compare the identifiers of variables to identify the variants above, it is crucial that two identifiers cannot refer to the same memory location at runtime to guarantee reversibility of assignments at runtime. This so-called *aliasing* can occur when the same variable is passed more than once as an argument to a procedure. Because parameters are passed by reference, the names of the parameters would then refer to the same memory location. This check is part of the semantic analysis, too.

3.2 Compiler Back-End

In this chapter, we are going to explain the translation for the most important Janus elements to RSSA, followed by some remarks on the translation from RSSA to C. To our knowledge, this is the first compiler not only handling reversible languages but also itself using reverse code.

Since Janus' semantics requires the reversal of code for some language features – for instance, to destroy all local variables with the inverse function used for the construction of these – we have chosen to build our compiler in a way that allows for the code generator to also emit inverse code in reverse order. Thus, for the language constructs requiring reversal, we have only implemented the forwards translation and let the code generator itself create the inverse code. A good example is the translation of the Janus' stack operations *push* and *pop*, where we only have implemented *push* and let the compiler work backwards when a *pop* operation occurs. In [18] a similar idea is described, but for an interpreter for Janus written in Janus itself.

More details will be provided below after the explanation of the basics of code generation.

3.3 Assignments and Expressions

Janus assignments to simple variables are of the form $V \oplus = E$, where $\oplus$ is a reversible operator. We will come to assignments to arrays in Subsect. 3.7. Mogensen defines assignments in RSSA in a similar fashion and allows $\oplus =$ to be either $+=$, $-=$, $\hat{}=$. Hence, such assignments are reversible: $+=$ and $-=$ are the inverse of each other, and $\hat{}=$ is inverse to itself.

While the translation of simple expressions such as constants and simple variables is straightforward, composite expressions with multiple operators require a split into multiple assignments to temporary variables as shown in Fig. 3. This is due to the fact that temporary variables will need to be destroyed again with a finalizer (see the fourth line in the example) to avoid producing garbage. As explained in the section about RSSA, we need to ensure that we use "fresh" versions of the variables to ensure that there is always at most one assignment to a variable. Hence, we append version numbers to the names of the variables – see the example depicted in Fig. 3 with the Janus code on the left- and the RSSA code on the right-hand side.

Janus	RSSA
`procedure main()`	`begin main(n0, m0)`
`  int n`	`  T0 := 0 ^ (2 * 3)`
`  int m`	`  n1 := n0 + (1 + T0)`
	`  0 := T0 ^ (2 * 3)`
`  n += 1+2*3`	`  m1 := m0 + (n1 + 1)`
`  m += n+1`	`end main(n1, m1)`

Fig. 3. Example for translation of simple expressions

Note: While RSSA uses only a limited set of arithmetic operators, we have chosen to add the remaining Janus operators, too.

Janus also provides primitives to manipulate stacks and their translation will be shown in Subsect. 3.7.

3.4 If-Then-Else

The only means in RSSA for expressing conditions is via entry points and exit points: An exit point $C \to L_1(Y, \ldots)L_2$ firstly evaluates the condition C, and if true, jumps to L_1, or to L_2 otherwise. Similarly, a conditional entry point $L_1(Y, \ldots)L_2 \leftarrow C$ consists of two labels L_1 and L_2. Via the Ys parameters can be passed from an exit to an entry point. As previously explained, Φ-functions can thus easily be implemented in RSSA.

The parameters in the entry point will always have a higher version number than the corresponding parameters in the exit points to ensure proper SSA form.

Hence, for a Janus if-then-else statement *if C then S_1 else S_2 fi E*, we first evaluate the Boolean expression C and use it in a conditional exit point to label L_1 (if the condition was evaluated to true) or L_2 otherwise.

We then create the label $L_1(\ldots)$ as an unconditional entry point, followed by the translation of S_1 and a jump to L_3. Now comes the else-part starting with label L_2 and ending in a jump to L_4. Lastly, we join both branches with the conditional entry point $L_3(\ldots)L_4 \leftarrow E$ – see Fig. 4.

```
if (n=0)
  then n+=1
  else n-=1
fi (n=1)
```

```
n0 == 0 -> L1(n0)L2
L1(n1) <-
n2 := n1 + (1 ^ 0)
-> L3(n2)
L2(n3) <-
n4 := n3 - (1 ^ 0)
-> L4(n4)
L3(n5)L4 <- n5 == 1
```

Fig. 4. Example for translation of if-then-else statements

3.5 Loops

The translation of a loop *from C do S_1 loop S_2 until E* into RSSA basically follows the same rules as for the conditional statement. Firstly, we will unconditionally jump to L_1, which is used to mark the entry point of the loop. The condition is evaluated and used in a conditional entry point $L_1(\ldots)L_3$ followed by the translation of the body S_1. Now, according to the semantics of Janus, we have to evaluate the end-condition E and conditionally jump to L_4, respectively L_2.

Label L_2 marks the body S_2 and thereafter, we jump back to the start of the loop at L_3. Lastly, we emit label L_4, which will be reached should the end-condition E evaluate to true.

Figure 5 shows a procedure that sums up all numbers from 0 to n and returns the result in r.

``
procedure sum(int n, int r)
  local int k = 0
 	
  from k = 0 do
        r += k
  loop
        k += 1
  until k = n
 	
  delocal int k=n
``
	``
	begin sum(n0, r0)
	  k0 := 0 ^ (0 ^ 0)
	  -> L1(n0, k0, r0)
	  L1(n1, k1, r1)L3 <- k1 == 0
	  r2 := r1 + (k1 ^ 0)
	  k1 == n1 -> L4(k1, n1, r2)L2
	  L2(k2, n2, r3) <-
	  k3 := k2 + (1 ^ 0)
	  -> L3(n2, k3, r3)
	  L4(k4, n3, r4) <-
	  0 := k4 ^ (n3 ^ 0)
	end sum(n3, r4)
	``

```
procedure sum(int n, int r)
  local int k = 0

  from k = 0 do
        r += k
  loop
        k += 1
  until k = n

  delocal int k=n
```

```
begin sum(n0, r0)
  k0 := 0 ^ (0 ^ 0)
  -> L1(n0, k0, r0)
  L1(n1, k1, r1)L3 <- k1 == 0
  r2 := r1 + (k1 ^ 0)
  k1 == n1 -> L4(k1, n1, r2)L2
  L2(k2, n2, r3) <-
  k3 := k2 + (1 ^ 0)
  -> L3(n2, k3, r3)
  L4(k4, n3, r4) <-
  0 := k4 ^ (n3 ^ 0)
end sum(n3, r4)
```

Fig. 5. Example for translation of loops

3.6 Procedure Calls

Procedure calls can easily be translated to RSSA, since RSSA provides a call mechanism, too. Janus' procedure calls have already been designed in a careful way that avoids problems with reversibility. That is, call-by-reference is the only parameter-passing mode, there are no global variables, and it is not possible to use a variable multiple times in the same procedure call to prohibit aliasing. Hence, we can use the RSSA call statement *(y, ...):=call l(x, ...)* where l will be the name of the called procedure, the *x's* are the parameters, which will be destroyed after the call in case they are variables. The final values of the parameters after the call will be copied into the *y's*. An example is shown in Fig. 6.

3.7 Arrays and Stacks

Arrays and stacks are the only type constructors available in Janus. Arrays provide, as one would expect, random access to a defined number of integers only in one dimension.

Since access to memory locations using *M[A]* is restricted to specific operands and instructions in RSSA, code has to be generated to manage the memory used for arrays and stacks, as well as code to implement array access and operations on the stack. Suppose a has been declared as an array of 10 integer values, the RSSA code to access an array element *a[index]* will contain the evaluation of

```
procedure inc(int n, int res)      begin inc(n0, res0)
  res += n                           res1 := res0 + (n0 ^ 0)
  res += 1                           res2 := res1 + (1 ^ 0)
                                   end inc(n0, res1)
procedure main()
  int i                            begin main(i0, x0)
  int x                              i1 := i0 + (10 ^ 0)
  i += 10                            (i2,x1) := call inc(i1,x0)
  call inc(i, x)                   end main(i2, x1)
```

Fig. 6. Example for translation of a procedure call

the index expression, add the result to the base address, e.g. $a_{ref} + index$ and assign it to a temporary variable T. After accessing the memory at address T, we need to destroy T with a finalizer. Lastly, we compute the reverse of the index expression (see Sect. 3.3).

In addition to arrays, Janus provides the feature to declare stacks as data structures. As usual, stacks can be manipulated using *empty*, *top*, *push*, and *pop* operations.

We have to add a remark here: The language definition of Janus [17] is – to our minds – not entirely clear about whether stacks can be assigned to each other, and what the semantics are (deep copy vs. shallow copy), e.g. *local stack s1=s2*. We have chosen to allow these kinds of assignments and apply deep copies. The implementation of this behaviour is encapsulated in a separate class and could easily be adapted to different semantics.

Please note that in RSSA M is not an identifier denoting the name of an array, rather M is fixed and stands for "memory", but can be addressed "like" an array.

Hence, the generated code for a stack s contains two variables s_{ref} and s_{top} with s_{ref} being the base address of the stack in memory and s_{top} pointing to the next free element of the stack.

A *push(n,s)* operation (n being a variable and s being a stack) will be translated into multiple RSSA commands: Firstly, we have to compute the address of the memory location where the value to be pushed will be stored, i.e. $s_{ref} + s_{top}$. Due to the limitations of RSSA, this address needs to be stored in a temporary variable $T0$. Now, $M[T0]$ needs to be updated with the current value of n. Moreover, the semantics of Janus is that the variable n will be "zero-cleared" thereafter. Hence, we use the special assignment $n1 := M[T0] := n$ as defined in RSSA, which will set the top-most element of the stack to n, destroy n, and store the former value into $n1$, i.e. the next version of n. In the subsequent RSSA instruction, it will be verified that $n1$ is zero. Lastly, we have to undo the computation of the temporary variable $T0$ with the inverse of its computation, and increment *s_top*.

As mentioned before, we have not defined the translation scheme for *pop*, rather we instruct the code generator to work backwards, i.e., the RSSA commands will be

emitted in reverse order and each command will be inverted, that is, in an assignment left- and right-hand side are switched, plus becomes minus, etc.

The Java code for the implementation of the reverse code generator is remarkably small and itself uses a stack to intermediately store "forward" RSSA commands. Figure 7 shows an example.

```
procedure main()
  stack s
  int n
  int m

  push(n, s)
  pop(m, s)
```

```
begin main(s_ref0, s_top0, n0, m0)
  // push(n,s)
  T0 := 0 + (s_ref0 + s_top0)
  n1 := M[T0] := n0
  0 := 0 ^ (n1 ^ 0)
  0 := T0 - (s_ref0 + s_top0)
  s_top1 := s_top0 + (1 ^ 0)
  // pop(m, s)
  s_top2 := s_top1 - (1 ^ 0)
  T1 := 0 + (s_ref1 + s_top2)
  0 := 0 ^ (m0 ^ 0)
  m1 := M[T1] := m0
  0 := T1 - (s_ref1 + s_top2)
end main(s_ref1, s_top2, n1, m1)
```

Fig. 7. Example for translation of stacks

3.8 Implementation

The translations, which are part of the RSSA backend, have been implemented as a syntax-directed translation, using the well-known visitor pattern [6].

For each of Janus' constructs, we have defined a translation scheme as explained above. When given the abstract syntax-tree of an input program, our backend traverses this tree and generates instructions according to the scheme. Statements are translated into a series of instructions, while expressions are mainly translated as operands. If it is not possible to translate an expression directly into a single operand, we need to emit instructions, which bind the value of the expression to a temporary variable and destroy this variable afterwards.

The output of our code generator is a list of the generated instructions. When an instruction is emitted during code generation, it is appended to the list.

As mentioned earlier, our code generator is capable of reverse code generation. This behavior is provided by a single method, that accepts an unparameterized lambda expression, whose body may contain arbitrary Java statements. Before these Java statements are executed, the internal state of the code generator is changed so that emitted instructions are no longer appended to the output list but rather to a temporary data structure. The Java statements passed to this method are then executed, which are able to emit instructions without being aware of the inverted generation direction. After the Java statements have

been executed, the temporary data structure holds all instructions emitted by these statements. These instructions are then retrieved in reverse order and are inverted according to the rules described in [13] before they are emitted by the code generator.

Using this technique we were able to considerably reduce the amount of Java code in the code generator as opposed to the former code generator with "hand-crafted" reverse code.

4 Results

We have described rc3, consisting of a frontend for Janus as well a set of backends with RSSA being the most important of them.

Since the compiler developed by Axelsen [4] was not available to us, we used the online Janus interpreter ("Janus playground") [15] by Copenhagen University to compare the results of the execution of Janus test programs against our interpreter, RSSA, and Three-Address-Code back-ends.

Some results for sample Janus programs are shown in Table 2.

Table 2. Experiments

Sample Program		original	optimized
feistelcipher.ja	Janus loc	211	
	rssa instructions (loc)	598	531
	executed rssa instructions	61355	59152
C Exec	average c execution time in clocks	159	156
Janus-to-RSSA	average compile time in ms	189	228
Janus-to-RSSA-to-C	average compile time in ms	271	244
teax.ja	Janus loc	88	
	rssa instructions (loc)	632	418
	executed rssa instructions	30957	20513
C Exec	average c execution time in clocks	101	92
Janus-to-RSSA	average compile time in ms	163	191
Janus-to-RSSA-to-C	average compile time in ms	191	189
njvm-v4.ja + simple.bin	Janus loc	544	
	rssa instructions (loc)	2369	2109
	executed rssa instructions	24203	22440
C Exec	average c execution time in clocks	105	91
Janus-to-RSSA	average compile time in ms	573	671
Janus-to-RSSA-to-C	average compile time in ms	680	661

Feistelcipher and Teax are both programs implementing encryption schemes; NJVM is a virtual machine for a small imperative language. Lines of code are

excluding empty lines as well as lines with comments only. As you can see, the time required for compilation even for the largest program is well below one second. For instance, the Feistelcipher Janus program is compiled to a program of 598 RSSA instructions. Running this program executes 61335 instructions in the version further compiled to C, requiring 159 microseconds. In case of the Teax program, the speedup through optimization (see Table 2) is 33%, measured in the number of RSSA instructions executed. All tests were executed on a Linux computer with an AMD Ryzen 5 processor with a clock speed of 3600 Mhz using GCC 10.2.0 to create executables from the generated C-code.

Due to the "classic" implementation of the compiler in multiple phases, it is easy to add more language features to Janus, as well as to add further backends.

The ability to translate Janus and RSSA each to two non-reversible C-programs has also been proven to be extremely helpful as it facilitates quick regression testing of the compiler.

The compiler including all three backends comprises roughly 12.500 lines of Java code (excluding the generated lexer and parser). As you can see, the generated RSSA code is typically 3–7 times longer than the original Janus code, mostly due to the split of complex expressions into SSA's and the insertion of finalizers.

The C-Code generated from the RSSA code is considerably longer, mainly due to the insertion of C-functions for memory management, and of course because C-code is generated twice (forwards and backwards).

5 Conclusions and Outlook

We have shown a scheme for compiling reversible Janus programs into reversible RSSA code including the novel method for reverse code generation. The compiler has been implemented and is being extended to be able to optimize the generated RSSA code. In addition, a virtual machine for RSSA has been created that includes a debugger with a GDB-like CLI, allowing to step through the program forwards and backwards.

Mogensen [13] provides some suggestions on potential optimizations, further work or implementations of these are currently not known to us.

As per the time of writing of this paper, we have implemented local common subexpression elimination and constant propagation/folding (i.e., within a building block) using a directed acyclic multigraph. Work is underway to explore data-flow analysis of RSSA – however, the inherent requirements of reversible computing seem to require extensions of "traditional" algorithms to be able to apply them to reversible languages: Traditional data flow analysis determines IN and OUT sets either in a forwards manner, i.e. stepping from one basic block to its successors or vice versa. But, in a reversible world, "forwards" and "backwards" in terms of the direction of execution are not distinguishable, making the application of these well-known algorithms by Kildall [8] and others quite difficult.

A report on first optimizations (local common-subexpression elimination and constant propagation/folding) on the generated RSSA code will be presented at

ACM SOAP 2021. Future work will also include the formalization of the code generation scheme from Janus to RSSA, as well as a more detailed description of the compiler techniques applied for the reverse code generator.

The current state of the work looks very promising and should be helpful to gain further insights into reversible programming languages and compilers.

Acknowledgements. We would like to thank the reviewers who had provided valuable feedback and suggestions for improvement.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading (1986)
2. Alpern, B., Wegman, M.N., Zadeck, F.K.: Detecting equality of variables in programs. In: Ferrante, J., Mager, P. (eds.) Principles of Programming Languages (POPL 1988), pp. 1–11. ACM Press (1988). https://doi.org/10.1145/73560.73561
3. Appel, A.W.: Modern Compiler Implementation in Java. Cambridge University Press, Cambridge (1998)
4. Axelsen, H.B.: Clean translation of an imperative reversible programming language. In: Knoop, J. (ed.) CC 2011. LNCS, vol. 6601, pp. 144–163. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19861-8_9
5. Frank, M.P.: The future of computing depends on making it reversible. IEEE Spectrum, September 2017
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Boston (1995)
7. Hudson, S., Flannery, F., Scott Ananian, C., Petter, M.: CUP User's Manual. Technische Universität München, v0.11b edn. (2014) http://www2.in.tum.de/projects/cup/docs.php
8. Kildall, G.A.: A unified approach to global program optimization. In: Fischer, P.C., Ullman, J.D. (eds.) Principles of Programming Languages (POPL 1973), pp. 194–206. ACM Press (1973)
9. Klein, G., Rowe, S., Décamps, R.: JFlex User's Manual, version 1.8.2 edn. (2020). https://www.jflex.de/manual.html
10. Landauer, R.: Irreversibility and heat generation in the computing process. IBM J. Res. Dev. **5**(3), 183–191 (1961)
11. Lutz, C.: Janus - A Time-Reversible Language (1986), http://tetsuo.jp/ref/janus.pdf, Letter to R. Landauer
12. Mogensen, T.Æ.: Partial evaluation of the reversible language Janus. In: Khoo, S., Siek, J.G. (eds.) SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2011), pp. 23–32. ACM Press (2011). https://doi.org/10.1145/1929501.1929506
13. Mogensen, T.Æ.: RSSA: a reversible SSA form. In: Mazzara, M., Voronkov, A. (eds.) PSI 2015. LNCS, vol. 9609, pp. 203–217. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41579-6_16
14. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann (1997)
15. Nielsen, C.S., Budde, M., Thomsen, M.K.: Program Inversion and Reversible Computation - Janus Extended Playground http://topps.diku.dk/pirc/?id=janusP

16. Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Global value numbers and redundant computations. In: Ferrante, J., Mager, P. (eds.) Principles of Programming Languages (POPL 1988), pp. 12–27. ACM Press (1988). https://doi.org/10.1145/73560.73562
17. Yokoyama, T., Axelsen, H.B., Glück, R.: Principles of a reversible programming language. In: Ramírez, A., Bilardi, G., Gschwind, M. (eds.) Computing Frontiers, pp. 43–54. ACM Press (2008). https://doi.org/10.1145/1366230.1366239
18. Yokoyama, T., Glück, R.: A reversible programming language and its invertible self-interpreter. In: Ramalingam, G., Visser, E. (eds.) SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM 2007), pp. 144–153. ACM Press (2007). https://doi.org/10.1145/1244381.1244404
19. Yokoyama, T., Axelsen, H.B., Glück, R.: Fundamentals of reversible flowchart languages. Theoret. Comput. Sci. **611**, 87–115 (2016). https://doi.org/10.1016/j.tcs.2015.07.046