Partial Evaluation of Reversible Flowchart Programs

Normann, Louis Marott; Glück, Robert

Published in:
PEPM 2024 - Proceedings of the 2024 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation

DOI:
10.1145/3635800.3636967

Publication date:
2024

Document version
Publisher's PDF, also known as Version of record

Citation for published version (APA):
https://doi.org/10.1145/3635800.3636967
Partial Evaluation of Reversible Flowchart Programs

Louis Marott Normann
DIKU, Department of Computer Science
University of Copenhagen
Denmark
lono@di.ku.dk

Robert Glück
DIKU, Department of Computer Science
University of Copenhagen
Denmark
glueck@di.ku.dk

Abstract
Flowchart languages are traditionally used to study the foundations of partial evaluation. This article presents a systematic and formal development of a method for partial evaluation of a reversible flowchart language. The results confirm that partial evaluation in this unconventional computing paradigm shows effects consistent with traditional partial evaluation. Experiments include specializing a symmetric encryption algorithm and a reversible interpreter for Bennett’s reversible Turing machines. A defining feature of reversible languages is their invertibility. This study reports the first experiments composing program inversion and partial evaluation. The presented method is fully implemented. It is potentially of interest because reversible computing has found applications in areas as diverse as low-power computing, debugging, robotics, and quantum-inspired computing.

CCS Concepts: • Software and its engineering → Software maintenance tools; Language features; Imperative languages; • Theory of computation → Operational semantics.

Keywords: Partial Evaluation, Reversible Computing, Flowchart Languages, Program Transformations, Program Inversion

1 Introduction
Partial evaluation has been carried out in several different types of languages. However, little work has been done focusing on unconventional language paradigms, such as reversible computing.

One motivation for reversible languages, which is also mentioned in related work [30], is the idea of energy efficiency [6, 28]. By specializing a program, the residual program will compute results in fewer computations than the original, thereby increasing efficiency.

Sometimes reversibility is a necessity, as in the case of quantum-inspired computing, where the irreversible erasure of information is considered harmful. Only recently it was shown that symbolic techniques and partial evaluation enable efficient simulations of quantum circuits [5].

Flowchart languages are traditionally used to study the foundations of partial evaluation (e.g., [4, 11, 18, 20, 22, 23]). They are the Swiss Army knife of partial evaluation. The semantic foundations of reversible flowchart languages [34] have been established, covering a variety of imperative reversible languages ranging from reversible machine code to high-level structured reversible languages, such as Janus [35], RL [34], and R-WHILE [15, 16]. Defining features are their invertibility and computation without information erasure.

The language used in this study is based on the reversible flowchart language RL, extended with constructs from R-WHILE. It features reversible store updates, reversible pattern matching, lists as dynamic data structures and symbolic data, opening new possibilities for reversible program manipulation. Little is known about the possibilities for partial evaluation in reversible computing. The few works we are aware of were mentioned above [5, 30].

Similarly to the partial evaluator for Janus [29], we use offline partial evaluation [25, 26]. The strengths of the two partial evaluators are comparable as both use a uniform binding-time analysis [20]. We are handling a reversible flowchart language with pattern matching and lists, while the partial evaluator for Janus handles a procedural language with arrays, although an unstructured language is used internally [30].

The goal of this study is to contribute to the foundation of partial evaluation in this unconventional computing paradigm. The main practical challenge is ensuring the reversibility of the specialized programs.
The results also show where there is room for improvement: in the optimization of residual assertions and in more accurate analyses of reversible pattern matching.

Further details about the language will be described in Sec. 2. An ad hoc example of partial evaluation in RL will be in Sec. 3, followed by an analysis of what was learned during the specialization. This analysis ends with the selection of the type of partial evaluation for RL. The partial evaluator is presented and formalized in Sec. 4. Experimental results of partial evaluation and the combination with program inversion are reported in Sec. 5. Conclusions are drawn and future work is discussed in Sec. 6.

We follow the partial evaluation terminology for flowchart languages [20, 25]. Reversible computing from a programming language perspective is presented elsewhere [17].

1. Syntax

The syntax for RL can be seen in Fig. 1 with an example program in Lst. 1. The first part of an RL program is the variable declaration, which shows input and output variables. Optionally one can also declare a set of temporary variables. All non-input variables are initialized to the atom nil, while all non-output variables must be nil at the end of the program.

The declaration is followed by at least one block. Each block is identified by a unique label, and has three parts. Firstly, the come-from statement specifies which other blocks the current block is identified by a unique label, and has three parts. Secondly, the body is a list of steps. Finally, the jump-statement specifies which blocks can follow during evaluation. Come-from statements are unconditional (from l), conditional (fi e from l else l), where the previous block can be determined by the value of the expression e, or the program entry. Jump statements follow the same convention. For a program to be well-formed, it must have one entry and exit.

There are four kinds of steps. Reversible assignments (x⊕= e) update the value of x in the store with the reversible operator ⊕ and the value of the expression e. For well-formedness, x must not occur in e. Reversible replacements (q₁<-q₂) obtain a value from the pattern q₂ and update the value of the pattern q₁ with this value. Variables can only be mentioned once in a pattern. skip and assert have no effect on the store. All steps are injective mappings on the store [15, 34].

In the core grammar, operators are n-ary functions. In the example programs, binary operators are written in infix form. Expressions and constants are defined as usual. Also, for RL programs to be well-formed, all blocks must have unique labels, and each label used in the jump of a block must refer to another block in the program which also must refer to the original block in its come-from statement. All variables must be declared at the beginning of the program. Well-formedness ensures the reversibility of programs.

2. The Reversible Flowchart Language

This section presents RL, a reversible flowchart language with symbolic data and lists. RL is based on a previous language with the same name, which had arrays and integers [34] instead. The symbolic constructs are inspired by the structured reversible language, R-WHILE [15, 16].

<table>
<thead>
<tr>
<th>1</th>
<th>(ys n) -&gt; (xs ys) with (x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>init: entry</td>
</tr>
<tr>
<td>3</td>
<td>if n = '0 goto stop else loop</td>
</tr>
<tr>
<td>4</td>
<td>loop: fi xs = 'nil from init else loop</td>
</tr>
<tr>
<td>5</td>
<td>(x . ys) &lt;- ys</td>
</tr>
<tr>
<td>6</td>
<td>xs &lt;- (x . xs)</td>
</tr>
<tr>
<td>7</td>
<td>n -= '1</td>
</tr>
<tr>
<td>8</td>
<td>if n = '0 goto stop else loop</td>
</tr>
<tr>
<td>9</td>
<td>stop: fi xs = 'nil from init else loop</td>
</tr>
<tr>
<td>10</td>
<td>n ^= '0</td>
</tr>
<tr>
<td>11</td>
<td>exit</td>
</tr>
</tbody>
</table>

Listing 1. List partition program

---

1https://github.com/Yakokse/PERevFlow
2.2 An Example Program

An example program can be seen in Lst. 1. The program takes the first $n$ elements of a list $ys$ and puts them in a list $xs$. For simplicity, the elements in $xs$ are in reversed order.

The program's main structure is a loop that is repeated $n$ times. In this loop the head of $ys$ is popped into the variable $x$ on line 6 and it is then pushed onto $xs$ on line 7. After the loop, since $n$ is not a part of the output, it must be $nil$ at the exit. The value of $n$ is changed from 0 to $nil$ using the symbolic xor operator $\oplus$ defined in Sec. 2.3, Eq. (1).

The program can be traced with a given input. This will require representing the computational state of the program before evaluating each block. Traditionally this consists of the current store and the label of the current block, however the previous label is needed due to come-from statements. Therefore the state will be represented by the tuple $(\sigma, l_1, l_2)$ where $\sigma$ is the current store, and $l_1$ and $l_2$ are the previous and current label respectively. For the sake of saving space in the following trace, the store $\sigma = [xs \mapsto a, ys \mapsto b, n \mapsto c, x \mapsto d]$ will be denoted as $[a, b, c, d]$, and traditional lists $(a, b, nil)$ will be denoted as $[a, b]$.

Here is a trace of the example program with input $[ys \mapsto [1, 2, 3], n \mapsto 2]$ and the output $[xs \mapsto [2, 1], ys \mapsto [3]]$.

$\sigma_{from} \mapsto l_1 \Rightarrow \text{cleared-in}(\sigma)$
$\sigma_{from} \mapsto entry \Rightarrow \text{entry}$
$\sigma_{from} \mapsto froml \Rightarrow l_1 \mapsto \text{is-true}(\sigma)$
$\sigma_{from} \mapsto fi \mapsto from1i \mapsto else \mapsto l_2 \mapsto \text{is-false}(\sigma)$

$\sigma_{jump} \mapsto j \Rightarrow \text{cleared-out}(\sigma)$

Figure 2. Operational semantics of RL.

$\sigma_{jump} \mapsto goto \Rightarrow \text{goto l} \mapsto \text{exit}$

$\sigma_{jump} \mapsto exit \Rightarrow \text{exit}$

Figure 3. Semantic Values
of the $\Rightarrow_p$ transition is taken, where each transition gives the next computational state.

The jump judgement $(\sigma \triangleright_{\text{jump}} j \Rightarrow l)$ and come-from judgement $(\sigma \triangleright_{\text{from}} k \Rightarrow l)$ show the appropriate label for the given statement and store. The "is-true" and "is-false" predicates are defined s.t. is-false(nil) and is-true(\sigma), $\forall \sigma \neq \text{nil}$. The "cleared-in" predicate is defined s.t. cleared-in(\sigma) for all stores \sigma where all non-input variables are nil. The "cleared-out" predicate is defined identically except all non-output variables must be nil.

The expression judgement $(\sigma \triangleright_{\exp} e \Rightarrow \nu)$ denotes the value \nu of an expression e under a given store \sigma. $\llbracket \cdot \rrbracket$ is a function that maps syntactic objects to their respective semantic objects. For a formal description of all operator semantics, see Appdx. A.

The pattern judgement $(\sigma \triangleright_{\text{pat}} q \Rightarrow (\sigma', \nu))$ shows the relation between stores, patterns, and values. The judgement is only used with reversible replacements $(q_1 \leftarrow q_2)$. It is first used to find an intermediary value and store from $q_2$ while setting all the value of all variables in $q_2$ to nil. The intermediary value is then assigned to $q_1$, assigning a new value to all of the variables that occur in the pattern. Patterns are linear, meaning a variable can only occur once in a pattern.

The step judgement $(\sigma \triangleright_{\text{step}} s \Rightarrow \sigma')$ shows the resulting store \sigma' from evaluating the step s under the store \sigma. The skip statement has no effect, while the assert(e)-statement has no effect, but only succeeds if the value of the expression e is true. The reversible replacement is described previously with the pattern judgement. The reversible update ($x \equiv e$) has C-like semantics, except that x must not occur in e. The reversibility of an update requires e to be independent of x.

The assignment operators + and − perform modular arithmetic on integers and $\oplus$ (symbolic xor) works on all values, and has the following definition:

$$\llbracket \oplus \rrbracket(v_1, v_2) = \begin{cases} v_2, & \text{if } v_1 = \text{nil} \\ \text{nil}, & \text{if } v_1 \neq \text{nil} \end{cases}$$ (1)

3 First Steps: Partial Evaluation of RL

A partial evaluator pe for RL has the following property [30]. For all RL-programs p, r and stores $\sigma_1, \sigma_2, \sigma_3, \sigma_4$:

$$\llbracket \llbracket p \rrbracket \rrbracket (r, \sigma_1) \land \llbracket \llbracket r \rrbracket \rrbracket (r, \sigma_2) = \sigma_4$$

$$\quad \Rightarrow \llbracket \llbracket \llbracket p \rrbracket \rrbracket \rrbracket (r, \sigma_1 \cup \sigma_2) = \sigma_3 \cup \sigma_4$$ (2)

Program p is partially evaluated by pe w.r.t. to the static (known) part $\sigma_1$ of its input store $\sigma_1 \cup \sigma_2$. The result is the residual program r written in RL and the static part $\sigma_2$ of its output store $\sigma_2 \cup \sigma_4$, which can be pre-computed from $\sigma_1$. Given the remaining dynamic (unknown) part $\sigma_2$, r computes the remaining $\sigma_4$. Program p applied to the complete input store $\sigma_2 \cup \sigma_4$ returns the complete output store $\sigma_2 \cup \sigma_4$. Thus, pe stages the one-stage computation of p into two stages. Note that not only the arity but also the coarity of p is reduced when it is specialized into r.

A traditional partial evaluator reduces the arity of the source program and leaves its coarity unchanged [25, 32]. The reason for the unconventional definition in (2), where pe also returns the static store $\sigma_3$, is the reversibility of the RL-programs, which does not allow deletion of values. A program in an irreversible flowchart language copies the desired result from the store with a return-statement and discards all other values [20, Def. 2]. A reversible flowchart program in RL returns the output store specified in the variable declaration, i.e. it implements an n-to-m function. Definition (2) discards no parts of the output store.

3.1 An Example of Ad Hoc Partial Evaluation

As a first step towards an RL-partial evaluator, we will naively follow a three-step procedure akin to the one for FCL [20, 25], an irreversible flowchart language:

1. State collection: In the given program p, start from $(\sigma, \text{entry}, l)$, where $\sigma$ is a store of the variables in p with some values abstracted by the symbol D and collect all reachable states $(\sigma_i, l_i, l_i')$. For each state, put $(l_i, \sigma_i, l_i')$ in a set $S$, where $(l_i, \sigma_i)$ is the label and state upon entry of the current block, and $(l_i', \sigma_i')$ is the label and state upon entry of the previous block. The second tuple will be needed when verifying and rewriting come-from statements.

2. Program point specialization: For each state $(l_i, \sigma_i, l_i', \sigma_i') \in S$, specialize the block $l_i$ with respect to $\sigma_i$, and $(l_i', \sigma_i')$. This new specialized block is placed into the residual program.

3. Transition compression: The residual program is optimized by merging blocks that are directly connected by unconditional goto’s and from’s.

3.1.1 State Collection. We will try to specialize the list partition program in Lst. 1. First an abstract trace is made to collect all reachable states. For this we use $\sigma_1 = \{ \sigma \mapsto \text{2} \}$ as the known (static) input. Values that are unknown (dynamic) at specialization time are marked D. The trace is then as follows, showing the elements in $S$ and the exit state. We will follow the same store notation as in Sec. 2.2.

- (init, [], D, 2, nil), entry, [], D, 2, nil)
- (loop, D, 1, [D, 2, nil], loop, D, 2, nil)
- (stop, D, 1, [D, 2, nil], stop, D, 0, D)
- (exit, D, 2, nil, D, stop, D, 0, D)

Each line of the trace is then a reachable state collected. As seen in the trace, the program will not go in an infinite loop with the given static input. However, there is still no guarantee that an arbitrary program with a finite non-repeating trace will terminate successfully, given possible sources of failure (e.g. assertions and come-from statements).
3.1.2 Program Point Specialization. For simplicity, each residual block will be labeled by the tuple \((l_1, \sigma)\), where \(\sigma\) is the store at the entry of the block. The blocks are specialized as follows and the residual blocks can be seen in Lst. 2:

- State \((\text{init}, [[], D, 2, \text{nil}])\), entry, \([[], D, 2, \text{nil}]\): Since \(n\) is static, the if statement can be determined and is therefore converted into a goto. Block in lines 1-2.
- State \((\text{loop}, [[], D, 2, \text{nil}]), \text{init}, [[], D, 2, \text{nil}]\): The if statement can be determined since it only depends on the static \(x_s\). The residual label in the from-statement is generated using the store from the previous state. The two replacements depend on the dynamic \(y_s\), so they must be in the generated residual code. Since \(x\) and \(x_s\) depend on \(y_s\), they become dynamic in the store. The decrement of \(n\) and the if can both be optimized since they are static. Block in lines 4-7.
- State \((\text{loop}, [D, D, 1, D]), \text{loop}, [[], D, 2, \text{nil}]\): From now on, the conditional \(fi\) in the loop block will not be reduced to an unconditional from since \(x_s\) is dynamic. The label \(\text{init}\) cannot be decorated with a store because it is not yet known whether and, if so, which store is used to jump from \(\text{init}\) to \((\text{loop}, [D, D, 1, D])\). Block in lines 9-13.
- State \((\text{stop}, [D, D, 0, D]), \text{loop}, [D, D, 1, D]\): The value of \(n\) becomes \(\text{nil}\), which means all statically known non-output variables have the required value \(\text{nil}\). Block in lines 15-17.

3.1.3 Transition Compression. Compressing all unconditional jumps while ensuring that the partial evaluator does not loop [20] is vital for transition compression in an irreversible language. However there are further restrictions in RL, namely from-statements. As mentioned previously, the semantics of a reversible program must be preserved. This means that generally the conditional expression in a \(fi\) must be checked and cannot be removed.

In the residual program generated above (Lst. 2), the only goto-from transition that can be compressed is between the blocks \((\text{init}, [[], D, 2, \text{nil}])\) and \((\text{loop}, [[], D, 2, \text{nil}])\).

### Listing 2. Program in Lst. 1 specialized ad hoc w.r.t. \(n \mapsto 2\)

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>((\text{init}, [[], D, 2, \text{nil}]), \text{entry} \Rightarrow \text{goto} (\text{loop}, [[], D, 2, \text{nil}]))</td>
</tr>
<tr>
<td>2</td>
<td>((\text{loop}, [[], D, 2, \text{nil}]), \text{from} (\text{init}, [[], D, 2, \text{nil}]) \Rightarrow (x \ . y) \leftarrow y_s \Rightarrow x_s \leftarrow (x \ . x_s) \Rightarrow \text{goto} (\text{loop}, [D, D, 1, D]))</td>
</tr>
<tr>
<td>3</td>
<td>((\text{loop}, [D, D, 1, D]), \text{if} x_s = \text{`nil'} \Rightarrow \text{assert} (l_1) \Rightarrow \text{from} (\text{init}, [[], D, 2, \text{nil}]) \Rightarrow (x \ . y) \leftarrow y_s \Rightarrow x_s \leftarrow (x \ . x_s) \Rightarrow \text{goto} (\text{stop}, [D, D, 0, D]))</td>
</tr>
<tr>
<td>4</td>
<td>((\text{stop}, [D, D, 0, D]), \text{if} x_s = \text{`nil'} \Rightarrow \text{assert} (l_1), \text{else} (\text{loop}, [D, D, 1, D]))</td>
</tr>
<tr>
<td>5</td>
<td>((\text{stop}, [D, D, 0, D]), \text{exit} \Rightarrow \text{assert} (l_1))</td>
</tr>
</tbody>
</table>

3.2 Ensuring Reversibility of Residual Programs

We now analyze the main problem points that we need to consider when defining a partial evaluator for RL.

3.2.1 Reducing Conditional From-Statements. As seen in Lst. 2, only the label of the previous block is decorated with the store in a conditional from-statement. If \((l_1, \sigma)\) goes to a block with \(fi\) \(e\) from \(l_1\) else \(l_2\), it does not mean that \((l_2, \sigma)\) will do so because different blocks can have different effects on the store. It is possible that only \((l_1, \sigma)\) goes to that block or that \(l_2\) is never reached during specialization.

To create a well-formed RL-program, we replace all ‘one-armed’ conditional from-statements by unconditional from-statements and preserve the condition \(e\) in an assert:

\[
\begin{align*}
fi & \ e \ \text{from} \ (l_1, [...]) \Rightarrow \ \text{from} \ (l_1, [...]) \\
\text{else} & \ (l_2, [...]) \ \Rightarrow \ \text{assert} \ (e)
\end{align*}
\]

Another approach is to insert a dummy block [30]. In both cases, this transformation is only possible in post-processing after the complete residual program has been generated.

3.2.2 Single Exit Requirement. A possible danger during partial evaluation is specializing the exit block multiple times due to it being reached with multiple different states. While it did not happen in our example, in general it would result in the residual program being malformed. In a conventional language multiple exits are not an issue, but since RL is reversible there can only be one single exit block.

Fixing this while still doing more than a trivial partial evaluation, will require changing the approach. Some variables may be explicitly declared dynamic through generalization [20] until the exit block is reached with only a single uniform state. Lifting static values can also be done by a post-processor, but requires additional blocks that merge the control flow so that it ends in a single exit block.

In another specialization of a reversible language [30], this is solved by explicitly declaring variables as input, output, or local. It then requires that all output variables are either dynamic or constant (read only) throughout the specialization, and local variables must be cleared to a default value (e.g., \(\text{nil}\)) before the program exit (the converse of initializing a local variable to a default value). We will adopt this conservative approach in the partial evaluator.

3.2.3 Injectable Static Store Update. However, there is another important concern for RL. If the specialization of a block with respect to two different static stores ends with the same static store, then they will jump to the same next block in the residual program. This then means that the next residual block cannot distinguish which of the previous two blocks the control flow comes from. The residual program would then be irreversible. It follows that the update of the static store at specialization time must be injective.
In RL, two statements can update the store:

1. reversible assignments and
2. replacements.

Consider specializing the block

1: from 11
y := x
x := y
goto 12

w.r.t. the stores \([x \mapsto 3, y \mapsto D]\) and \([x \mapsto 5, y \mapsto D]\), then \(x\) becomes dynamic because it depends on the dynamic \(y\). The resulting stores are identical, \([x \mapsto D, y \mapsto D]\), and jump to the same residual block. Thus, the store transformation is not injective and results in the insertion of a new join point into the residual program. A simple solution is to enforce a uniform division \([25]\) that applies to the entire program.

Another potential problem is the specialization of a replacement by partially evaluating the matching. Consider specializing the replacement

\[ y \leftarrow (x \cdot y) \]

w.r.t. the same two stores as above. We know that \(x\) will be \(\text{nil}\)-cleared regardless of its initial value (here 3 or 5), and this can be used to statically update \(x\) to \(\text{nil}\). Here too, the store transformation is not injective because the resulting stores are identical \([x \mapsto \text{nil}, y \mapsto D]\). Our solution is to require that all variables in a replacement must be static in order to perform matching. Otherwise, all variables in a replacement must be dynamic so that no variable is updated, which also means that no lifting of static values occurs in a pattern.

### 3.2.4 Failing Assertions

In a conventional language the only error that can occur at partial evaluation time is in the evaluation of static expressions (e.g., division by zero). Gracefully aborting partial evaluation is considered acceptable and ‘defensive programming’ is seen as responsibility of the user. In RL, failure may occur at two new places: assertions and replacements. There are simply more chances to fail. Assertions are tested in from- and assert-statements and implicitly at the exit of a program to make sure that all non-output variables are \(\text{nil}\)-cleared. There are different design choices to handle expression, assertion and matching failures, and one choice will be presented in Sec. 4.

### 4 Formalization of the RL Partial Evaluator

In this section we will present and formalize the partial evaluator for RL. We begin by presenting the two-level language RL2 (Sec. 4.1) and defining well-formed RL2-programs (Sec. 4.2). The highlight of this section is the complete formalization of the specialization core of RL2 programs (Sec. 4.3). The formalization is completed by the definition of the RL-partial evaluator (Sec. 4.4). The section is illustrated by examples to explain the formal definitions.

---

**Listing 3. Annotated Partition program**

```
1 (ys n) -> (xs ys) with (x)
2 init: entry
3   if n = '0 goto stop else loop
4   loop: fi xs = lift('nil) from init else loop
5     (x \cdot ys) <= ys
6     xs \leftarrow (x \cdot xs)
7     n \rightarrow '1
8     if n = '0 goto stop else loop
9   stop: fi xs = lift('nil) from init else loop
10    n \rightarrow '0 goto stop else loop
11 exit
```

---

### 4.1 Two-Level Language

A common technique in partial evaluation is to annotate programs using a two-level language \([25]\). As in previous work \([20]\), a two-level language is defined for RL. The syntax can be seen in Appdx. B and is illustrated with an example in Lst. 3. Underlined constructs indicate that they are residual and non-underlined constructs are eliminable. The new RL2 syntax metavariabes are marked with a hat ‘\(\hat{\}\)’. The un-annotated versions of goto, exit, from, and entry are not used, but are still present for completeness.

There is a new unary operator, \(\text{lift}\), which occurs in expressions. It is intended to transform an eliminable piece of code into a residual value. There are no annotations on patterns since variables cannot be lifted in patterns, as seen in Sec. 3.2.3. Therefore, there is no need to annotate patterns.

The variables in the variable declaration are annotated, showing which variables are eliminable.

### 4.2 Well-Formedness for Two-level Programs

Introducing annotations to RL also introduces new sources of malformedness \([20]\). Consider for example the following two expressions:

\[ x + y, \ a \times b \]

The first expression is incorrect due to \(y\) being annotated as residual, but \(\times\) expecting two values. The second expression is incorrect due to a being annotated as eliminable and will produce a value while \(\times\) expects some residual code.

**Well-Formedness.** This notion of well-formedness is also expressed as a new rule system. The new semantic values and well-formedness judgements are shown in Fig. 4. The **variable judgement** \(\Delta \vdash_{\text{var}} \hat{x} [\hat{x}]\) shows how the annotated variable \(\hat{x}\) is well-formed w.r.t. the variable \(x\) and the division \(\Delta\). The rules are separated from the expression judgement, due to also being relevant in the variable declaration in the program judgement.

The **expression judgement** \(\Delta \vdash_{\text{exp}} e [\hat{e} : t]\) shows the well-formedness of an annotated expression \(\hat{e}\) w.r.t. the source expression \(e\) and a division \(\Delta\). Each annotated expression is also given a BT-type \(t\), signifying whether the expression is
New semantic values:
\[ t \in BT-\text{Type} = \{S, D\} \]
\[ \Delta \in \text{Divisions} = \text{Variables} \rightarrow BT-\text{Type} \]

Judgement \[ \Delta \vdash_{\text{var}} x [\hat{x}] \cdot \]

VarS \[ \Delta(x) = S \]
\[ \Delta \vdash_{\text{var}} x [x] \]

Judgement \[ \Delta \vdash_{\text{exp}} e [\hat{e} : t] \cdot \]

ConstS \[ \Delta \vdash_{\text{exp}} ^c \cdot [\hat{c} : S] \]
\[ \Delta \vdash_{\text{exp}} \cdot [\hat{c} : D] \]

Var \[ \Delta \vdash_{\text{exp}} x [\hat{x}] [\Delta(x)] \]
Lift \[ \Delta \vdash_{\text{exp}} e [\hat{e} : S] \]
\[ \forall i. \Delta \vdash_{\text{exp}} e_i [\hat{e}_i : S] \]

OrS \[ \Delta \vdash_{\text{exp}} \otimes (e_1, \ldots, e_n) [\otimes (\hat{e}_1, \ldots, \hat{e}_n) : S] \]
Cons \[ \Delta \vdash_{\text{pat}} q_1 [\hat{q}_1 : t] \]
\[ \Delta \vdash_{\text{pat}} q_2 [\hat{q}_2 : t] \]

Judgement \[ \Delta \vdash_{\text{step}} s [\hat{s}] \cdot \]

Skip \[ \Delta \vdash_{\text{step}} \text{skip} [\text{skip}] \]
Assignments \[ \Delta \vdash_{\text{step}} x @= e [\hat{x} @= \hat{e}] \]
\[ \Delta \vdash_{\text{step}} x @= e [\hat{x} @= \hat{e}] \]

AssertS \[ \Delta \vdash_{\text{step}} \text{assert}(e) [\text{assert}(\hat{e})] \]
\[ \Delta \vdash_{\text{step}} e [\hat{e} : D] \]

AssertD \[ \Delta \vdash_{\text{step}} \text{assert}(e) [\text{assert}(\hat{e})] \]
\[ \Delta \vdash_{\text{step}} q_1 [\hat{q}_1] \]
\[ \Delta \vdash_{\text{step}} q_2 [\hat{q}_2] \]

ReplaceS \[ \Delta \vdash_{\text{step}} q_1 \leftarrow q_2 [\hat{q}_1 \leftarrow \hat{q}_2] \]
\[ \Delta \vdash_{\text{step}} q_1 \leftarrow q_2 [\hat{q}_1 \leftarrow \hat{q}_2] \]

ReplaceD \[ \Delta \vdash_{\text{step}} q_1 \leftarrow q_2 [\hat{q}_1 \leftarrow \hat{q}_2] \]

Judgement \[ \Delta \vdash_{\text{jump}} f [\hat{f}] \cdot \]

From \[ \Delta \vdash_{\text{from}} \text{from} [\hat{f}] \]
Entry \[ \Delta \vdash_{\text{from}} \text{entry} [\text{entry}] \]

ForS \[ \Delta \vdash_{\text{exp}} e [\hat{e} : S] \]
\[ \Delta \vdash_{\text{exp}} e [\hat{e} : D] \]

ForD \[ \Delta \vdash_{\text{exp}} e [\hat{e} : D] \]

Judgement \[ \Delta \vdash_{\text{steps}} s^* [\hat{s}^*] \cdot \]

StepZero \[ \Delta \vdash_{\text{steps}} \cdot [\hat{s}] \]

StepMany \[ \Delta \vdash_{\text{steps}} s \hat{s}^* [\hat{s}^*] \]
\[ \Delta \vdash_{\text{steps}} s \hat{s}^* [\hat{s}^*] \]

Judgement \[ \Delta \vdash_{\text{block}} b [\hat{b}] \cdot \]

Block \[ \Delta \vdash_{\text{from}} k [\hat{k}] \]
\[ \Delta \vdash_{\text{steps}} s^* [\hat{s}^*] \]
\[ \Delta \vdash_{\text{jump}} j [\hat{j}] \]

Judgement \[ \Delta \vdash_{\text{blocks}} b^* [\hat{b}^*] \cdot \]

BlockOne \[ \Delta \vdash_{\text{blocks}} b [\hat{b}] \]
\[ \Delta \vdash_{\text{blocks}} b [\hat{b}] \]

BlockMany \[ \Delta \vdash_{\text{blocks}} b^* [\hat{b}^*] \]
\[ \Delta \vdash_{\text{blocks}} b^* [\hat{b}^*] \]

Judgement \[ \Delta \vdash_{\text{prog}} p [\hat{p}] \cdot \]

Prog \[ \forall x. \Delta \vdash_{\text{var}} x [\hat{x}] \]
\[ \Delta \vdash_{\text{blocks}} b^* [\hat{b}^*] \]
\[ \Delta \vdash_{\text{blocks}} b^* [\hat{b}^*] \]

[Figure 4. Wellformedness condition for RL2]
static (S) or dynamic (D). One can see that the lift operator transforms a static annotated expression into a dynamic one.

Since no annotation of patterns is needed, the pattern judgement $(Δ \vdash_{\text{pat}} q \{t\})$ only assigns a type to each pattern $q$. Any associated annotation happens in the replacement statement with the step judgement.

The step judgement $(Δ \vdash_{\text{step}} s \{\hat{s}\})$ is as expected, with the well-formedness of the annotation of each step $s$ depending on their associated expression or pattern. An exception to this is the skip-statement, which has no expression or pattern, and therefore can always be eliminable and residual.

The come-from judgement $(Δ \vdash_{\text{from}} k \{\hat{k}\})$ is notable, as it mirrors the jump judgement $(Δ \vdash_{\text{jump}} j \{\hat{j}\})$. This is a desirable property due to them being inverses, as seen in the inverter in Appdx. C. Furthermore, the jump judgement resembles that of previous work in FCL [20], which has a similar syntax and semantics for jump statements.

The remaining judgements serve to propagate the well-formedness requirements of the annotated program.

There are still several different well-formed ways of annotating an arbitrary program. These will differ by the produced residual code. A desired property of an annotation algorithm is that it produces the least residual constructs as possible. It is seen as relatively clear how to accomplished this in similar work [20]; therefore omitted due to space restrictions.

**Example.** Seen in Lst. 3 is the annotated version of the program seen in Lst. 1 w.r.t. a static $n$. All jumps except for the exit are marked as eliminable as they only depend on $n$. The 'nil’s in the come-froms are lifted since they are marked as static by our annotator, but are needed in a dynamic context with $x$s. This is because our annotator always lifts static expressions, even though constants could be marked as residual instead. This has no effect on the residual program.

**Binding-Time Analysis.** Before one can annotate the program, one first has to find a congruent division [22]. It is desirable that the division has as many static variables as possible while still producing a well-formed program. However there are various well-known algorithms for doing a uniform binding-time analysis (BTA) in the literature (e.g., fix-point iteration, constraint-based analyses) and we are not going to discuss it in detail (cf. [25]). The actual implementation uses a fix-point iteration. The uniform approach is similar to the core part of the Janus partial evaluator [30].

### 4.3 Specialization of RL2-Programs

The specialization phase is responsible for translating an input state along with an RL2 program into a residual RL program. The new semantic values and the judgements for expressions and steps can be found in Fig. 5, and the remaining judgements in Fig. 6.

The labels of the residual program will be annotated, meaning that they are a combination of their original source label and the static store at the start of their block. This can be changed in post-processing to produce lexically valid labels. This is needed due to blocks possibly being specialized multiple times with different stores.

**Failure Handling.** The set $A_{\perp}$ in Fig. 5 shows the syntax for annotating domains where failure might occur. In RL’s case, failure can occur in assertion statements (i.e. the associated expression evaluates to $\text{nil}$), in expressions (division by 0, $\text{hd}$ of an atom, etc.), in come-from statements (if the labels do not match), and at program exit (if non-output variables are not $\text{nil}$). In the interest of readability and to avoid overloading the rules, we focus only on the failures arising from assertions and come-from statements.

When a failure is encountered during the specialization of a block, the source program would also fail if it would enter the block with the same state. For further description of how failure is handled, see Sec. 4.4.

**Judgements.** The expression judgement $(\sigma \vdash_{\text{exp}} \hat{e} \Rightarrow w)$ shows how specializing an annotated expression $\hat{e}$ either gives an expression or a value. As the specialization only has to follow the annotations of the RL2 program, it can be seen that each rule knows whether the annotated sub-expressions will be specialized to expressions or values. For example OrS expects all sub-expressions to become values, while OrD expects them to become expressions. All of this is guaranteed by the well-formedness of the RL2 program. In the same way that $[\cdot]$ maps syntactic elements to their semantic counterparts, $[\cdot]^{-1}$ will map semantic elements to syntactic ones.

The step judgement $(\sigma \vdash_{\text{step}} \bar{s} \Rightarrow \langle \sigma', s' \rangle_{\perp})$ can transform the store $\sigma$ and produce residual steps $\bar{s}$. However, as indicated by the $\perp$, specializing a step may lead to failure. This is due to assert statements. In general the rules follow the approach one might expect as well, except for a few. The rule AssertE is responsible for handling failed assert statements, which will be propagated. Since patterns are not annotated there is no need for special judgements in the specializer, instead ReplaceS references the original semantic judgement $(\sigma \vdash_{\text{pat}} q \Rightarrow \langle \sigma', v \rangle_{\perp})$ defined in Fig. 2.

The step-list judgement $(\sigma \vdash_{\text{steps}} \bar{s} \Rightarrow \langle \sigma', s' \rangle_{\perp})$ can now also fail due to the step judgement failing. In general, the judgement will specialize an annotated step-list in sequence, giving the final store and the residual step-list, but if a failure is encountered anywhere in the sequence, it will fail as well.

The come-from judgement $(\sigma \vdash_{\text{from}} \langle \hat{k}, \hat{l} \rangle \Rightarrow k_{\perp})$ is responsible for residualizing a come-from statement, while also checking that the control flow obeys all restrictions. Assuming that the source program was well formed, this could then only fail in a fail statement. We choose only to fail gracefully when the error can occur in well-formed programs.
New semantic values:
\[ I \in \text{PE-Labels} = \text{Labels} \times \text{Stores} \]
\[ W \in \text{PE-Values} = \text{Values} \cup \text{Expressions} \]
\[ a \in A_L = A \cup \{ \bot \} \]

Judgement \[ \sigma \stackrel{\text{spec}}{\mapsto} e \Rightarrow w \]

ConstS: \[ \sigma \stackrel{\text{spec}}{\mapsto} \mathcal{C} \Rightarrow [\llbracket \mathcal{C} \rrbracket] \]

VarS: \[ \sigma \stackrel{\text{spec}}{\mapsto} x \Rightarrow \sigma(x) \]

∀i. \[ \sigma \stackrel{\text{spec}}{\mapsto} e_i \Rightarrow v_i \]

\[ \bigotimes(v_1, \ldots, v_n) = v \]

OpS: \[ \sigma \stackrel{\text{spec}}{\mapsto} \mathcal{O}(e_1, \ldots, e_n) \Rightarrow v \]

OrD: \[ \sigma \stackrel{\text{spec}}{\mapsto} \bigotimes(e_1, \ldots, e_n) \Rightarrow \mathcal{O}(e_1, \ldots, e_n) \]

Lift: \[ \sigma \stackrel{\text{spec}}{\mapsto} \text{lift}(\hat{e}) \Rightarrow \llbracket \mathcal{O} \rrbracket^{-1} \]

Judgement \[ \sigma \stackrel{\text{spec}}{\mapsto} s \Rightarrow \langle \sigma', s' \rangle \] \(\bot\)

Skips: \[ \sigma \stackrel{\text{step}}{\mapsto} \text{skip} \Rightarrow \langle \sigma, [\bot] \rangle \]

StepE1: \[ \sigma \stackrel{\text{step}}{\mapsto} \text{skip} \Rightarrow \langle \sigma, [\text{skip}] \rangle \]

Assignments: \[ \sigma \stackrel{\text{step}}{\mapsto} x \Rightarrow \sigma \]

AssignmentsD: \[ \sigma \stackrel{\text{step}}{\mapsto} x \Rightarrow \sigma \]

AssertS: \[ \sigma \stackrel{\text{step}}{\mapsto} \text{assert}(e) \Rightarrow \langle \sigma, [\bot] \rangle \]

AssertS: \[ \sigma \stackrel{\text{step}}{\mapsto} \text{assert}(e) \Rightarrow \langle \sigma, [\bot] \rangle \]

ReplaceS: \[ \sigma \stackrel{\text{step}}{\mapsto} \text{pat} q_2 \Rightarrow \langle \sigma', [\bot] \rangle \]

Replaced: \[ \sigma \stackrel{\text{step}}{\mapsto} q_1 \Leftarrow q_2 \Rightarrow \langle \sigma, [\bot] \rangle \]

Judgement \[ \sigma \stackrel{\text{steps}}{\mapsto} s \Rightarrow \langle \sigma', s' \rangle \] \(\bot\)

StepZero: \[ \sigma \stackrel{\text{steps}}{\mapsto} \rightarrow \Rightarrow \bot \]

StepEl: \[ \sigma \stackrel{\text{steps}}{\mapsto} s \Rightarrow \bot \]

\[ \sigma \stackrel{\text{steps}}{\mapsto} s = \langle \sigma', s' \rangle \]

\[ \sigma \stackrel{\text{steps}}{\mapsto} s \Rightarrow \langle \sigma', s' \rangle \]

StepMany: \[ \sigma \stackrel{\text{steps}}{\mapsto} s \Rightarrow \langle \sigma', s' \rangle \]

\[ \sigma \stackrel{\text{steps}}{\mapsto} s \Rightarrow \langle \sigma', s' \rangle \]

StepE2: \[ \sigma \stackrel{\text{steps}}{\mapsto} s \Rightarrow \bot \]

Figure 5. Specialization for RL2 programs, part 1

A new auxiliary function \(\text{label}(l, \sigma)\) is defined to help extracting the source label from an annotated label.

\[ \text{label}(l, \sigma) = l \quad (3) \]

There is nothing to do except updating the label for the \(\text{from}\) statement. However there are many rules for an eliminable \(\text{if}\) statement, due to the many possibilities. The \(\text{FiT}\) and \(\text{FiF}\) rules cover the case where the control flow is correct and the statement can be reduced into a \(\text{from}\) statement, while \(\text{FiTE}\) and \(\text{FiFE}\) cover the failure case, where it does not match as expected. For the residual \(\text{Fi}\) statement it is not determinable whether the expression holds or not, however the correct label must still be annotated.

Another auxiliary function \(\text{update}\) is defined to only annotate if the labels match.

\[ \text{update}(l, \hat{l}) = \begin{cases} l & \text{if } l = \text{label}(\hat{l}) \\ \hat{l} & \text{otherwise} \end{cases} \quad (4) \]

The annotation to the labels could be computed here because blocks perform reversible updates to the store, but this is less efficient and storing the previous state is convenient.

The jump judgement \(\sigma \stackrel{\text{spec}}{\mapsto} \text{jump}, \hat{j} \Rightarrow \langle [\hat{l}], j \rangle \rangle\) is similar to previous work [20]. It takes an annotated jump statement and gives the residual jump statement and the new states to be specialized. It converts eliminable \(\text{if}\) statements into goto’s. The exit is also annotated with the store, along with the other jumps. This is due to Definition (2), which states that the static output must be preserved.

The block judgement \(\sigma \stackrel{\text{spec}}{\mapsto} \text{block}(\hat{b}, \hat{i}) \Rightarrow \langle [\hat{l}], b, \bot \rangle \rangle\) uses the previously defined judgements to residualize an annotated block. It also annotates the label of the residual program. It can fail, due to both a step or come-from judgements failing.

The main judgement \(\tau (\mathcal{P}, S, R) \Rightarrow (\mathcal{P}’, S’, \mathcal{R}’)\) consists of three sets, the set of states pending to be specialized \(\mathcal{P}\), the set of states already specialized \(S\), and the set of residual blocks \(R\). This is also reminiscent of previous work [20]. In each rule, a state is extracted from \(\mathcal{P}\). If this state is seen before or fails to be specialized, it is ignored (the SpecZero and SpecErr rules). If instead it does specialize to a residual block (the SpecOne rule), then the sets need to be updated.

\(\mathcal{P}\) consists of the state to be specialized as usual, but it also contains the annotated label of the source that lead to the current state. This is needed for correction and annotation purposes in the come-from judgement.

One might imagine that \(S\) only needs to record the current state. However, the annotated label of the previous state has affected the residual block and if the block has a \(\text{Fi}\) statement, one would wish to annotate both labels if possible. We decide the previous label in \(S\) so that if the other branch is encountered, it will be specialized.

This also leads to the auxiliary function \("merge\". The residual block cannot simply be added to \(\mathcal{R}\) since it might have been specialized before. merge is defined such that the residual block is added if it does not exist in \(\mathcal{R}\). If instead it does
exist, this must be a result of a fi statement. One come-from will have the form "fi e from (l₁, σ₁) else l₂" and the other the form "fi e from l₁ else (l₂, σ₂)". These are combined to the desired "fi e from (l₁, σ₁) else (l₂, σ₂)".

4.4 The RL Partial Evaluator

Finally, we can formalize the notion of RL-partial evaluation. Let p = (x₁ → x₁,..., xₙ → x₁,..., z₁ → z₁,..., z₀) → (y₁,..., yₗ) yₗ₊₁,..., yₘ be a well-formed program. Let Δ be a congruent division for p, such that all variables in the set {x₁,..., xₙ} \cup {y₁,..., yₗ} \cup {z₁,..., z₀} \cup \{w₁,..., wₗ\} are static and the remaining are dynamic. Let p₂ be the RL2 program constructed from p and Δ. Let Γ be the block-map for p₂ and l be the label of the entry block of p₂. Let post be the post-processing defined later. We can then define partial evaluation [|p|] as follows:

\[
|p| = \begin{cases} (p_{res}, σ_{out}) & \text{if } \Gamma \langle p, \{} \rangle \Rightarrow^* \langle \{} S, R \rangle \\ \perp & \text{otherwise} \end{cases}
\]

where

\[
σ_{init} = σ \cup \{ y₁ \mapsto nil, \ldots, yₗ \mapsto nil, w₁ \mapsto nil, \ldots, wₗ \mapsto nil \}
\]

\[
P = \{ \langle (l, σ_{init}), (entry, σ_{init}) \rangle \}, \quad \text{post}(R) = (bs, σ_{out})
\]

Ps = (x₁ → x₁,..., xₙ → x₁,..., z₁ → z₁,..., z₀) → (y₁,..., yₗ) yₗ₊₁,..., yₘ with (w₁,..., wₗ) bs

Post-processing. In contrast to Hatcliff’s two partial evaluators, where post-processing is an option, post-processing is a necessity after the specialization phase to obtain well-formed RL residual programs. As a minimum post must take
Partial Evaluation of Reversible Flowchart Programs

5 Evaluation and Experiments

To evaluate the quality of the previous theory, a prototype was created. The prototype is written in Haskell and consists of roughly 2,000 lines of code. It implements all of the previously explained theory as well as some more post-processing for partial evaluation. In addition, it also can invert programs and run them w.r.t. a given input. For simplicity, the prototype handles all possible failures in a similar manner to failures encountered in assertions. However, this has no impact on the quality of the programs reported here. Due to space limitations, the source and residual programs are not shown, but they can be found in the repository.

The prototype’s post-processor uses transition compression, constant folding, and improved error handling in addition to the steps described in Sec. 3.2.1.

5.1 Experimental Results

We now discuss the experimental results. Tab. 1 shows the number of blocks in the source and residual programs, along with the number of statements, including the come-from and jump statements, as well as the steps.

Tab. 2 shows how many operations are done in the source and residual programs on the same input. Performance is measured in the steps and non-exit jumps done during evaluation. Note that each jump also leads to an implicit come-from assertion, and is therefore counted twice in the total.

- **FibPair**: The Fibonacci Pair program computes the \(n\)’th and \(n + 1\)’th Fibonacci number. As the program only has a single input \(n\) the result is fully static when specialized w.r.t. \(n \mapsto 10\). The residual program is then as expected when it consists only of the entry and the exit, as seen in Tab. 1. This means the partial evaluator has done as much work as possible. This differs from previous work, where with the same experiment, 34 lines and 95 residual steps were reported.

- **Partition, Repeat**: The list partition program (Lst. 1) and the repeat program have a very similar structure. Both are specialized w.r.t. \(n \mapsto 3\). This unfolds the loop and results in a single block after transition compression. The steps remain the same despite specialization due new assertions being introduced by the post-processor. In both cases the program has a single block, which is the best one can expect after transition compression. There is room for improvement. As we have seen in Lst. 4, dynamic assertions remain even though they will always succeed. This shows that the removal of redundant assertions is an important task. The development of a simple and effective method for assertion reduction in RL is a task for future work.

- **Encryption**: The symmetric encryption program is ported from Janus and specialized w.r.t. the encryption key. The increase in the residual program size is due to abstraction-based specialization [21].

---

1 https://github.com/Yakose/PERevFlow
Table 1. Size of specialized programs

<table>
<thead>
<tr>
<th>Program</th>
<th>Blocks</th>
<th>Lines</th>
<th>Res. Blocks</th>
<th>Res. Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>FibPair</td>
<td>3</td>
<td>12</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Partition</td>
<td>3</td>
<td>10</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>Repeat</td>
<td>3</td>
<td>11</td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>Encryption</td>
<td>6</td>
<td>33</td>
<td>58</td>
<td>232</td>
</tr>
<tr>
<td>RTM-Cons</td>
<td>24</td>
<td>70</td>
<td>7</td>
<td>27</td>
</tr>
<tr>
<td>RTM-Inc</td>
<td>24</td>
<td>70</td>
<td>22</td>
<td>72</td>
</tr>
<tr>
<td>RTM-Flip</td>
<td>24</td>
<td>70</td>
<td>25</td>
<td>80</td>
</tr>
</tbody>
</table>

Table 2. Speed of specialized programs

<table>
<thead>
<tr>
<th>Program</th>
<th>Steps</th>
<th>Jumps</th>
<th>Total</th>
<th>Res. Steps</th>
<th>Res. Jumps</th>
<th>Res. total</th>
</tr>
</thead>
<tbody>
<tr>
<td>FibPair</td>
<td>33</td>
<td>11</td>
<td>55</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Partition</td>
<td>10</td>
<td>4</td>
<td>18</td>
<td>10</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>Repeat</td>
<td>11</td>
<td>4</td>
<td>19</td>
<td>11</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>Encryption</td>
<td>88</td>
<td>58</td>
<td>204</td>
<td>67</td>
<td>38</td>
<td>143</td>
</tr>
<tr>
<td>RTM-Cons</td>
<td>28</td>
<td>25</td>
<td>78</td>
<td>11</td>
<td>4</td>
<td>19</td>
</tr>
<tr>
<td>RTM-Inc</td>
<td>216</td>
<td>250</td>
<td>716</td>
<td>39</td>
<td>41</td>
<td>121</td>
</tr>
<tr>
<td>RTM-Flip</td>
<td>324</td>
<td>381</td>
<td>1086</td>
<td>49</td>
<td>59</td>
<td>167</td>
</tr>
</tbody>
</table>

Figure 7. Inversion and partial evaluation of an encrypter

Figure 8. Inversion and partial evaluation of an RTM-interpreter w.r.t. an RTM-program for binary increment

5.2 Composing Inversion and Partial Evaluation

One of the outstanding features of reversible languages is that efficient inverse programs can be generated by a syntax-directed program inverter [17], which otherwise is a challenging task (cf. [13, 19, 27]). The program inverter \( J_{RL} \) for RL can be found in Appdx. C. This opens up new ways of composing program transformations, in particular the specialization of inverted programs and the inversion of residual programs.

Experiment 1. An example is the transformation of the symmetric encrypter \( enc \) in Sec. 5.1 into a specialized decrypter \( enc^{-1}_key \). Two ways to achieve this transformation in RL (Fig. 7): (i) by partial evaluation of \( enc \) w.r.t. a \( key \) followed by inversion of \( enc_{key} \) and (ii) by inversion of \( enc \) followed by partial evaluation of \( enc^{-1}_key \) w.r.t. the \( key \). This raises an interesting question: In what order should specialization and inversion be performed to achieve the best result?

The labels of the blocks in the resulting programs differ. This is expected, due to the labeling by the transition compression. Inspecting the two resulting programs shows that their control flow and block bodies are identical!

This experiment shows that Fig. 7 not only holds semantically, which must be the case for correct transformations, but can also hold in terms of the operational quality of the resulting programs. We found that this diagram is commutative for \( enc \), even in practice.

Experiment 2. This opens a new solution to the problem of inverting programs in another reversible language: the partial evaluation of an inverted interpreter for that language.

We use the RTM-interpreter \( RTM \) in Sec. 5.1 for this experiment. It is remarkable that the inversion of an RTM-program into RL can be achieved by specializing the inverted interpreter \( RTM^{-1} \). This is an application of the theory of non-standard semantics [1]. In analogy to the first Futamura projection [7], this performs the first inversion projection [1], which defines program inversion through the specialization of inverse interpreters. This is the first successful application of the first projection in the context of reversible languages.
In a related experiment [14], interpreters were transformed into inverse interpreters by partial evaluation of URA [2].

One wonders whether the results differs depending on the level at which inversion is performed, i.e. whether (i) the RTM-program or (ii) the RTM-interpreter is inverted (Fig. 8).

The programs are very similar (except for the labels), but there are also a few differences in the control flow and block bodies. This results in case (i) doing 71 steps and 79 jumps on an input tape with binary number 100000, while in case (ii) 75 steps and 83 jumps are performed on the same input.

An inspection of the residual programs reveals that this is caused by checking the applicability of the transition rules in different orders. This type of slight volatility depends on the order of the rules that define the RTM-program. This means that with the RTM- interpreter, the results can be affected by the RTM-inverter (e.g., by reordering the transition rules).

With all this in mind, we conclude that the structural quality of (i) and (ii) is the same and their performance is very similar. This means that Fig. 8 (almost) commutes in practice for the RTM-interpreter.

6 Conclusion and Future Work

We developed and formalized a method for partial evaluation in the reversible flowchart language RL and identified the necessary techniques to generate residual programs that are still reversible. We also experimentally evaluated the method using a prototype implementation and confirmed that it gives results comparable to conventional flowchart languages. It is remarkable that the specialization of reversible interpreters for reversible languages is possible, as shown for reversible Turing machines [3]. We conducted the first experiments composing program inversion and partial evaluation of RL confirming in practice the hypothesis that they should commute. This is a desirable quality and a step towards exploring novel combination of transformations [1, 14]. This is the first successful application of the first inversion projection [1, 2] in the context of reversible languages.

The possibility to explore the combined power of partial evaluation and inversion sheds light on new avenues for program transformation that yet wait to be explored in more depth, not only in reversible languages, but also in quantum-based computing and in conventional language paradigms.

The results also show that there is room for improvement. Assertions play a key role in reversible languages [17], but conservatively adding them introduces redundant assertions. A task for future work will be to explore effective methods for optimizing assertion, such as abstraction-based specialization [21], generalized partial computation [8], and assertion removal by an SMT-solver during post-processing [31]. If automatic assertion removal were performed, we would expect the speedups of the residual programs to be larger.

There are injective store updates that are currently excluded as variables are made dynamic due to the uniform BTA. It is a task for future work to investigate possibilities for a more precise BTA of reversible pattern matching (placements). Currently, no such reversibility analysis exists.

Another direction for future work is to investigate the transformation and optimization of reversible programs by interpreter specialization (e.g., [9, 10, 12, 24]).

Acknowledgments

The authors would like to thank the anonymous reviewers for their useful comments.

A Operator Semantics

The semantics of all operators used in the language definition in Sec. 2 can be seen in Fig. 9.

**Binary Operators.**

\[
\text{[[\cdot]]}(v_1, v_2) = (v_1, v_2)
\]

\[
\text{[[\^\cdot]]}(v_1, v_2) = \begin{cases} v_2 & \text{if } v_1 = \text{nil} \\ \text{nil} & \text{if } v_1 = v_2 \neq \text{nil} \end{cases}
\]

\[
\text{[[\^\cdot]]}(v_1, v_2) = \begin{cases} \text{true} & \text{if } v_1 = v_2 \\ \text{nil} & \text{otherwise} \end{cases}
\]

\[
\text{[[\^\cdot]]}(v_1, v_2) = \begin{cases} v_1 & \text{if } v_1 \neq \text{nil} \\ v_2 & \text{otherwise} \end{cases}
\]

\[
\text{[[\&\&]]}(v_1, v_2) = \begin{cases} \text{nil} & \text{if } v_1 = \text{nil} \\ v_2 & \text{otherwise} \end{cases}
\]

\[
\text{[[\^\cdot]]}(i_1, i_2) = \begin{cases} \text{true} & \text{if } i_1 < i_2 \\ \text{nil} & \text{otherwise} \end{cases}
\]

\[
\text{[[\^\cdot]]}(i_1, i_2) = [[\text{<}]](i_2, i_1)
\]

\[
\text{[[\^\cdot]]}(i_1, i_2) = i_1 + i_2 \mod (\text{maxint} +1)
\]

\[
\text{[[\^\cdot]]}(i_1, i_2) = i_1 - i_2 \mod (\text{maxint} +1)
\]

\[
\text{[[\&\&]]}(i_1, i_2) = [i_1 \div i_2] \mod (\text{maxint} +1)
\]

**Unary Operators.**

\[
\text{[[\text{hd}]]}(v_1, v_2) = v_1
\]

\[
\text{[[\text{hd}]]}(v_1, v_2) = v_2
\]

\[
\text{[[\text{hd}]]}(v) = \begin{cases} \text{true} & \text{if } v = \text{nil} \\ \text{nil} & \text{otherwise} \end{cases}
\]

**Figure 9.** Semantics of all RL operators

B RL2 Syntax

The syntax of the two-level language RL2 used by the partial evaluator in Sec. 4 can be seen in Fig. 10.
The functions in Fig. 11 can structurally invert an arbitrary program inverter used for the experiments in Sec. 5.2.

**C Program Inverter for RL**

The functions in Fig. 11 can structurally invert an arbitrary RL program. The rev function reverses a step-list. This is the program inverter used for the experiments in Sec. 5.2.

\[
\begin{align*}
I_{RL}[[x_1] \Rightarrow (x_2')] &\text{ with (} x_2' \text{) } b^*] = (x_2') \Rightarrow (x_1') \text{ with (} x_1' \text{) } I_{block}[[b]]^+ \\
I_{block}[[l; k s' j]] &= l: I_{jump}[[j]] \text{ rev}(I_{step}[[s]])^* I_{from}[[k]] \\
I_{jump}[[goto l]] &= \text{from } l \\
I_{jump}[[if e goto l \text{ else } l_2]] &= \text{if from } l_1 \text{ else } l_2 \\
I_{jump}[[\text{exit}]] &= \text{entry} \\
I_{from}[[k]] &= I_{jump}^{-1}[[k]] \\
I_{step}[[x \oplus e]] &= x \oplus e \text{ where } \oplus = I_{op}[[\oplus]] \\
I_{step}[[q_1 \prec q_2]] &= q_2 \prec q_1 \\
I_{step}[[assert(e)]] &= assert(e) \\
I_{step}[[\text{skip}]] &= \text{skip} \\
I_{op}[[\ast]] &= - \\
I_{op}[[\neg]] &= + \\
I_{op}[[\ast]] &= + \\
\end{align*}
\]

**Figure 10.** Syntax of the two-level language RL2

**Figure 11.** Program inverter \(I_{RL}\) for RL

### References


Received 2023-10-20; accepted 2023-11-20