Reversing Imperative Parallel Programs

James Hoey Irek Ulidowski Shoji Yuen

Department of Informatics
University of Leicester, UK

Graduate School of Information Science
Nagoya University, Japan

jbh11@leicester.ac.uk iu3@leicester.ac.uk yuen@is.nagoya-u.ac.jp

We propose an approach and a subsequent extension for reversing imperative programs. Firstly, we produce both an augmented version and a corresponding inverted version of the original program. Augmentation saves reversal information into an auxiliary data store, maintaining segregation between this and the program state, while never altering the data store in any other way than that of the original program. Inversion uses this information to revert the final program state to the state as it was before execution. We prove that augmentation and inversion work as intended, and illustrate our approach with several examples. We also suggest a modification to our first approach to support non-communicating parallelism. Execution interleaving introduces a number of challenges, each of which our extended approach considers. We define annotation and redefine inversion to use a sequence of statement identifiers, making the interleaving order deterministic in reverse.

1 Introduction

Reverse computation has been an active research area for a number of years. The ability to reverse execute, or invert, a program is desirable due to its potential applications. The relationship with the Landauer principle shows reverse computation to be a feasible solution for producing low power, energy efficient computation [10]. In this paper, we consider reverse computation within the setting of imperative programs. We first propose a state-saving approach for reversing such programs consisting of assignments, conditional statements and while loops. We display an example of this approach showing the execution can now be reversed, and we verify that this reversal is correct. Secondly, we discuss the challenges faced when introducing parallelism, as well as the required modifications to our first approach in order to support it. The formal definition and accompanying example demonstrate the reversal of a parallel program. Finally, we present correctness results for this modified approach.

The most obvious approach to implementing program reversal is to record the entire program state before executing the program. Recording all of the initial variable values does allow immediate reversal to the original state, however suffers several setbacks, including not re-creating the intermediate program states, and the production of garbage data. We propose an approach that records the necessary information to reverse an execution step-by-step, re-creating intermediate steps faithfully allowing movement in both directions at any point. Any information we save as a result of this is used during inversion, meaning no garbage data is produced.

Inspired by the Reverse C Compiler (RCC) [12, 3], our initial approach takes an original program and produces two versions. The first is the *augmented version*, which becomes the program used for forward execution, and has the capability to save all information necessary for inversion, termed *reversal information*. This is implemented via the function *aug* that analyses the original program statement by statement, producing the augmented version. The execution of this version populates a collection of initially empty stacks, termed an auxiliary store δ, with this reversal information. Consider the program shown in Figure 1, producing the `Nth` element of a Fibonacci-like sequence beginning with the values

```
if X > Y then
  Z = Y;
  Y = X;
  X = Z;
else
  skip
end

while N−2 > 0 do
  Z = X;
  X = Y;
  Y += Z;
  N −= 1;
end
```

Figure 1: Original program

```
while pop(δ(W)) do
  N +=1;
  Y −= Z;
  X = pop(δ(X));
  Z = pop(δ(Z));
end

if pop(δ(B)) then
  X = pop(δ(X));
  Y = pop(δ(Y));
  Z = pop(δ(Z));
else
  skip
end
```

Figure 2: Inverted program

of `X` and `Y`. Let the initial state σ consist of `X=4`, `Y=3`, `Z=0`, `N=5` and the initial auxiliary store δ consist of empty stacks. The execution of the augmented version (displayed later in Section 4, Figure 3) under these stores results in the state σ' where `X=11`, `Y=18`, `Z=7`, `N=2` and auxiliary store δ' containing reversal information detailed in Section 4. With this version now being used for forwards execution, it is crucial that the behaviour with respect to the program state is unchanged. Our first result ensures that if the state σ' is produced via the original execution, then it must also be produced via the augmented execution.

The second version, termed the *inverted version*, is generated via the function *inv*. This version follows the inverted execution order of the original, containing a statement corresponding to each of those of the original. Each inverted statement will typically use information from the auxiliary store to revert all of the effects caused via execution of the original. Consider again the example in Figure 1. Application of *inv* to this program produces the inverted version, shown in Figure 2. Execution of this program under the stores σ' and δ' produces the state σ'' where `X=4`, `Y=3`, `Z=0`, `N=5`, and the auxiliary store δ'' containing only empty stacks. Our second result validates that $\sigma'' = \sigma$ and $\delta'' = \delta$, meaning the inversion has happened correctly and the initial program state has been restored. Doing so proves that the augmented program saves the required information, that the inverted program is capable of using this to restore the program state to exactly as it was before, and that augmentation produces no garbage data.

In the second part of the paper, we define a modified approach, this time capable of supporting non-communicating parallelism [9]. Issues introduced such as a non-deterministic execution order make our previous approach insufficient without further state-saving. The *interleaving order*, or order in which the statements are executed, now forms part of the reversal information, captured and stored at runtime. Storing this interleaving order makes the program deterministic in reverse, guaranteeing the execution of the inverted program always follows exactly the inverse execution order of the original. Modifications are made to the process of augmentation from our first approach, with all state-saving implemented at runtime via a set of modified operational semantics for forward execution. A similar reasoning is applied to the process of inversion, resulting in a modified set of operational semantics for reverse execution. These semantics are responsible for using the reversal information, including the interleaving order, to reverse the statements of the original program in exactly the opposite order. Finally the correctness results of our second approach are presented.

The paper is organised as follows. Section 2 introduces the programming language and its notion of program state, with the operational semantics given in Section 3. Section 4 describes the process of augmentation and the information that must be saved, as well as proving the correctness of this augmentation. Section 5 defines the process of inversion and again proves the correctness of this process. Section 6 introduces an updated approach capable of supporting parallel composition, as well as presenting the correctness results.

1.1 Related Work

Program inversion has been discussed for many years, including the work by Gries [8] and by Glück and Kawabe [6, 7]. The Reverse C Compiler as described by Perumalla et al. [12, 3] is one example of a state-saving approach for the reversal of C programs. We relate very closely to this approach, but with differences including that we currently support a smaller language, and we record a while loop sequence in order to avoid modifying the behaviour of the original program (see Section 4). To the best of our knowledge, there is no formal proof of correctness of RCC, and so this is a major focus of our work. Our approach proposes the foundation from which a formally proved approach for a more complex language could emerge. Other work has been produced on reverse computation used within Parallel Discrete Event Simulation (PDES), a simulation methodology capable of executing events speculatively [5, 4]. The backstroke framework [17] and subsequent work on it by Schordan et al. [15, 16] relates slightly less closely to our work as it focuses on this application to PDES. Backstroke is capable of both a state-saving approach and a more advanced, path regeneration method for reverse computation. Other applications include to debugging, with examples being [2, 1]. Similarly to program inversion, the reversible programming language Janus, originally proposed in [11] requires additional information within the source code. Any program written in Janus is fully reversible, without the requirement for any control information to be recorded, but with a requirement for additional assertions that make the program deterministic in both directions [19, 18].

2 Programming Language and Program State

The programming language used for our first approach is similar to any *while language*, particularly that of Hüttel [9]. This consists of destructive and constructive assignments, with the expression not containing the variable in question, or any side effects. Conditional statements and loops are also supported, implemented using both arithmetic and Boolean expressions. Let the set of variables $\mathbb{V}$ be ranged over by $\mathtt{X}$, $\mathtt{Y}$, $\mathtt{Z}\ldots$, the set of integers $\mathbb{Z}$ be ranged over by $\mathtt{l}$, $\mathtt{m}$ and $\mathtt{n}$ and the set of Boolean values $\mathbb{B}$ be $\{\mathtt{T},\mathtt{F}\}$. Also let $\mathtt{Cop}$ be the set of constructive assignment operators $\{\mathtt{+=},\mathtt{-=}\}$ with $\mathtt{cop} \in \mathtt{Cop}$, and $\mathtt{Op}$ be the set of arithmetic operators $\{\mathtt{+},\mathtt{-}\}$ with $\mathtt{op} \in \mathtt{Op}$.

$$\begin{array}{lcl}
\mathtt{P} & ::= & \varepsilon \mid \mathtt{S;\ P} \\
\mathtt{S} & ::= & \mathtt{skip} \mid \mathtt{X\ =\ Exp} \mid \mathtt{X\ Cop\ Exp} \mid \mathtt{if\ B\ then\ P\ else\ P\ end} \mid \\
 & & \mathtt{while\ B\ do\ P\ end} \\
\mathtt{B} & ::= & \mathtt{T} \mid \mathtt{F} \mid \neg\mathtt{B} \mid \mathtt{(B)} \mid \mathtt{Exp\ ==\ Exp} \mid \mathtt{Exp\ >\ Exp} \mid \mathtt{B} \land \mathtt{B} \\
\mathtt{Exp} & ::= & \mathtt{X} \mid \mathtt{n} \mid \mathtt{(Exp)} \mid \mathtt{Exp\ Op\ Exp}
\end{array}$$

$\mathbb{P}$ is the set of programs, ranged over by $\mathtt{P}$, $\mathtt{Q}$ and $\mathtt{R}$. $\mathbb{S}$ is the set of statements, ranged over by $\mathtt{S}$. Expressions $\mathtt{Exp}$ are ranged over by $\mathtt{a}$, $\mathtt{a}_0$, $\mathtt{a}'_0$, $\mathtt{a}_1$, $\mathtt{a}'_1$, Boolean expressions $\mathtt{B}$ are ranged over by $\mathtt{b}$, $\mathtt{b}'$, $\mathtt{b}_0$, $\mathtt{b}_1$ and expressions that can be either are ranged over by $\mathtt{ba}$, $\mathtt{ba}_0$, $\mathtt{ba}'_0$, $\mathtt{ba}_1$, $\mathtt{ba}'_1$.

The program state is represented via a data store σ, responsible for mapping each variable to the value it currently holds. A data store is represented as a set of pairs, with the first element of the pair being the variable name, and the second being its current value. A data store is represented formally as the partial function $\sigma : \mathbb{V} \to \mathbb{Z}$.

Such stores are manipulated using the following notation. Assuming $\mathtt{v} \in \mathbb{Z}$, $\sigma(\mathtt{X})$ returns the value currently associated to the variable `X`, while $\sigma[\mathtt{X} \mapsto \mathtt{v}]$ produces a store identical to σ, but with the variable `X` now holding the value `v`.

Consider the store σ, consisting of two variables `X` and `Y`, with values 3 and 5 respectively, described as $\sigma = \{(\mathtt{X},\ 3), (\mathtt{Y},5)\}$. The statement $\sigma(\mathtt{X})$ returns 3, and $\sigma[\mathtt{X} \mapsto 10]$ results in the store $\{(\mathtt{X},10), (\mathtt{Y},5)\}$.

3 Structured Operational Semantics

This section defines the Structured Operational Semantics (SOS) of the programming language described above. These are defined in the traditional way, following closely with those of Hüttel [9]. The parameter δ, representing the auxiliary store, is not strictly necessary at this point, but is required later and included here for consistency.

3.1 Arithmetic Statements

Let $\mathtt{v} \in \mathbb{Z}$ and recall $\mathtt{op} \in \mathtt{Op}$.

$$\frac{}{(\mathtt{X},\sigma,\delta) \to (\sigma(\mathtt{X}),\sigma,\delta)} \quad \frac{\mathtt{v}\ =\ \mathtt{n\ op\ m}}{(\mathtt{n\ op\ m},\sigma,\delta) \to (\mathtt{v},\sigma,\delta)} \quad \frac{}{(\mathtt{(v)},\sigma,\delta) \to (\mathtt{v},\sigma,\delta)} \quad \frac{(\mathtt{a}_0,\sigma,\delta) \to (\mathtt{a}_0',\sigma',\delta')}{(\mathtt{(a_0)},\sigma,\delta) \to (\mathtt{(a_0')},\sigma',\delta')}$$

$$\frac{(\mathtt{a}_0,\sigma,\delta) \to (\mathtt{a}_0',\sigma',\delta')}{(\mathtt{a}_0\ \mathtt{op}\ \mathtt{a}_1,\sigma,\delta) \to (\mathtt{a}_0'\ \mathtt{op}\ \mathtt{a}_1,\sigma',\delta')} \quad \frac{(\mathtt{a}_1,\sigma,\delta) \to (\mathtt{a}_1',\sigma',\delta')}{(\mathtt{a}_0\ \mathtt{op}\ \mathtt{a}_1,\sigma,\delta) \to (\mathtt{a}_0\ \mathtt{op}\ \mathtt{a}_1',\sigma',\delta')}$$

3.2 Boolean Expressions

Let $\mathtt{bop} \in \{\mathtt{>},\mathtt{==}\}$ if used between two arithmetic expressions or $\mathtt{bop} \in \{\wedge,\mathtt{==}\}$ if used between two Boolean expressions.

$$\frac{}{(\neg\mathtt{T},\sigma,\delta) \to (\mathtt{F},\sigma,\delta)} \quad \frac{}{(\neg\mathtt{F},\sigma,\delta) \to (\mathtt{T},\sigma,\delta)} \quad \frac{(\mathtt{b},\sigma,\delta) \to (\mathtt{b}',\sigma',\delta')}{(\neg\mathtt{b},\sigma,\delta) \to (\neg\mathtt{b}',\sigma',\delta')} \quad \frac{\mathtt{ba}_2 = \mathtt{ba}_0\ \mathtt{bop}\ \mathtt{ba}_1}{(\mathtt{ba}_0\ \mathtt{bop}\ \mathtt{ba}_1,\sigma,\delta) \to (\mathtt{ba}_2,\sigma,\delta)}$$

$$\frac{(\mathtt{ba}_0,\sigma,\delta) \to (\mathtt{ba}_0',\sigma',\delta')}{(\mathtt{ba}_0\ \mathtt{bop}\ \mathtt{ba}_1,\sigma,\delta) \to (\mathtt{ba}_0'\ \mathtt{bop}\ \mathtt{ba}_1,\sigma',\delta')} \quad \frac{(\mathtt{ba}_1,\sigma,\delta) \to (\mathtt{ba}_1',\sigma',\delta')}{(\mathtt{ba}_0\ \mathtt{bop}\ \mathtt{ba}_1,\sigma,\delta) \to (\mathtt{ba}_0\ \mathtt{bop}\ \mathtt{ba}_1',\sigma',\delta')}$$

3.3 Program Statements

Let $\mathtt{v} \in \mathbb{Z}$ and recall $\mathtt{cop} \in \mathtt{Cop}$. Let op be + if cop = +=, otherwise let op be -.

$$[\text{Skip}]\ \frac{}{(\mathtt{skip;P},\sigma,\delta) \to (\mathtt{P},\sigma,\delta)} \qquad [\text{Seq}]\ \frac{(\mathtt{S},\sigma,\delta) \to (\mathtt{S}',\sigma',\delta')}{(\mathtt{S;P},\sigma,\delta) \to (\mathtt{S}'\mathtt{;P},\sigma',\delta')}$$

$$[\text{DA1}]\ \frac{}{(\mathtt{X\ =\ v},\sigma,\delta) \to (\mathtt{skip},\sigma[\mathtt{X} \mapsto \mathtt{v}],\delta)} \qquad [\text{DA2}]\ \frac{(\mathtt{a},\sigma,\delta) \to (\mathtt{a}',\sigma',\delta')}{(\mathtt{X\ =\ a},\sigma,\delta) \to (\mathtt{X\ =\ a}',\sigma',\delta')}$$

$$[\text{CA1}]\ \frac{}{(\mathtt{X\ cop\ v},\sigma,\delta) \to (\mathtt{skip},\sigma[\mathtt{X} \mapsto \sigma(\mathtt{X})\ \mathtt{op}\ \mathtt{v}],\delta)} \qquad [\text{CA2}]\ \frac{(\mathtt{a},\sigma,\delta) \to (\mathtt{a}',\sigma',\delta')}{(\mathtt{X\ cop\ a},\sigma,\delta) \to (\mathtt{X\ cop\ a}',\sigma',\delta')}$$

$$[\mathrm{C1}]\ \frac{}{(\texttt{if T then P else Q end},\sigma,\delta)\rightarrow(\texttt{P},\sigma,\delta)} \qquad [\mathrm{C2}]\ \frac{}{(\texttt{if F then P else Q end},\sigma,\delta)\rightarrow(\texttt{Q},\sigma,\delta)}$$

$$[\mathrm{C3}]\ \frac{(\texttt{b},\sigma,\delta)\rightarrow(\texttt{b}',\sigma',\delta')}{(\texttt{if b then P else Q end},\sigma,\delta)\rightarrow(\texttt{if b}'\texttt{ then P else Q end},\sigma',\delta')}$$

$$[\mathrm{Wh}]\ \frac{}{(\texttt{P},\sigma,\delta)\rightarrow(\texttt{if b then Q;P else skip end},\sigma,\delta)} \qquad \text{where } \texttt{P} = \texttt{while b do Q end}$$

4 Augmentation

The first step of our first approach is to generate the *augmented version* through a process termed augmentation. This process takes each statement of the original program in succession, and returns a semantically equivalent (with respect to the data store) code fragment containing any required state-saving operations. These fragments are then combined to produce the augmented version.

The information required to be saved depends on the type of statement. Destructive assignments discard the old value of a variable, meaning it must be saved. Constructive assignments do not suffer this problem meaning they are reversible without state-saving. Due to no guarantee that a condition is invariant, conditional statements must save control information indicating which branch was executed. While loops not having a fixed number of iterations means the number of times the loop should be inverted is unknown. Therefore a sequence of Booleans representing the while loop is saved.

Saving the result of evaluating conditional statements and while loops removes the burden of re-evaluating these expressions during inversion, unlike the reversible programming language Janus that does require this. In an effort to ensure that the state-saving does not affect the behaviour of the program (w.r.t. the data store), all reversal information is stored separately in an auxiliary data store.

4.1 Auxiliary Data Store

Recall that $\mathbb{V}$ is the set of program variable names, and now let both `B` and `W` be reserved keywords that cannot appear within this set. An auxiliary data store δ is a set of stacks, consisting of one self-named stack for each program variable within $\mathbb{V}$, one stack `B` for all conditional statements and one stack `W` for all while loops. More formally, $\delta : (\mathbb{V} \rightarrow \mathbb{X}) \cup (\{\texttt{B},\texttt{W}\} \cup \mathbb{B}')$, where $\mathbb{X}$ is the set of stacks of integers and $\mathbb{B}'$ is the set of stacks of Booleans. Auxiliary stores will be represented as a set of pairs. Each pair represents a stack, with the first element being the stack name and the second element being the sequence of its elements. The order of this sequence reflects that of the stack, with the left-most element being the head of the stack. Consider a program consisting of one variable `X` (initially `1`) destructively assigned twice (to `3` and `5`), one conditional statement that evaluates to `T` and a while loop with one iteration. The final auxiliary store would be $\{(\texttt{X},\{\texttt{3,1}\}), (\texttt{B},\{\texttt{T}\}), (\texttt{W},\{\texttt{T,F}\})\}$.

The stacks on δ are manipulated in the traditional manner [9], using `push` and `pop` operations introduced via augmentation. The notation $\delta[\texttt{v} \mapsto \texttt{X}]$ and `push(v,X)` both represent pushing the value `v` to the stack `X`, while $\delta[\texttt{X}]$ and `pop(X)` represent popping the stack `X`. Further notation includes $\delta(\texttt{S})$ that returns the stack named `S`, `v:S` that indicates a stack with head `v` and tail `S`, and $\delta[\texttt{X}/\texttt{X}']$ that states the stack `X` is replaced by `X'`. The SOS rules are defined, where $\texttt{v} \in \mathbb{Z} \cup \mathbb{B}$.

$$[\mathrm{Pop}]\ \frac{\delta(\texttt{X})\ \texttt{=}\ \texttt{v:X}'}{(\texttt{pop}(\delta(\texttt{X})),\sigma,\delta)\rightarrow(\texttt{v},\sigma,\delta[\texttt{X}/\texttt{X}'])} \qquad [\mathrm{Push1}]\ \frac{}{(\texttt{push(v,}\delta(\texttt{X})\texttt{)},\sigma,\delta)\rightarrow(\texttt{skip},\sigma,\delta[\texttt{v}\mapsto\texttt{X}])}$$

$$\text{[Push2]}\quad \frac{(\mathtt{ba},\sigma,\delta)\to(\mathtt{ba}',\sigma',\delta')}{(\mathtt{push(ba,}\delta(\mathtt{X})\mathtt{)},\sigma,\delta)\to(\mathtt{push(ba}'\mathtt{,}\delta(\mathtt{X})\mathtt{)},\sigma',\delta')}$$

We are now ready to introduce the function that performs the augmentation.

4.2 Augmentation Function

Let $\hat{\mathbb{P}}$ be the set of augmented programs. The function $aug : \mathbb{P} \to \hat{\mathbb{P}}$ takes the original program and recursively applies the function $a : \mathbb{S} \to \hat{\mathbb{P}}$ to each statement, producing its augmented version.

Destructive assignments are augmented into two statements, one to push the old value of the variable to its self-named stack on δ, and a second to perform the assignment (see 4). Constructive assignments are left unchanged due to their reversibility (see 5). Conditional statements have each branch recursively augmented, as well as extended with an operation that stores a Boolean indicating whether the true or false branch was executed (see 6). As such, *aug* and *a* are now defined, where `cop` $\in$ `Cop`.

$$aug(\varepsilon) = \varepsilon \tag{1}$$

$$aug(\mathtt{S;P}) = a(\mathtt{S});\ aug(\mathtt{P}) \tag{2}$$

$$a(\mathtt{skip}) = \mathtt{skip} \tag{3}$$

$$a(\mathtt{X\ =\ a}) = \mathtt{push(}\sigma(\mathtt{X}),\delta(\mathtt{X})\mathtt{);\ X\ =\ a} \tag{4}$$

$$a(\mathtt{X\ cop\ a}) = \mathtt{X\ cop\ a} \tag{5}$$

$$\begin{aligned} a(\mathtt{if\ b\ then\ P\ else\ Q\ end}) = {} & \mathtt{if\ b\ then}\ aug(\mathtt{P});\ \mathtt{push(T,}\delta(\mathtt{B})\mathtt{)} \\ & \quad \mathtt{else}\ aug(\mathtt{Q});\ \mathtt{push(F,}\delta(\mathtt{B})\mathtt{)\ end} \end{aligned} \tag{6}$$

The traditional approach of handling while loops by initialising a counter and incrementing it for each iteration is not used here due to its adverse effects on the behaviour of the program w.r.t. the data store. While loops are instead augmented to save a sequence of Booleans representing its execution. Generating the sequence in the intuitive way (of a `T` for each iteration and finally an `F`) and storing this onto a traditional stack will require the sequence to be manipulated before being used. Such manipulation is both difficult, due to ambiguities within such sequences, and avoidable, by storing a usable order to begin with.

The desired order is that of the intuitive approach, but with any opening `T` switched with its corresponding closing `F`, while maintaining any nested `T` elements. This sequence can be generated provided we can distinguish between the first iteration of a loop and any other. The first iteration now requires an `F`, while any subsequent iteration (including the unsuccessful last iteration) requires a `T` (see 7).

$$\begin{aligned} a(\mathtt{while\ b\ do\ P\ end}) = {} & \mathtt{if\ b\ then} \\ & \quad \mathtt{push(F,}\delta(\mathtt{W})\mathtt{);}\ aug(\mathtt{P}); \\ & \quad \mathtt{while\ b\ do} \\ & \quad\quad \mathtt{push(T,}\delta(\mathtt{W})\mathtt{);}\ aug(\mathtt{P}) \\ & \quad \mathtt{end;\ push(T,}\delta(\mathtt{W})\mathtt{)} \\ & \mathtt{else\ push(F,}\delta(\mathtt{W})\mathtt{)\ end} \end{aligned} \tag{7}$$

We now return to our example discussing Figure 1. The augmented version of this program is shown in Figure 3. The destructive assignment of `Z` on line 2 of Figure 1 corresponds to line 2 of Figure 3,

```
 1 if X > Y then
 2   push(σ(Z),δ(Z)); Z = Y;
 3   push(σ(Y),δ(Y)); Y = X;
 4   push(σ(X),δ(X)); X = Z;
 5   push(T,δ(B))
 6 else
 7   skip;push(F,δ(B))
 8 end
 9 if N−2 > 0 then
10   push(F,δ(W));
11   push(σ(Z),δ(Z)); Z = X;
12   push(σ(X),δ(X)); X = Y;
13   Y += Z;
14   N −= 1;
15   while N−2 > 0 do
16     push(T,δ(W));
17     push(σ(Z),δ(Z)); Z = X;
18     push(σ(X),δ(X)); X = Y;
19     Y += Z;
20     N −= 1
21   end
22   push(T,δ(W))
23 else
24   push(F,δ(W)); end
```

Figure 3: Augmented Version of the Program in Figure 1

where the push statement is used to first save the old value. Lines 5 and 7 of Figure 3 contain inserted operations to save the result of evaluating the conditional statement, while lines 10, 16, 22 and 24 are inserted commands to save the sequence of Boolean values representing the execution of the while loop. Execution of this program under the initial stores $\sigma = \{(\texttt{X},4), (\texttt{Y},3), (\texttt{Z},0), (\texttt{N},5)\}$ and $\delta = \{(\texttt{X},\{\}), (\texttt{Y},\{\}), (\texttt{Z},\{\}), (\texttt{N},\{\}), (\texttt{B},\{\}), (\texttt{W},\{\})\}$, produces the final stores $\sigma' = \{(\texttt{X},11), (\texttt{Y},18), (\texttt{Z},7), (\texttt{N},2)\}$ and $\delta' = \{(\texttt{X},\{7,4,3,4\}), (\texttt{Y},\{3\}), (\texttt{Z},\{4,3,3,0\}), (\texttt{N},\{\}), (\texttt{B},\{\texttt{T}\}), (\texttt{W},\{\texttt{T},\texttt{T},\texttt{T},\texttt{F}\})\}$. The two final stores now contain all of the necessary information for reversal.

We are now ready to state our first result. Firstly, Proposition 1 states that if the execution of an original program terminates, then the execution of the augmented version of that program also terminates (where a program terminates if its execution finishes with the configuration $(\texttt{skip},\sigma^*,\delta^*)$ for some σ^* and δ^*). Secondly, Proposition 1 states that augmentation produces an augmented version that modifies the data store σ in exactly the same way as that of the original program to σ', while also populating the auxiliary store δ with reversal information producing δ'.

Proposition 1. *Let* P *be a program that does not interact with the auxiliary store, σ be an arbitrary initial data store and δ be an arbitrary initial auxiliary data store. Firstly, if* $(\texttt{P},\sigma,\delta) \rightarrow^* (\texttt{skip},\sigma',\delta'')$*, for some σ' and δ'', then* $(aug(\texttt{P}),\sigma,\delta) \rightarrow^* (\texttt{skip},\sigma'',\delta''')$ *for some σ'' and δ'''. Secondly, if* $(\texttt{P},\sigma,\delta) \rightarrow^* (\texttt{skip},\sigma',\delta)$*, for some σ', then* $(aug(\texttt{P}),\sigma,\delta) \rightarrow^* (\texttt{skip},\sigma',\delta')$ *for some δ'.*

We note that the inverse implication, namely that if $(aug(\texttt{P}),\sigma,\delta) \rightarrow^* (\texttt{skip},\sigma',\delta')$, for some σ' and δ', then $(\texttt{P},\sigma,\delta) \rightarrow^* (\texttt{skip},\sigma',\delta)$, would also be valid. However we defer this proof to future work, and now return to proving the second part of Proposition 1 (with the first following correspondingly).

Proof. By induction on the length of the sequence $(\texttt{P},\ \sigma,\delta) \rightarrow^* (\texttt{skip},\ \sigma',\delta)$. Since there are no transitions of length 0, the proposition holds vacuously. Assume that the proposition holds for programs R, stores σ^* and auxiliary stores δ^*, such that $(\texttt{R},\sigma^*,\delta^*) \rightarrow^* (\texttt{skip},\sigma_1^*,\delta^*)$ is shorter than $(\texttt{P},\sigma,\delta) \rightarrow^* (\texttt{skip},\sigma',\delta)$. Further assuming P is of the form S;P′ such that S is a statement and P′ is the remaining program, we have that

$$(\texttt{S;P}',\sigma,\delta) \rightarrow^* (\texttt{skip},\sigma',\delta)$$

for some σ'. Through use of the SOS rules Seq and Skip, we have

$$(\texttt{S;P}',\sigma,\delta) \rightarrow^* (\texttt{skip;P}',\sigma'',\delta) \rightarrow (\texttt{P}',\sigma'',\delta) \rightarrow^* (\texttt{skip},\sigma',\delta)$$

for some σ''. With this in mind, we need to show that $(aug(\mathtt{S;P'}),\sigma,\delta) \rightarrow^* (\mathtt{skip},\sigma',\delta')$ for some δ'.

By the definition of *aug*, clause (2) we have $(aug(\mathtt{S;P'}),\sigma,\delta) = (a(\mathtt{S});aug(\mathtt{P'}),\sigma,\delta)$, meaning it is sufficient to prove

$$(a(\mathtt{S});aug(\mathtt{P'}),\sigma,\delta) \rightarrow^* (\mathtt{skip};aug(\mathtt{P'}),\sigma'',\delta'') \rightarrow (aug(\mathtt{P'}),\sigma'',\delta'') \rightarrow^* (\mathtt{skip},\sigma',\delta')$$

for some σ'', δ'' and δ'. Since $(\mathtt{S;P'},\sigma,\delta) \rightarrow^* (\mathtt{P'},\sigma'',\delta)$, then repeated use of the Seq rule (from conclusion to premises) produces $(\mathtt{S},\sigma,\delta) \rightarrow^* (\mathtt{skip},\sigma'',\delta)$. Now assume $a(\mathtt{S}) = \mathtt{P}_S$ for each type of statement S. Then by Lemma 1 below, we have that $(\mathtt{P}_S,\sigma,\delta) \rightarrow^* (\mathtt{skip},\sigma'',\delta'')$ for some δ''. Using the Seq rule (from premises to conclusion) we obtain

$$(\mathtt{P}_S;aug(\mathtt{P'}),\sigma,\delta) \rightarrow^* (\mathtt{skip};aug(\mathtt{P'}),\sigma'',\delta'')$$

Then by the Skip rule, we get $(\mathtt{skip};aug(\mathtt{P'}),\sigma'',\delta'') \rightarrow (aug(\mathtt{P'}),\sigma'',\delta'')$. The induction hypothesis on $(\mathtt{P'},\sigma'',\delta'')$ gives us

$$(aug(\mathtt{P'}),\sigma'',\delta'') \rightarrow^* (\mathtt{skip},\sigma',\delta')$$

for some δ'. Therefore we have obtained $(a(\mathtt{S});aug(\mathtt{P'}),\sigma,\delta) \rightarrow^* (aug(\mathtt{P'}),\sigma'',\delta'') \rightarrow^* (\mathtt{skip},\sigma',\delta')$, meaning $(aug(\mathtt{S;P'}),\sigma,\delta) \rightarrow^* (\mathtt{skip},\sigma',\delta')$ holds as required. Therefore the proposition holds, provided the following lemma holds. □

Lemma 1. *Let S be a statement that does not interact with the auxiliary store, σ be an initial data store and δ be an initial auxiliary data store. If $(\mathtt{S},\sigma,\delta) \rightarrow^* (\mathtt{skip},\sigma',\delta)$ for some σ' then $(a(\mathtt{S}),\sigma,\delta) \rightarrow^* (\mathtt{skip},\sigma',\delta')$ for some δ'.*

Proof. We consider each type of statement S in turn. Due to space constraints, we only include one case, with the other cases following similarly. The notation $(\mathtt{Q},\sigma,\delta) \xrightarrow[X]{}^l (\mathtt{Q'},\sigma^\dagger,\delta^\dagger)$ denotes l transitions by the SOS rule X produces the program Q′, store $\sigma^\dagger$ and auxiliary store $\delta^\dagger$.

Case 1.1. Consider statement X=a and its execution under the initial stores σ and δ where X is initially v′ and a evaluates to v in l steps such that

$$(\mathtt{X=a},\sigma,\delta) \xrightarrow[\text{DA2}]{}^l (\mathtt{X=v},\sigma,\delta) \xrightarrow[\text{DA1}]{} (\mathtt{skip},\sigma[\mathtt{X} \mapsto \mathtt{v}],\delta).$$

Recall that $\sigma(\mathtt{X}) = \mathtt{v'}$. The execution of the augmented version of X=a is

$$\begin{aligned}(\mathtt{push(v',}\delta\mathtt{(X));X=a},\sigma,\delta) &\xrightarrow[\text{Push}_1,\text{ Skip}]{} (\mathtt{X=a},\sigma,\delta[\mathtt{v'} \mapsto \mathtt{X}]) \\ \xrightarrow[\text{DA2}]{}^l (\mathtt{X=v},\sigma,\delta[\mathtt{v'} \mapsto \mathtt{X}]) &\xrightarrow[\text{DA1}]{} (\mathtt{skip},\sigma[\mathtt{X} \mapsto \mathtt{v}],\delta[\mathtt{v'} \mapsto \mathtt{X}])\end{aligned}$$

As such, this case holds with $\sigma' = \sigma[\mathtt{X} \mapsto \mathtt{v}]$ and $\delta' = \delta[\mathtt{v'} \mapsto \mathtt{X}]$.

With all other cases following in a similar manner, Lemma 1 holds. □

5 Inversion

The second step of our initial approach is to generate the inverted version through a process named inversion. Inversion takes each statement in reverse order, generates the code fragment necessary to undo its effects, before combining these fragments to generate the inverted version. The majority of the returned code fragments will use the reversal information on the auxiliary store, meaning the augmented version must be executed prior to the execution of this version.

Destructive assignments are replaced with another destructive assignment to the same variable, but this time assigning the value currently at the top of the self-named stack (see 11). Constructive assignments require no reversal information, and can simply be replaced by their inverse (see 12). Conditional statements saved a Boolean indicating which branch was executed, meaning the retrieval and evaluation of this now replaces the original condition, along with the recursive inversion of the branches (see 13). While loops saved a sequence of Booleans in the desired order, meaning the while loop can continually iterate until the top of the stack `W` is no longer true (see 14), along with the recursive inversion of the body. As mentioned earlier, the reverse execution of conditionals and loops does not require their conditions to be re-evaluated, increasing efficiency.

Let $\mathbb{P}^{-1}$ be the set of inverted programs. The function $inv : \mathbb{P} \to \mathbb{P}^{-1}$ takes the original program and recursively applies the function $i : \mathbb{S} \to \mathbb{P}^{-1}$ to each statement in reverse order, producing its inverted version. We now define *inv* and *i*, where `cop` $\in$ `Cop`, and `icop` = `+=` if `cop` = `-=`, and `-=` otherwise.

$$inv(\varepsilon) = \varepsilon \tag{8}$$

$$inv(\texttt{S;P}) = inv(\texttt{P});\ i(\texttt{S}) \tag{9}$$

$$i(\texttt{skip}) = \texttt{skip} \tag{10}$$

$$i(\texttt{X = a}) = \texttt{X = pop(}\delta\texttt{(X))} \tag{11}$$

$$i(\texttt{X cop a}) = \texttt{X icop a} \tag{12}$$

$$i(\texttt{if b then P else Q end}) = \texttt{if pop(}\delta\texttt{(B)) then } inv(\texttt{P}) \texttt{ else } inv(\texttt{Q}) \texttt{ end} \tag{13}$$

$$i(\texttt{while b do P end}) = \texttt{while pop(}\delta\texttt{(W)) do } inv(\texttt{P}) \texttt{ end} \tag{14}$$

We now return to our example code shown in Figure 1. Applying the function *inv* to this program produces the inverted program shown in Figure 2. The overall program order has been inverted, with the while loop now being executed first. An example destructive assignment is on line 2 of Figure 1, and is inverted via the line 11 of Figure 2. Execution of this version under the final stores σ' = {(X,11), (Y,18), (Z,7), (N,2)} and δ' = {(X,{7,4,3,4}), (Y,{3}), (Z,{4,3,3,0}), (N,{}), (B,{T}), (W,{T,T,T,F})}, produces the initial stores σ'' = {(X,4), (Y,3), (Z,0), (N,5)} and δ'' = {(X,{}), (Y,{}), (Z,{}), (N,{}), (B,{}), (W,{})}. As should be clear, $\sigma'' = \sigma$ and $\delta'' = \delta$, meaning the reversal has executed successfully.

We will now present our second result for $(\texttt{P}, \sigma, \delta)$. Recall that by Proposition 1, the execution of the augmented version of `P` produces the modified auxiliary store δ', which plays a crucial role in Proposition 2 below. Firstly, Proposition 2 states that if the original program `P` terminates on σ and δ, producing σ', then the execution of the inverted version of `P` terminates on σ' and the modified auxiliary store δ'. Secondly, Proposition 2 states that given the final stores σ' and δ' produced via execution of the augmented version of `P`, executing the corresponding inverted version on these stores restores the initial state, namely σ and δ.

Proposition 2. *Let* `P` *be a program that does not interact with the auxiliary store,* σ *be an arbitrary initial data store and* δ *be an arbitrary initial auxiliary data store. Firstly, if* $(\texttt{P}, \sigma, \delta) \to^* (\texttt{skip}, \sigma', \delta'')$*, for some* σ' *and* δ''*, then* $(inv(\texttt{P}), \sigma', \delta') \to^* (\texttt{skip}, \sigma'', \delta''')$*, for some* σ'' *and* δ'''*. Secondly, if* $(\texttt{P}, \sigma, \delta) \to^* (\texttt{skip}, \sigma', \delta)$*, for some* σ'*, then* $(inv(\texttt{P}), \sigma', \delta') \to^* (\texttt{skip}, \sigma, \delta)$*, for some* δ'*.*

We note that the first result in Proposition 2 would be valid, but postpone the proof to future work and now return to proving the second part of Proposition 2.

Proof. By induction on the length of the sequence $(\texttt{P}, \sigma, \delta) \rightarrow^* (\texttt{skip}, \sigma', \delta)$. Since there are no transitions of length 0, the proposition holds vacuously. Assume that the proposition holds for programs $\texttt{R}$, stores σ^* and auxiliary stores δ^*, such that $(\texttt{R}, \sigma^*, \delta^*) \rightarrow^* (\texttt{skip}, \sigma_1^*, \delta^*)$ is shorter than $(\texttt{P}, \sigma, \delta) \rightarrow^* (\texttt{skip}, \sigma', \delta)$. Further assume $\texttt{P}$ is of the form $\texttt{S;P}'$ such that $\texttt{S}$ is a statement and $\texttt{P}'$ is the remaining program. Let $(\texttt{S;P}', \sigma, \delta) \rightarrow^* (\texttt{skip}, \sigma', \delta)$ for some σ'. This means that

$$(\texttt{S;P}', \sigma, \delta) \rightarrow^* (\texttt{skip;P}', \sigma'', \delta) \rightarrow (\texttt{P}', \sigma'', \delta) \rightarrow^* (\texttt{skip}, \sigma', \delta)$$

for some σ''. By applying the Seq rule (conclusion to premises) to $(\texttt{S;P}', \sigma, \delta) \rightarrow^* (\texttt{skip;P}', \sigma'', \delta)$, we obtain $(\texttt{S}, \sigma, \delta) \rightarrow^* (\texttt{skip}, \sigma'', \delta)$. By the definition of *aug*, clause (2) we have $(aug(\texttt{S;P}'), \sigma, \delta) = (a(\texttt{S}); aug(\texttt{P}'), \sigma, \delta)$ and by Proposition 1 we have

$$(a(\texttt{S}); aug(\texttt{P}'), \sigma, \delta) \rightarrow^* (\texttt{skip}; aug(\texttt{P}'), \sigma'', \delta'') \rightarrow (aug(\texttt{P}'), \sigma'', \delta'') \rightarrow^* (\texttt{skip}, \sigma', \delta')$$

for some δ'', δ'. Using Seq (conclusion to premise) on $(a(\texttt{S}); aug(\texttt{P}'), \sigma, \delta) \rightarrow^* (\texttt{skip}; aug(\texttt{P}'), \sigma'', \delta'')$ we obtain $(a(\texttt{S}), \sigma, \delta) \rightarrow^* (\texttt{skip}, \sigma'', \delta'')$.

We need to show that given σ' and δ', $(inv(\texttt{S;P}'), \sigma', \delta') \rightarrow^* (\texttt{skip}, \sigma, \delta)$. By the definition of *inv*, clause 9, we have $inv(\texttt{S;P}') = inv(\texttt{P}'); i(\texttt{S})$, meaning we shall show

$$(inv(\texttt{P}'); i(\texttt{S}), \sigma', \delta') \rightarrow^* (i(\texttt{S}), \sigma^\dagger, \delta^\dagger) \rightarrow^* (\texttt{skip}, \sigma, \delta)$$

for some $\sigma^\dagger$ and $\delta^\dagger$. The induction hypothesis for $(\texttt{P}', \sigma'', \delta'') \rightarrow^* (\texttt{skip}, \sigma', \delta'')$, where δ'' is obtained by augmentation of $\texttt{S}$ on δ, gives us $(inv(\texttt{P}'), \sigma', \delta') \rightarrow^* (\texttt{skip}, \sigma'', \delta'')$ where δ' is obtained by augmentation of $\texttt{P}'$ on σ'' and δ'' as shown by $(aug(\texttt{P}), \sigma'', \delta'') \rightarrow^* (\texttt{skip}, \sigma', \delta')$ above. Using the rule Seq (premise to conclusion) repeatedly we get

$$(inv(\texttt{P}'); i(\texttt{S}), \sigma', \delta') \rightarrow^* (\texttt{skip}; i(\texttt{S}), \sigma'', \delta'') \rightarrow (i(\texttt{S}), \sigma'', \delta'')$$

Therefore $\sigma^\dagger = \sigma''$ and $\delta^\dagger = \delta''$. All that remains now is to prove $(i(\texttt{S}), \sigma'', \delta'') \rightarrow^* (\texttt{skip}, \sigma, \delta)$, which is done in Lemma 2 below. □

Lemma 2. *Let* S *be a statement that does not interact with the auxiliary store,* σ *be an arbitrary initial data store and* δ *be an arbitrary initial auxiliary data store. Then if* $(\texttt{S}, \sigma, \delta) \rightarrow^* (\texttt{skip}, \sigma', \delta)$ *for some* σ'*, then* $(i(\texttt{S}), \sigma', \delta') \rightarrow^* (\texttt{skip}, \sigma, \delta)$ *for some* δ'*.*

Proof. We consider each type of statement S in turn. Due to space constraints, we only include one case, with the other cases following similarly.

Case 2.1. Consider statement X=a and its execution under the initial stores σ and δ where X is initially $\texttt{v}'$ and a evaluates to v in l steps such that

$$(\texttt{X=a}, \sigma, \delta) \xrightarrow[\text{DA2}]{}^{l} (\texttt{X=v}, \sigma, \delta) \xrightarrow[\text{DA1}]{} (\texttt{skip}, \sigma[\texttt{X} \mapsto \texttt{v}], \delta)$$

Then by Proposition 1 and Lemma 1 Case 1.1, we have that $(a(\texttt{X=a}),\sigma,\delta) \rightarrow \ldots \rightarrow (\texttt{skip},\sigma',\delta')$, such that $\sigma' = \sigma[\texttt{X} \mapsto \texttt{v}]$ and $\delta' = \delta[\texttt{v}' \mapsto \texttt{X}]$. Then the execution of the inverted version of X=a is

$$(i(\texttt{X=a}), \sigma', \delta') \xrightarrow[\text{Pop}]{} (\texttt{X=v}', \sigma', \delta'[\texttt{X}]) \xrightarrow[\text{DA1}]{} (\texttt{skip}, \sigma'[\texttt{X} \mapsto \texttt{v}']), \delta'[\texttt{X}])$$

such that $\sigma'[\texttt{X} \mapsto \texttt{v}'] = \sigma$ since $\texttt{v}'$ is equal to the initial value of X retrieved by the pop operation, and $\delta'[\texttt{X}] = \delta$. Therefore the stores have been restored to their initial states, as required, meaning the case holds.

With all other cases following in a similar manner, the Lemma is proved to be correct. □

```
X+=Y+2 par (Y=X+2; X=4)
```

Figure 4: Original program

```
X+=Y+2[] par (Y=X+2[]; X=4[])
```

Figure 5: Annotated program

6 Adding Parallelism

We will now modifiy our first approach to support non-communicating parallelism [9], also referred to as *interleaving*, where the execution of two (or more) programs are interleaved while each individually maintains program order. To the best of our knowledge, RCC does not support parallelism in any form. Due to space constraints, we restrict the language to assignments and parallelism only. Conditionals and loops can be modified in a similar way and so are omitted. Let us reuse previous notation such that $\mathbb{P}$ is now the set of programs of this restricted language, $\hat{\mathbb{P}}$ is now the set of annotated programs, $\mathbb{P}^{-1}$ is now the set of inverted programs and $\mathbb{S}$ is now the set of statements of this restricted language. The definition of a statement becomes

$$\mathrm{S} ::= \texttt{skip} \mid \texttt{X = Exp} \mid \texttt{X Cop Exp} \mid \texttt{P par P}$$

6.1 Challenges

Supporting parallelism introduces three challenges. Execution of parallel programs results in a non-deterministic execution order. Our first approach works as the programs are sequential, allowing the inverted program to follow the *inverted program order*. However there may be different execution orders of the same parallel program due to interleaving. So, without care, programs can be executed forwards under one interleaving and reversed under another, which is clearly incorrect. Consider the program in Figure 4 represented via (`P1 par (Q1;Q2)`), where the three possible execution interleavings are (`P1;Q1;Q2`), (`Q1;P1;Q2`) or (`Q1;Q2;P1`). Imagine the program here is executed under the first interleaving with the initial data store σ where `X=1` and `Y=1`, resulting in the final state σ' where `X=4` and `Y=6`. Without further information, inversion may assume the third interleaving was executed and so inverts the statements in the order (`P1;Q2;Q1`), clearly producing the incorrect final state where `X=4` and `Y=1`. We therefore will require both the auxiliary store δ and the inverse program as before, as well as the order in which the statements were executed forwards, termed *interleaving order*.

Another challenge is the *atomicity of statements*. Execution of a statement typically takes several steps to complete, increasing the possible execution paths and likelihood of races. Consider Figure 4 with no assumption of atomicity and initial state σ as above. Imagine `Y` is first evaluated in `P1`, then all of `Q1` and `Q2` are executed, before `P1` finally completes. This leads to the final state σ' where `X=7` and `Y=3`, values not reachable when assuming statement atomicity.

Finally, `push` operations inserted in our first approach relate to a specific statement and these must be executed atomically. Consider the program `X+=1 par X=5` augmented via our first approach into `X+=1 par push(`σ`(X),`δ`(X)); X=5`, with no such atomicity and `X` initially `1`. Assume that the `push` statement executes first, storing the value `1` onto δ`(X)`. The constructive assignment is then executed, incrementing `X` by one to `2`. Finally the destructive assignment is executed, updating `X` to `5`. The inverted version of this program is `X-=1 par X=pop(`δ`(X)`. Reversing the same interleaving first inverts the destructive assignment, assigning the value `1` from δ`(X)` to the variable `X`. The constructive assignment is then inverted, decrementing `X` by one to `0`. Clearly, this reversal has been unsuccessful.

6.2 Overcoming these Challenges

We update our approach to now capture the interleaving order. An identifier, or element of the set of natural numbers used in ascending order, is associated with each statement, each time it is executed, and stored onto a stack within the source code, very much like the *communication keys* of CCSK [13, 14]. These identifiers index any reversal information stored, and are used to direct the execution of the inverted version. This makes the execution order deterministic, thus removing the first challenge.

The updated approach will not introduce `push` statements in order to avoid issues relating to statement atomicity. A combination of this, and the fact that the interleaving order is not determined until runtime, mean all state-saving will be deferred to the operational semantics. A separate set of operational semantics are defined for forward execution. As such, inversion will no longer introduce `pop` statements, with all interaction with the reversal information being deferred to another separate set of operational semantics. In future work, we will add support for conditionals and loops, and further extend this with locks and mutual exclusion to allow this approach to be implemented within the language syntax.

We make the assumption of the atomicity of statements, restricting all interleaving to statement level, though this will be removed in future work.

6.3 Annotation and Forward Execution

The process of augmentation is replaced with *annotation*. This takes the original program, appends the necessary stacks into the program statements' source code, before returning the annotated version. Each statement is associated with a stack, necessary for programs containing loops as each statement may be executed multiple times requiring multiple identifiers. Stacks are not strictly necessary for our restricted language, however will be vital when we introduce conditionals and loops and so are included here for continuity. Each of these stacks is initially empty, and named uniquely via the function `nextS`. Let $\mathbb{S}'$ be the set of annotated statements. The function $ann : \mathbb{P} \to \hat{\mathbb{P}}$ takes the original program and recursively applies the re-defined function $a : \mathbb{S} \to \mathbb{S}'$ to each statement, producing the annotated version, where `e` is an arithmetic expression.

$$\begin{array}{rlrl} ann(\varepsilon) = & \varepsilon\ \mathtt{A} & ann(\mathtt{S;P}) = & a(\mathtt{S}); ann(\mathtt{P}) \\ a(\mathtt{skip}) = & \mathtt{skip\ A} & a(\mathtt{X\ =\ e}) = & \mathtt{X\ =\ e\ A} \\ a(\mathtt{X\ cop\ e}) = & \mathtt{X\ cop\ e\ A} & a(\mathtt{P\ par\ Q}) = & ann(\mathtt{P})\ \mathtt{par}\ ann(\mathtt{Q}) \end{array}$$

At this point, we have introduced a new syntactic category for annotated programs. Annotated programs and statements are defined below, with arithmetic and Boolean expressions as in Section 2. The sets $\mathbb{P}$ and $\mathbb{S}$ are also extended accordingly.

$$\begin{array}{l} \mathtt{AP} ::= \varepsilon\ \mathtt{A} \mid \mathtt{AS;\ AP} \\ \mathtt{AS} ::= \mathtt{skip\ A} \mid \mathtt{X\ =\ Exp\ A} \mid \mathtt{X\ Cop\ Exp\ A} \mid \mathtt{AP\ par\ AP} \end{array}$$

Consider Figure 4. Applying the function *ann* to this program produces the annotated version in Figure 5, where each statement now has an empty stack.

As mentioned previously, annotation does not handle state-saving with this now implemented within the operational semantics. Each time a statement execution completes, a unique identifier is added to that statement's source code stack. To synchronise the use of these identifiers, the next available identifier is retrieved through the function `next()`. To avoid further data races, there is mutual exclusion on the use of this atomic function between the parallel programs. This function, typically used as `m = next()`,

```
X+=Y+2[1] par (Y=X+2[2]; X=4[3])
```

Figure 6: Final Annotated program

```
X−=Y+2[1] par (X=4[3]; Y=X+2[2])
```

Figure 7: Inverted program

assigns the value of the next available identifier to `m`, while incrementing the value it will return next time by one. Identifiers will index reversal information, meaning the stacks within the auxiliary store now consist of elements of the form (`i`,`v`), where `i` is an identifier and `v` is a value. The following operational semantics defines the forwards execution, where $f(\mathtt{A})$ indicates an update of the source code stack `A`.

$$[\text{Skip}]\ \frac{}{(\mathtt{skip\ A;P},\sigma,\delta)\rightarrow(\mathtt{P},\sigma,\delta)} \qquad [\text{Seq1}]\ \frac{(\mathtt{S\ A},\sigma,\delta)\rightarrow(\mathtt{S}'\ f(\mathtt{A}),\sigma',\delta')}{(\mathtt{S\ A;P},\sigma,\delta)\rightarrow(\mathtt{S}'\ f(\mathtt{A});\mathtt{P},\sigma',\delta')}$$

$$[\text{DA1}]\ \frac{}{(\mathtt{X\ =\ v\ A},\sigma,\delta)\rightarrow(\mathtt{skip\ m:A},\sigma[\mathtt{X}\mapsto\mathtt{v}],\delta[(\mathtt{m},\sigma(\mathtt{X}))\mapsto\mathtt{X}])}\quad \text{where } \mathtt{m}=\mathtt{next()}$$

$$[\text{CA1}]\ \frac{}{(\mathtt{X\ cop\ v\ A},\sigma,\delta)\rightarrow(\mathtt{skip\ m:A},\sigma[\mathtt{X}\mapsto\sigma(\mathtt{X})\ \mathtt{op\ v}],\delta)}\quad \text{where } \mathtt{m}=\mathtt{next()}$$

$$[\text{DA2}]\ \frac{(\mathtt{e},\sigma,\delta)\rightarrow(\mathtt{e}',\sigma',\delta')}{(\mathtt{X\ =\ e\ A},\sigma,\delta)\rightarrow(\mathtt{X\ =\ e'\ A},\sigma',\delta')} \qquad [\text{CA2}]\ \frac{(\mathtt{e},\sigma,\delta)\rightarrow(\mathtt{e}',\sigma',\delta')}{(\mathtt{X\ cop\ e\ A},\sigma,\delta)\rightarrow(\mathtt{X\ cop\ e'\ A},\sigma',\delta')}$$

$$[\text{P1}]\ \frac{}{(\mathtt{P\ par\ skip},\sigma,\delta)\rightarrow(\mathtt{P},\sigma,\delta)} \qquad [\text{P2}]\ \frac{}{(\mathtt{skip\ par\ Q},\sigma,\delta)\rightarrow(\mathtt{Q},\sigma,\delta)}$$

$$[\text{P3}]\ \frac{(\mathtt{P},\sigma,\delta)\rightarrow(\mathtt{P}',\sigma',\delta')}{(\mathtt{P\ par\ Q},\sigma,\delta)\rightarrow(\mathtt{P'\ par\ Q},\sigma',\delta')} \qquad [\text{P4}]\ \frac{(\mathtt{Q},\sigma,\delta)\rightarrow(\mathtt{Q}',\sigma',\delta')}{(\mathtt{P\ par\ Q},\sigma,\delta)\rightarrow(\mathtt{P\ par\ Q}',\sigma',\delta')}$$

Sequential composition is handled similarly to our first approach, with the exception that the source code stacks are present and potentially modified. The expressions within assignments are handled either by [DA2] or [CA2] respectively, with no identifier association due to our assumption of the atomicity of statements. When the execution of a destructive assignment completes, the rule [DA1] associates a new identifier `m` within the source code stack `A`, and uses it to index the old value $\sigma(\mathtt{X})$ stored on δ. Constructive assignments complete via the rule [CA1], where an identifier is associated but no reversal information is stored. Parallel composition executes as expected, with either program able to make a step of execution, until one side is complete meaning the statement becomes sequential.

After the execution of the annotated program under these semantics, the *final* annotated version with populated stacks is produced. Linking again to our example with the execution interleaving (`P1;Q1;Q2`), initial state σ with values `X=1` and `Y=1` and initial auxiliary store δ, the final annotated version is shown in Figure 6. The program state σ' after this execution has the values `X=4` and `Y=6`, while δ' contains the necessary reversal information.

6.4 Inversion and Reverse Execution

Inversion now takes the *final annotated program* and produces a relatively similar inverted version. This contains all statements of the given program in its inverted program order, with the inverted version of all constructive assignments. Due to the similarity between annotated and inverted versions, we now let $\mathbb{S}'$ be the set of both annotated and inverted statements of this approach. The function $inv : \hat{\mathbb{P}} \rightarrow \mathbb{P}^{-1}$ recursively applies the re-defined function $i : \mathbb{S}' \rightarrow \mathbb{S}'$ to each statement in reverse order. Both *inv* and *i* are now given, with `icop` as defined in Section 5.

$$\begin{array}{ll} \mathit{inv}(\varepsilon\ \mathtt{A}) = \varepsilon\ \mathtt{A} & \mathit{inv}(\mathtt{AS;AP}) = \mathit{inv}(\mathtt{AP}); i(\mathtt{AS}) \\ i(\mathtt{skip\ A}) = \mathtt{skip\ A} & i(\mathtt{X\ =\ e\ A}) = \mathtt{X\ =\ e\ A} \\ i(\mathtt{X\ cop\ e\ A}) = \mathtt{X\ icop\ e\ A} & i(\mathtt{P\ par\ Q}) = (\mathit{inv}(\mathtt{P}))\ \mathtt{par}\ (\mathit{inv}(\mathtt{Q})) \end{array}$$

The inverted version does not make use of the reversal information, and instead must be executed under a separate set of operational semantics for reverse execution. These semantics are responsible for all interaction with any information saved, as well as using the identifiers to direct inversion along the correct interleaving order. This is implemented using the mutually exclusive and atomic function `previous()`, related to the function `next()` such that `next() = previous() + 1`. The statement `m = previous()` checks that the current value of `m` matches the current value of `previous()`, as well as decrementing the value the function will return next time by 1. The statement `m == previous()` again checks that `m` is equal to `previous()`, but does not decrement the value it will return next time. This forces all steps of the evaluation to happen sequentially, reflecting our assumption of statement atomicity. The functions `previous()` and `next()` are strongly related, meaning the execution of one must update the value of the other accordingly. Here the rules for sequential and parallel composition are similar to those in Section 6.3, but with the transition relation $\rightsquigarrow$ replacing $\rightarrow$, hence they are omitted to save space.

$$[\text{RDA}]\quad \frac{\mathtt{A = m:A'} \quad \delta\mathtt{(X) = (m,v):X'} \quad \mathtt{m = previous()}}{(\mathtt{X = e\ A}, \sigma, \delta) \rightsquigarrow (\mathtt{skip\ A'}, \sigma[\mathtt{X} \mapsto \mathtt{v}], \delta[\mathtt{X/X'}])}$$

$$[\text{RCA1}]\quad \frac{\mathtt{A = m:A'} \quad \mathtt{m = previous()}}{(\mathtt{X\ cop\ v\ A}, \sigma, \delta) \rightsquigarrow (\mathtt{skip\ A'}, \sigma[\mathtt{X} \mapsto \sigma\mathtt{(X)\ op\ v}], \delta)}$$

$$[\text{RCA2}]\quad \frac{(\mathtt{e}, \sigma, \delta) \rightsquigarrow (\mathtt{e'}, \sigma', \delta') \quad \mathtt{A = m:A'} \quad \mathtt{m == previous()}}{(\mathtt{X\ cop\ e\ A}, \sigma, \delta) \rightsquigarrow (\mathtt{X\ cop\ e'\ A}, \sigma', \delta')}$$

Destructive assignments are handled via the single rule [RDA] as no evaluation of the expression `e` is required. A destructive assignment can be executed provided its most recent identifier matches both the current value of `previous()` and the index of the top element of its stack on δ. Provided these conditions hold, the variable is restored to its previous value retrieved from its stack on δ, before both of these stacks are popped. Constructive assignments require the two rules [RCA1] and [RCA2] as the expression must still be evaluated. Each step of the evaluation is executed sequentially by ensuring the identifiers match without removing them. Only when the assignment has executed will the identifiers be removed, restricting interleaving until this point.

Applying the function *inv* to the final annotated program in Figure 6 produces the inverted version in Figure 7. Execution of this inverted version under the reverse operational semantics starting with the state σ' with values `X=4` and `Y=6`, results in the reverse statement order of `Q2;Q1;P1`, the state σ with values `X=1` and `Y=1` and the auxiliary store δ. Therefore the execution has been successfully reversed with all variables restored to their initial values.

6.5 Correctness

We now outline our correctness results for the second approach. Annotation of a program `P` assigns empty stacks to the statements of the program. During execution, these stacks are populated with identifiers. Let's denote such an *update* of the stacks of *ann*(`P`) as $\rho(\mathit{ann}(\mathtt{P}))$. We now give propositions corresponding to those in Sections 4 and 5, however we defer all termination parts to future work. Proposition 3 shows that the behaviour of the original and annotated programs are semantically equivalent with respect to the data store σ, and that the annotated program will populate both the stacks within the source code and the auxiliary store.

Proposition 3. *Let* $\mathtt{P}$ *be a program and* $ann(\mathtt{P}) = \mathtt{P}'$. *If* $(\mathtt{P},\sigma,\delta) \rightarrow^* (\mathtt{skip},\sigma',\delta)$ *for some* σ'*, then* $(\mathtt{P}',\sigma,\delta) \rightarrow^* (\mathtt{skip\ C},\sigma',\delta')$ *for some* $\mathtt{C}$ *and* δ' *and the computation* $(\mathtt{P}',\sigma,\delta) \rightarrow^* (\mathtt{skip\ C},\sigma',\delta')$ *produces an update* $\rho(\mathtt{P}')$ *for some* ρ.

Proposition 4 shows that executing the inverted program under the two stores and the updated source code stacks does indeed reverse all components to their initial values, as well as using the identifiers stored in the code to direct the execution.

Proposition 4. *Let* $\mathtt{P}$ *be a program and* $ann(\mathtt{P}) = \mathtt{P}'$. *If* $(\mathtt{P},\sigma,\delta) \rightarrow^* (\mathtt{skip},\sigma',\delta)$ *for some* σ'*, then* $(inv(\rho(\mathtt{P}')),\sigma',\delta') \rightsquigarrow^* (\mathtt{skip\ C},\sigma,\delta)$ *for some* $\mathtt{C}$*,* δ' *and* ρ.

At least two additional lemmas are used throughout the proofs of the two propositions above. These correspond to the lemmas used in Sections 4 and 5, and are listed below.

Lemma 3. *Let* $\mathtt{S}$ *be a program statement and* $ann(\mathtt{S}) = \mathtt{S\ A}$ *for some* $\mathtt{A}$. *If* $(\mathtt{S},\sigma,\delta) \rightarrow^* (\mathtt{skip},\sigma',\delta)$ *for some* σ'*, then* $(\mathtt{S\ A},\sigma,\delta) \rightarrow^* (\mathtt{skip\ A}',\sigma',\delta')$ *for some* $\mathtt{A}'$ *and* δ'.

Lemma 4. *Let* $\mathtt{S}$ *be a program statement and* $ann(\mathtt{S}) = \mathtt{S\ A}$ *for some* $\mathtt{A}$. *If* $(\mathtt{S},\sigma,\delta) \rightarrow^* (\mathtt{skip},\sigma',\delta)$ *for some* σ'*, then* $(inv(\mathtt{S\ A}'),\sigma',\delta') \rightsquigarrow^* (\mathtt{skip\ C},\sigma,\delta)$ *for some* $\mathtt{A}'$*,* δ' *and* $\mathtt{C}$.

7 Conclusion

We have presented an approach to reversing an imperative programming language, using the state-saving notion. We have defined two functions, namely *aug* and *inv*, capable of producing the augmented version and inverted version of an originally irreversible program, respectively. We have proved that our augmentation does not alter the behaviour of the program with respect to the data store, and that it saves the necessary information to revert the program state after execution to that of before. The auxiliary store used to save this reversal information is also proved to revert to its initial state, ensuring no extra garbage data is produced.

We also described a modification to our first approach to include parallelism within a restricted language, while avoiding a number of issues parallelism introduces. We defined a function *ann* and redefined *inv* to support the recording of the interleaving order into the source code. Two sets of operational semantics are defined, one performing the state-saving for forwards execution, and another performing the inversion for reverse execution. Finally, we propose the correctness results for this modified approach.

In the future, we shall relax the language restriction and support both conditional statements and while loops alongside parallel statements. The assumption of statement atomicity will be removed when considering a richer language which supports locks and mutual exclusion, allowing the approach described here to be implemented within the language itself. We will continue to extend the approach towards the complexity of C.

Acknowledgements

We are grateful to the referees for their detailed and helpful comments and suggestions. The authors acknowledge partial support of COST Action IC1405 on Reversible Computation - extending horizons of computing. The second author acknowledges the support by the University of Leicester in granting him Academic Study Leave, and thanks Nagoya University for support during the study leave. The third author acknowledges the support by JSPS KAKENHI grants JP17H0722 and JP17K19969.

References

[1] H. Agrawal, R. A. DeMillo & E. H. Spafford (1991): *An Execution-Backtracking Approach to Debugging*. *IEEE Software* 8(3), pp. 21–26, doi:10.1109/52.88940.

[2] B. Biswas and R. Mall (1999): *Reverse Execution of Programs*. *SIGPLAN Notices* 34(4), pp. 61–69, doi:10.1145/312009.312079.

[3] C. D. Carothers, K. S. Perumalla & R. Fujimoto (1999): *Efficient Optimistic Parallel Simulations using Reverse Computation*. *ACM Transactions on Modelling and Computer Simulation* 9(3), pp. 224–253, doi:10.1145/347823.347828.

[4] D. Cingolani, M. Ianni, A. Pellegrini & F. Quaglia: *Mixing Hardware and Software Reversibility for Speculative Parallel Discrete Event Simulation*. In: *RC 2016*, *LNCS* 9720, Springer, doi:10.1007/978-3-319-40578-0_9.

[5] R. Fujimoto (1990): *Parallel Discrete Event Simulation*. *Communications of the ACM* 33(10), pp. 30–53, doi:10.1145/84537.84545.

[6] R. Glück & M. Kawabe (2004): *Derivation of Deterministic Inverse Programs Based on LR Parsing*. In: *FLOPS 2004*, *LNCS* 2998, Springer, pp. 291–306, doi:10.1007/978-3-540-24754-8_21.

[7] R. Glück & M. Kawabe (2005): *Revisiting an Automatic Program Inverter for LISP*. *SIGPLAN Notices* 40(5), pp. 8–17, doi:10.1145/1071221.1071222.

[8] D. Gries (1981): *The Science of Programming*. Springer, doi:10.1007/978-1-4612-5983-1.

[9] H. Hüttel (2010): *Transitions and Trees - An Introduction to Structural Operational Semantics*. Cambridge University Press, doi:10.1017/CBO9780511840449.

[10] R. Landauer (1961): *Irreversibility and Heat Generation in the Computing Process*. *IBM Journal of Research and Development* 5(3), pp. 183–191, doi:10.1147/rd.53.0183.

[11] C. Lutz (1986): *Janus: A Time-Reversible Language. A Letter to Dr. Landauer*. `http://tetsuo.jp/ref/janus.pdf`.

[12] K. Perumalla (2014): *Introduction to Reversible Computing*. CRC Press.

[13] I. C. C. Phillips & I. Ulidowski (2007): *Reversing Algebraic Process Calculi*. *J. Log. Algebr. Program.* 73(1-2), pp. 70–96, doi:10.1016/j.jlap.2006.11.002.

[14] I. C. C. Phillips, I. Ulidowski & S. Yuen (2012): *A Reversible Process Calculus and the Modelling of the ERK Signalling Pathway*. In: *RC2012*, *LNCS* 7581, Springer, pp. 218–232, doi:10.1007/978-3-642-36315-3_18.

[15] M. Schordan, D. R. Jefferson, P. D. Barnes Jr., T. Oppelstrup & D. J. Quinlan (2015): *Reverse Code Generation for Parallel Discrete Event Simulation*. In: *RC 2015*, *LNCS* 9138, Springer, pp. 95–110, doi:10.1007/978-3-319-20860-2_6.

[16] M. Schordan, T. Oppelstrup, D. Jefferson, P. D. Barnes Jr. & D. J. Quinlan (2016): *Automatic Generation of Reversible C++ Code and Its Performance in a Scalable Kinetic Monte-Carlo Application*. In: *SIGSIM-PADS 2016*, ACM, pp. 111–122, doi:10.1145/2901378.2901394.

[17] G. Vulov, C. Hou, R. W. Vuduc, R. Fujimoto, D. J. Quinlan & D. R. Jefferson (2011): *The Backstroke Framework for Source Level Reverse Computation Applied to Parallel Discrete Event Simulation*. In: *WSC 2011*, WSC, doi:10.1109/WSC.2011.6147998.

[18] T. Yokoyama, H. B. Axelsen & R. Glück (2008): *Principles of a Reversible Programming Language*. In: *Proceedings of the 5th Annual Conference on Computing Frontiers, 2008*, ACM, pp. 43–54, doi:10.1145/1366230.1366239.

[19] T. Yokoyama & R. Glück (2007): *A Reversible Programming Language and its Invertible Self-interpreter*. In: *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, ACM, pp. 144–153, doi:10.1145/1244381.1244404.