

Reversing an Imperative Concurrent Programming Language

Thesis submitted for the degree of
Doctor of Philosophy
at the University of Leicester



by
James Hoey
Department of Informatics
University of Leicester

2020

Abstract

We introduce an approach to performing reversible executions of programs written in an imperative concurrent programming language. Our language contains assignments, conditional and loop statements, blocks, local variables, potentially recursive procedures and an interleaving concurrent composition operator `par`. The traditional execution of programs is defined using Structured Operational Semantics. Given an original, irreversible program we automatically generate two modified versions. The first, named the annotated version, performs forward execution and saves any lost information necessary for reversal. We address challenges of reversing a concurrent execution by using identifiers to capture a specific execution order. All information required for reversal is saved via the operational semantics. We define two further semantics. The first defines annotated execution, performing the expected forward execution and saving all reversal information. The second set defines the behaviour of the inverted version of a program. This forward-executing program simulates reversal, using identifiers to determine the (inverted) execution order, and other reversal information to undo each respective forward step.

We produce several results. We show that saving information during a forward execution does not change the behaviour of the underlying program, and that executing an inverted version correctly restores the state to as it was prior to the corresponding forward execution. All reversal information is used during an inverse execution meaning our approach is garbage-free.

A simulator, named Ripple, implementing our approach is introduced, based on our three semantics. This shows our approach works, and allows both testing and evaluation of the performance, specifically execution time and memory overheads. Our experimental results show that time and memory overheads increase linearly with respect to the size of the data or program. We explore the use of Ripple within reverse debugging, and identify future work, including optimizations and relaxing the inverted order of independent concurrent statements.

Acknowledgements

I would like to thank my supervisor Irek Ulidowski for his guidance, support and patience throughout. I am also grateful to my second supervisor Prof. Tom Ridge. Special thanks go to Prof. Shoji Yuen for help over the last four years, and for stimulating discussions on reversing programs for debugging. I would also like to thank Prof. Emilio Tuosto and Prof. Simon Gay for detailed comments and suggestions. I acknowledge the support I have received from the European COST Action IC1405 Reversible Computation - Extending Horizons of Computing, giving me the opportunity to collaborate with leading researchers in the field. Lastly, I would like to thank the Engineering and Physical Sciences Research Council (EPSRC) for the funding that made this PhD possible.

I dedicate this thesis to my inspirational parents, Tom and Karen, for giving me the greatest start in life I could ever have wished for. And to my brothers, Thomas and Sam, and my sister, Emily, for always picking me up when I'm down.

Contents

List of Figures	vi
1 Introduction	1
1.1 Reversible Computation	3
1.2 Making Irreversible Programs Reversible	8
1.2.1 Reverse C Compiler (RCC)	9
1.2.2 Backstroke Framework	11
1.3 Main Contributions	12
1.4 Outline of the Thesis	14
2 Concurrent While Language	16
2.1 Imperative While Language	16
2.1.1 Syntax	16
2.2 Program State	17
2.3 Traditional Operational Semantics	20
2.4 Examples of Traditional Execution	24
2.5 Conclusion	25
3 Reversing a Concurrent While Language	26
3.1 Parallel Composition Challenges	26
3.2 Annotation	28
3.2.1 Saving Lost Information	29
3.2.2 Interleaving Order and Identifiers	32
3.2.3 Annotation Function	36
3.3 Forward Semantics of Annotated Programs	39
3.4 Examples of Annotated Execution	42
3.5 Inversion	45
3.5.1 Inverse Interleaving Order	45
3.5.2 Using Reversal Information	46
3.5.3 Inversion Function	47
3.6 Reverse Semantics of Inverted Programs	48

3.7	Examples of Reverse Execution	52
3.8	Correctness of Reversal	54
3.9	Conclusion	57
4	Reversing an Imperative Concurrent Language with Blocks, Local Variables and Procedures	58
4.1	Motivating Example	58
4.2	Concurrent Language and Program State	59
4.2.1	Syntax of Programs	59
4.2.2	Program State Environments	60
4.3	Traditional Operational Semantics	64
4.4	Examples of Traditional Execution	68
4.5	Annotation	70
4.5.1	Extended Reversal Information	70
4.5.2	Extended Identifier Use	72
4.5.3	Annotation Function	73
4.6	Forward Semantics of Annotated Programs	74
4.7	Examples of Annotated Execution	78
4.8	Inversion	80
4.8.1	Inverse Interleaving Order	80
4.8.2	Statement Reversal	81
4.8.3	Inversion Function	81
4.9	Reverse Semantics of Inverted Programs	82
4.10	Examples of Reverse Execution	87
4.11	Correctness of Reversal	89
4.12	Conclusion	89
5	Challenges of Proving Correctness	91
5.1	Multiple Interleaving Orders	91
5.2	Partial Executions	96
5.3	Stopping an Inverted Execution	99
5.4	Conclusion	103
6	Correctness of Annotation and Inversion	104
6.1	Equivalent Program States: Forward	104
6.2	Annotation Result	106
6.3	Equivalent Program States: Reversal	107
6.4	Inversion Result	107
6.5	Proof of Statement Property	111
6.6	Proof of Program Property	128

6.6.1	Proof of Part 1	129
6.6.2	Proof of Part 2	136
6.7	Conclusion	140
7	Ripple: Simulation Tool and Performance Evaluation	141
7.1	Software Architecture of Ripple	141
7.2	Key Functionality of Ripple	145
7.3	User Interface of Ripple	147
7.4	Performance Evaluation	150
7.4.1	Forward Execution Time	150
7.4.2	Reverse Execution Time	152
7.4.3	Forward Memory Usage	155
7.4.4	Reverse Memory Usage	156
7.5	Conclusion	157
8	Application to Debugging	159
8.1	Suitability of the Approach	159
8.2	Record Mode and Execution Histories	160
8.3	Airline Example	162
8.4	Stack Example	164
8.5	Conclusion	167
9	Conclusion	168
9.1	Summary	168
9.2	Evaluation	169
9.3	Future Work	171
	Bibliography	174

List of Figures

2.1	Syntax of the imperative concurrent while language	17
2.2	An example loop and its renaming	19
2.3	Example loop code and matching while environment entry	21
3.1	An example loop and both the expected and actual boolean sequence saved for reversal	31
3.2	Programs that reverse incorrectly under specific interleavings	33
3.3	Example use of identifiers to capture interleaving order	34
3.4	Example program execution and its populated auxiliary store	35
3.5	Syntax of annotated versions of original programs	36
3.6	Annotation function	37
3.7	An original program and its corresponding annotated version	38
3.8	Syntax of inverted versions of original programs	47
3.9	Inversion function	47
3.10	An executed annotated program and its corresponding inverse version	48
4.1	Program determining whether a number is odd or even using mutually recursive procedure calls. Note that variable and procedure removal statements are introduced later and so are omitted here.	59
4.2	Syntax of our extended imperative concurrent language	60
4.3	A sample of a program and the scope information required for variable evaluation, where Z is a global variable.	62
4.4	An original procedure and a renamed version	64
4.5	Syntax of our annotated concurrent language	73
4.6	Annotation function	74
4.7	Syntax of inverted versions of original programs	82
4.8	Inversion function	83
4.9	An executed block statement and the corresponding inverted version	84
5.1	A program and its inverse with non-matching skip steps	92
5.2	A program, a standard execution and its uniform version	93
5.3	A complete program beginning with several skip steps	94

5.4	Syntax of partially executed programs	96
5.5	An original program and a possible partially executed version	97
5.6	Definition of the function $SE()$	98
5.7	Definition of the function $inv^+()$	100
6.1	Two equivalent, but not equal, variable states	105
6.2	Diagram representation of proof outline	109
6.3	Restricted syntax of complete programs beginning with skip steps	109
7.1	The class diagram representing each type of object and how it is linked to others, with only two sub-statement classes shown (namely DA and IFS).	142
7.2	A program and its linked list representation	143
7.3	A parallel program and its linked list representation	144
7.4	Simplified syntax of programs supplied to the simulator	145
7.5	The welcome screen of the command line interface	147
7.6	Successful reading of a file	148
7.7	Forward execution of a program	148
7.8	Reverse execution of a program	149
7.9	Manual interleaving decision presented to the user	149
7.10	Graph representation of the overhead of annotated execution for each of our programs, showing the percentage increase of execution time of each of the three versions.	151
7.11	The program Loop 1	152
7.12	The program Airline 1	152
7.13	The program General 1	153
7.14	The program Bsort 1	153
7.15	The overhead of inverted executions compared to the corresponding annotated execution. The percentage change in execution time of the three versions of each program.	154
7.16	The overhead of inverted executions compared to the corresponding original execution. The percentage change in execution time of the three versions of each program.	155
8.1	Example concurrent program under a specific interleaving	160
8.2	Example history files of an execution via Ripple	161
8.3	Airline Example	162
8.4	Executed annotated stack program	164
8.5	Inverted version of the stack program	165

Chapter 1

Introduction

Traditional program execution is fundamentally irreversible. A reversible execution is one that can execute in either the forward or reverse direction, with a step of the reverse execution inverting the effects on the program state of the corresponding forward step. As programs written in traditional programming languages execute on traditional computers, information critical to restoration of the original program state is lost. Imagine a simple assignment statement that overwrites the current value of a given variable, resulting in the loss of this value that cannot be restored. Landauer’s principle claimed this lost information is dissipated as heat, and theorised that truly reversible execution must require no information is lost and may therefore produce more energy-efficient computation [44].

Previous work on reverse execution of programs fits into one of two categories. Firstly, several reversible programming languages have been developed [55, 97, 26, 96, 79, 29]. Such languages only allow fully reversible programs to be written and therefore have no loss of information. Conditionals and loops are reversible due to their post-conditions, evaluated during the reverse execution to determine control flow [97]. Though this is an effective and elegant implementation of reversible execution, there remains the difficulty of automatically converting from a widely used irreversible language to a reversible equivalent. Specifically the automatic generation of post-conditions is problematic, and therefore impacts the usability of reversible languages on traditional programming code.

This difficulty is avoided via work in the second category, namely adding reversibility to irreversible programs [63]. Such works [11, 93, 75, 76] begin with a traditional and commonly used programming language (or some equivalent) and describe the process of adding reversibility to it. Such approaches support reversibility of existing programs written in traditional programming languages, with no need for conversion between languages as reversibility is added (potentially automatically) on top. Each irreversible step of an execution is made reversible by saving any information it loses, with approaches including checkpointing and incremental state

saving [63]. Doing so introduces a trade-off between an increase in either execution time or memory usage [92].

This category of work contains reversibility of message passing languages, including algebraic process calculi [15, 67, 64], where concurrency is supported. Concurrency, also referred to throughout as parallel composition, leads to non-deterministic execution orders, distinct final program states, and challenges associated with correctly reversing this execution. Current work in the literature using a shared memory setting typically does not involve concurrency, and therefore avoids these challenges. Approaches (in message passing systems) focus on either backtracking reversibility [9], where the steps of an execution are reversed in exactly the inverted order in which they were performed, or causal-consistent reversibility [15, 45, 67, 64], where independent steps can be reversed provided all of their consequences have been previously reversed.

At the beginning of this research, we set out to achieve the following:

- To propose a method of reversing executions of a concurrent imperative programming language;
- To support common programming constructs from typical, widely used programming languages;
- To allow concurrent composition of programs, and to reverse in backtracking order (reversing statements in exactly the inverted order in which they were performed forwards);
- To ensure that our proposed method performs correct reversibility and does not produce garbage;
- To implement our method in the form of a simulator, that can be used to show that our approach works, to perform tests that would be difficult to do so by hand, and to evaluate the performance in terms of execution time and memory usage overheads;
- To demonstrate the use of our method to the field of reverse debugging.

In this thesis we introduce an approach to reversing executions of an imperative, concurrent programming language using shared memory. We begin with an imperative while language that supports a form of concurrent composition known as interleaving. We describe our approach to first saving any required information, named reversal information, during the execution of a modified forward program. This includes information lost by each individual statement, and a sequence of identifiers that we introduce to capture a specific interleaving order. We detail how all

reversal information saved is used within an execution of a modified inverted version of a program, with statements inverted in backtracking order. Our approach restores the final program state (from after the forward execution) back to as it was prior to the corresponding forward execution. Crucially each intermediate program state is restored, as statements are reversed in a step-by-step manner. This is important when applying this approach to reverse debugging [16], as start-to-finish reverse execution is not typically helpful for finding errors. With all reversal information used during an inverse execution, our proposed method of reversibility is garbage free.

We extend our imperative concurrent language to include blocks, local variables and procedures. Our method of reversing the while language is extended, with support added for each of the additional constructs listed. We address challenges introduced by scope (local variables sharing names), by recursive procedure calls and by two (or more) concurrent calls to the same procedure (the same code executing in two or more places at once).

We address the correctness of our approach using a number of results, including showing that our method correctly restores the initial program state, using information saved during a forward execution in such a manner that does not affect the behaviour of the underlying program. Another result demonstrates that all reversal information is used during an inverse execution, meaning the reversibility method defined here is garbage free.

Our method of reversibility is implemented in the form of a simulator. This demonstrates that our approach can indeed reverse program executions, and allows testing to be performed using examples that would be difficult to execute by hand. The third reason for this implementation is to evaluate the performance, considering execution time and memory usage overheads incurred.

Finally we explore the link between our approach to reversibility and the field of reverse debugging [16]. Reverse debugging uses the ability to invert an execution step-by-step as a method to find the underlying cause of a bug, a feature supported within our simulator. Additional features are also discussed, including a record mode that focuses entirely on debugging.

1.1 Reversible Computation

Reversible computation is an interesting area of research due to the numerous benefits it can provide in a range of applications. From 1961, the Landauer principle states that the erasure of 1 bit of information costs at least “ $kT \ln 2$ ” in energy, where k is the Boltzmann constant and T is the temperature of the heat sink [44]. This suggests that as reversible computation requires no information is lost (and therefore not dissipated as heat), it could lead to computation with increased en-

ergy efficiency. Several commonly occurring misconceptions of this principle were addressed in 2018 by Frank [18]. Bennett suggested that any execution can be reversed by recording a history of all information lost throughout, and then using this information to undo, or reverse, the forward execution [8]. There are two forms of reversibility, namely physical reversibility and logical reversibility. Physical reversibility focuses on producing executions that can be performed without a loss of energy and that are therefore energy efficient. This includes producing reversible hardware. Early work in this area has shown reversible Turing machines [8] and reversible logic gates to exist, including Toffoli [84] and Fredkin gates [20]. Such gates are important to quantum computing, as any quantum operation is reversible. Other work includes reversible cellular automata [83, 39].

The second type of reversibility, namely logical reversibility, is the ability to reproduce a previous program state from that of later (or at the end of) a specific computation. The work presented in this thesis focuses on logical reversibility, with the background of appropriate lines of research described below. Abramov and Glück provide important background information into inverse computation, including key terminology in their paper [1]. We remark here that throughout this thesis, we use ‘inversion’ and ‘reversal’ to refer to both the process of producing an inverted version of an irreversible program (program inverter [1]), and to the process of executing such an inverted version.

The European Cooperation in Science and Technology (COST) project, named IC1405 Reversible Computation - Extending Horizons of Computing, has been active for the previous four years (2015–2019) and brought together many of the leading researchers in this area [36].

Reversible Algebraic Process Calculi, Event Structures and Petri Nets

Danos and Krivine introduced an extended version of the Calculus of Communicating Systems (CCS), named RCCS, where processes are extended to contain memories [15]. These memories identify processes used within actions, allowing actions to be reversed in any order that respects all causal dependencies (the first definition of *causal-consistent reversibility*). This differs to *backtracking reversibility*, where actions (or steps) are reversed in exactly the inverted order in which they executed forwards [9]. Phillips and Ulidowski introduced an alternative approach to extending CCS with reversibility, named CCS with Communication Keys (CCSK) [67, 64]. Actions are annotated with an identifier (communication key), such that the two parties of a communication use this same identifier. This ensures both are undone together (causal-consistency). This was later extended with controllers that support modification of the direction and pattern of an execution [68], where out of causal order reversibility was first presented. Kuhn and Ulidowski introduced the Calculus

of Covalent Bonding (CCB), based on CCSK, which contains an operator allowing the modelling of local reversibility of covalent bonding in chemical reactions [40, 41].

Ulidowski et al. modelled reversibility within concurrent computation defined using event structures [66, 65, 85], and Graversen et al. produced event structure semantics of reversible CCS [28]. Reversible Petri nets have been studied [7, 56], as has causally-consistent reverse broadcast in CCS [57].

Reversible Languages

Reversible programming languages, in which only reversible programs can be written, have been defined. The language Janus, originally defined by Lutz and Derby [55] and expanded upon by Yokoyama et al. [97, 94, 95], is an example of an imperative reversible language. Each valid program written using Janus can be executed in both the forward and reverse directions, where no information is required for correct reversal being lost. This is achieved using only constructive assignments, the name given to increments or decrements, each of which are fully reversible. For example, $X += 1$ is reversed via $X -= 1$. Conditional statements and while loops receive post-conditions that are used to determine the inverse control flow and reflect the result of forward evaluation (ensuring the same branch is reversed). Consider the Fibonacci example shown in [94], containing the following forward conditional statement (left side) and the inverted version (right side), with arguments and branches omitted.

<pre> procedure fib(...) if n=0 then ... else ... fi x1=x2 </pre>	<pre> procedure fib(...) if x1=x2 then ... else ... fi n=0 </pre>
---	---

The pre-condition $n=0$ is used during forward execution (left conditional above), determining which branch to execute. The post condition $x1=x2$ is an assertion that must evaluate to true if reached via the true branch, and false if reached via the false branch. The reverse execution of this program (right conditional above) switches the pre and post conditions, meaning the inverse conditional evaluates the expression $x1=x2$ to determine the branch to invert. This expression (which was the post condition of the forward version) is evaluated during the reverse execution, potentially increasing execution time when compared to retrieving a saved boolean value (in state-saving approaches).

Glück and Yokoyama introduced the reversible imperative while language with tree-like structure, named R-WHILE, in [26], and reduced this into the core language R-CORE in [27]. This core of a reversible language contains one reversible assignment, one control flow operator, namely a loop `from X loop C until Y` where `X` is an assertion and `Y` is a test (capable of simulating a conditional statement) and the number of variables is limited.

Reversible functional programming languages include that defined by Yokoyama et al. [96], and Rfun as defined by Thomsen et al. [81]. More recently, the typed functional language CoreFun was introduced by Jacobsen et al. [37], containing a description of how this could be used as a core language for the development of a more modern reversible functional language.

Another frequently used programming paradigm is object-orientation. The reversible object oriented language named Joule was introduced by Schultz [79], and extended in [78] to remove the limitations associated with stack allocation of objects and to instead use region based memory management. Another reversible object oriented language named ROOPL was introduced by Haulund in [29], and further extended within [30, 14].

Other notable work includes the definition of reversible semantics for the concurrent functional language Erlang [62, 48], the reversible procedural language R developed by Frank [19], the reversible computer architectures Pisa [89] and Bob [82] (including the instruction set BobISA).

Reversible Debugging

An introduction to software bugs and the process of debugging is provided by Zeller [98]. A software bug, or defect, is the name given to an error within a program that causes misbehaviour, while debugging is the process of relating a failure (misbehaviour) to the underlying defect and the fixing this [98]. Each problem caused by a bug can be classified based on severity, with categories including *critical* (crash and loss of data) and *minor* (small loss of functionality with a workaround) [98]. Vipindeep et al. produced a list of common software bugs and how each can be avoided [90]. Many studies have been performed on bugs, including that of Zhenmin Li et al. [52] and Shan Lu et al. [54]. The latter produced results showing the majority of non-deadlock bugs were either atomicity or order violations, and that the majority of concurrency bugs depend on the ordering of only two threads. Special consideration must be given to concurrent programs, as these contain the possibility of ‘Heisenbugs’, a software bug that only occurs under certain execution orders. Musuvathi et al. implemented the tool Chess for finding such bugs in [60].

The traditional approach to debugging, named *cyclic debugging* as the misbehaving program is executed many times, is ideal for sequential programs. Each

execution is guaranteed to have the same behaviour and therefore the bug occurs each time. Cyclic debugging is not appropriate for use with parallel (or concurrent) programs, where there exists the potential for two consecutive executions of the same program to have a different execution order. This makes reproducing the bug, and in turn finding the underlying cause of it, difficult. An example is of testing a possible fix of such a bug, where the user must determine if the bug does not occur as a result of the fix or because the specific execution order was not followed.

Reversible computation and its link to debugging has been the focus of some previous works. In most cases each line of research fits into two broad groups, the first being *record-replay debuggers* and the second being *reverse debuggers*. Record-replay debuggers aim to allow the application of cyclic debugging to parallel programs, where a specific execution of a program (experiencing a bug) is recorded with the recorded information used to force all future forward executions to behave the same. Examples include Instant Replay [50], RECPlay [72], Jockey [74], Liblog [22] BugNet [61]. Many record-replay debuggers use approaches described in Section 1.2.

Reverse debugging typically does not require re-execution of the program in the forward direction, and instead backtracks over an execution. Engblom provides a detailed review of reverse debugging in [16]. The unix based debugger GDB has reverse capabilities for sequential programs as shown in [13], while an example of a commercially available reversible debugger is UndoDB [86, 87]. Biswas and Mall [10] discussed the use of reverse execution in debugging and proposed an approach to execute C programs bidirectionally. Agrawal et al. proposed an execution backtracking approach to debugging, where a history file is populated during a forward execution and used to backtrack over an execution, restoring the program state at each step [4]. Another example of a debugger that uses *program instrumentation* to record certain information is [12]. Another example of a reverse debugger is [17].

Up to this point, all reversible debuggers discussed have followed backtracking order [9], where statements are reversed in exactly the inverted order in which they were executed forwards. Consider the program `(a) par (b;c)` containing a statement `a` in parallel with the sequentially composed statements `b` and `c`. If the order of forward execution is `b` followed by `a` and then `c`, backtracking order requires the reverse execution order to be `c`, then `a` followed by `b`. Recent work has focussed on causal-consistent reversibility, where actions can be reversed in any order that maintains all causal dependencies [15, 45, 67, 64]. Assuming that `a` is independent to `b` and `c`, and that `b` causes `c`, a valid reverse execution of the forward sequence `b, a, c` is the causally consistent sequence of `a, c, b` (which is not backtracking).

Giachino et al. explored using this with debugging of the language μOz in [24], introducing the first implementation of a causal-consistent debugger CaReDeb [23], and causal-consistent rollback [25]. Lanese et al. proposed a causal-consistent re-

versible debugger for Erlang [47], named Cauder [46], and later introduced controlled causal-consistent replay [49].

Memory Models

An important consideration for both interleaving parallel composition and for defining operational semantics is the use of memory models. A memory model is a set of rules governing how a compiler can reorder low level read and write instructions. Adve et al. produced a description of common memory models and shown examples of each in [2], including *sequential consistency* as defined by Lamport [43] and *weak ordering* introduced in [3]. Lamport also introduced the *happened-before relation* [42] between events within a distributed system. Jaffe et al. described the link between the choice of memory model and software reliability in [38], claiming that the frequency of concurrency bugs does increase in relaxed memory models and that this increase diminishes as the number of parallel threads increase. The tool ‘diy’ allows the definition of memory models and subsequent testing of them [6].

1.2 Making Irreversible Programs Reversible

A number of paradigms of reversible computation are summarised by Perumalla [63]. These include *Compute-Copy-Uncompute* (where a modified forward version of a program is executed, the final outcome is copied and finally the inverse version is executed), *Forward-Reverse-Commit* (a program is executed with the ability to be reversed, until the point at which reversal is definitely no longer required and the execution is committed) which is used by Backstroke [93] due to its suitability for optimistic Parallel Discrete Event Simulation (Section 1.2.2). Others include *Undo-Redo-Do* and *Begin-Rollback-Commit*.

The forward execution of a program can be reversed by saving enough information throughout to correctly restore the program state. One approach is named *checkpointing* and records snapshots of the program state. Recording the program state after each step of an execution is named *full checkpointing*, which requires a large amount of memory and potentially stores redundant information (a part of the state that has not changed since the previous snapshot), but that allows immediate restoration to a desired position. *Periodic checkpointing* reduces the frequency of snapshots stored (and thus memory usage), and reverses to a point between checkpoints by restoring to the closest previous snapshot and then re-executing forwards.

Incremental checkpointing removes the saving of any redundant information, by recording only the parts of the program state that change. These records can then be composed to reverse an execution to a desired position in a step-by-step manner, not

supporting an intermediate restoration as is the case for full checkpointing. *Differential checkpointing* is like incremental, but stores all changes at each snapshot with respect to the original program state. Examples of approaches using checkpointing to support either record-replay or reverse debugging include [91, 17, 51, 71, 73].

Alternative approaches involve generating an inverted version of a given program that performs the desired reverse execution. Such inverted versions are considered ‘normal’ programs, in that they are compiled and executed in the same way as original programs. Examples are *compiler-based*, where a compiler parses the original program and transforms it into a reversible version, or *interpreter based*, where the interpreter uses instrumented code to generate log files used for reversal [63].

Akgul and Mooney proposed an approach in [5] that reduces the amount of information saved during a forward execution. This approach, named *Reverse Code Generation* (RCG) algorithm, aims to produce a reverse version of a program capable of regenerating the intermediate program states instead of restoring them. Static analysis of the assembly code is used to provide instruction level reverse execution without forward re-execution. This is limited to single threaded programs [5].

Two examples of state-saving approaches are discussed in more detail below, each of which relate to the work we present here.

1.2.1 Reverse C Compiler (RCC)

The work we present here is related to the Reverse C Compiler (RCC), defined by Perumalla et al. [63, 11]. This is an example of a source-to-source approach to adding reversibility to the programming language C. This compiler is capable of producing two versions of a given code fragment, one that saves necessary information called transformed, and one that computes in reverse using it, called reversed. All information is recorded via ‘SAVE’ commands instrumented into the forward program, used by ‘RESTORE’ commands inserted into the reverse version, and saved onto runtime tapes, each of which operating as a stack (LIFO manner) [63].

RCC takes an original C program and first normalises it. Many constructs are reduced into equivalent versions using only constructs from a smaller subset. As a result, the number of constructs that must be considered is reduced. Any assignment statement that contains side effects, that is contains an expression that also modifies the value of a variable, are reduced into a sequence of assignments with no side conditions. From [63], the C code `int x=10, y=x++;` initialises `x` to 10, and sets `y` to `10 + 1` (while also incrementing the value of `x` to 11). The normalised version of this code is `int x=10, y=x; x++;`.

Another example is of a for loop occurring within an original program, which is changed into an equivalent version of the more general while loop. Consider the for

loop statement `for (int i = 0; i < 4; i++) { P }`, performing four iterations of the loop body `P`. The equivalent while loop version of this is `int i = 0; while (i < 4) { P; i++ }`.

As each forward statement executes, inserted `SAVE` commands record any lost information that is required to reversal. This includes the old value of any variable prior to its assignment, the result of conditional statement evaluation (a boolean value indicating the branch that was executed) and a loop iteration count. Consider the while loop `while (condition) {P}` based on code from [63]. Below is the transformed (left) and the reversed (right) versions of this loop produced by `RCC`, where `IP` is the reversed version of `P`. The transformed version of the loop initialises a loop counter variable `c`. For each iteration of the loop, this counter is incremented (`c++;`) meaning it holds the total number of iterations at the end. This value is then saved to the runtime tapes. The reversed version of this loop, re-initialises the loop counter variable `c` to the value saved during the forward execution, and then iterates until the count reaches 0 (via the condition `(c > 0)` with `c` decremented (`c--;`) for each iteration).

<code>int c = 0;</code>	<code>int c;</code>
<code>while (condition) {</code>	<code>RESTORE(c);</code>
<code>c++;</code>	<code>while (c > 0) {</code>
<code>P;</code>	<code>IP;</code>
<code>}</code>	<code>c--;</code>
<code>SAVE(c)</code>	<code>}</code>

Switch statements are handled similarly to conditionals, but with an integer saved to indicate which case has executed. Variables that go out of scope first have their final value saved, while jump statements (in the form of `goto`) are updated to be preceded by save commands recording the final value of any variables that will be out of scope as a result. The reversed version of each statement then uses this information saved to undo all effects of the forward execution, including undoing all the overwriting of all values and control flow. A crucial point to note is that some of the methods used here, including maintaining the loop counter variable, mean the original and transformed executions are *different* with respect to the program state. For example the program state after the transformed execution will contain an extra variable, namely `c`, that is present after the matching original execution. Further, parallel composition is not supported meaning all programs are sequential, and there is no proof of correctness to the best of our knowledge.

Perumalla et al. describe various optimisations that would be possible [63], including detecting invariant conditions (those that evaluate identically before and after the execution of the branch) and removing the need to save control flow information for it. We note that a single bit is sufficient to record the outcome of the condition evaluation, whereas the process of determining an invariant condition may take time. Further analysis is required to determine whether this trade-off is beneficial. A summary of the bit requirements for each type of supported statement is shown in [63].

1.2.2 Backstroke Framework

Our approach is also related to Backstroke introduced by Vulov et al. [93]. This is a framework developed to support automatic generation of reverse code for functions written in C++, built on the ROSE compiler [70]. A crucial distinction is that this focuses on its application to Parallel Discrete Event Simulation (PDES), a simulation methodology referring to the execution of a single discrete event simulation on a parallel computer [21]. Specifically this concerns optimistic PDES, where events are performed optimistically and rollback may need to be performed if causality errors are later detected [21].

Backstroke uses the Execute-Reverse-Commit paradigm as described above, and produces three versions of a given original function. From [93], the three versions of an original event *E* are named *E_forward* (equivalent to original execution but saves information needed for inversion), *E_reverse* (uses the saved information to undo all effects) and finally *E_commit* (that performs actions that are irreversible or should not be reversed). The first two of these versions behave similarly to the transformed and reverse versions generated by RCC. No code is generated for each of the reverse and commit versions, since they share the same implementation for all transformed events [75]. Each transformed version is linked with the Backstroke Runtime Library, where the forward execution records all information necessary to restore any previous state of the forward computation [75], and from which the reversed version of an event can be called.

As described in [93], Backstroke supports both snapshot inversion, where the initial values of all variables used within *E_forward* are saved prior to any execution, and a RCC-style incremental inversion, where previous values of variables and control flow information is saved. Finally, Backstroke also contains an alternative approach named path-oriented incremental inversion, where program analysis tools are used to evaluate the control flow of a program and generate a reverse version, with an example shown in [93] that does not require any information to be saved.

Backstroke has been used in more recent works by Schordan et al. [75, 76, 77]. In [75], Schordan et al. highlights it is sufficient to consider only memory modifying operations, namely assignment, memory allocation and memory deallocation, resulting in no control flow information being saved. Specifically, address-value pairs of modified memory locations are recorded, alongside details of dynamic memory management. All information is stored into the Run Time State Storage (RTSS) of Backstroke, which is essentially a double ended queue where forward and reverse code pushes and pops information from one end, while the commit code accesses information from the other. Schordan et al. summarised in [75] that “The drawback is the overhead in the forward code and a higher memory consumption (in most cases) than with approaches that take control flow into account”. Further work by Schordan et al. extended this approach with support for further C++ constructs such as templates, and the generation of reversible C++ assignment operators. This approach is demonstrated on the scalable kinetic Monte-Carlo C++ application. The performance evaluation shown there demonstrates increases in efficiency. To the best of our knowledge, no proof of correctness exists for the Backstroke framework.

1.3 Main Contributions

We define an approach to adding reversibility to an irreversible, imperative concurrent programming language. We begin by defining a concurrent programming language, supporting interleaving concurrent composition. Inspired by the use of Communication Keys in CCSK [67, 64], which are used to associate two processes involved in a communication together and where the necessary history is annotated into the calculus itself, we assign *identifiers* to statements in ascending order as they execute. We introduce *annotation* that modifies an original program such that each statement contains an *identifier stack*, and *inversion* that produces the inverted version. This is much like the annotated version but with an inverted statement order, using the identifiers in descending order to backtrack over the forward execution. Three Structured Operational Semantics (SOS) are defined, describing traditional (forward-only), annotated forward and reverse execution respectively. All information that must be saved (reversal information for a specific statement, e.g. the old value of a variable prior to an assignment statement), including identifiers to capture the interleaving order, is deferred to the annotated forward semantics. The reverse semantics use this extra information to reverse the effects of the respective forward statement (e.g. reversing an assignment statement by restoring it to the old value saved during the forward execution). Crucially each execution is defined using small step semantics, meaning each can be performed in a step-by-step manner. Each intermediate program state is restored, highlighting that we do not simply record a

snapshot of the program state prior to an execution and then restore to that, but that we incrementally build up our reversal information [63].

Block statements within our language introduce scope, where local variables can be declared and may share their name with at most one global variable and any number of other local versions. We implement blocks in such a way that they “clean up after themselves”. Any local variables or procedures declared within a block statement are deleted prior to the block completing its execution. This is performed via *removal statements* (like inverting a declaration statement). This introduces further information loss that could break reversibility if not considered.

The results regarding correctness show that both annotation and inversion behave as expected. This means that executing the annotated version of a program produces the same final program state as is produced by the original program, while also recording any information critical for reversal. Therefore annotation does not alter the behaviour of the original program (*annotation result*). Secondly, our results show that executing an inverted version of a program correctly restores the program state to as it was prior to the respective forward execution (*inversion result*). This also demonstrates that the environment used to record reversal information is restored to its initial state, meaning any information saved for reversal is used, and therefore shows no garbage data is produced. Such correctness results are often missing from the other approaches described within the literature.

We present a simulation tool, named Ripple, that automates the process of reversibly executing programs written in our concurrent language. This implements each of our three operational semantics, simulating traditional (forwards only with no information saving), annotated forward and reverse execution in either a step-by-step, or start-to-finish manner. Ripple allows our approach to be tested using large examples that would be difficult to perform by hand. Such tests have subsequently lead to adjustments to our method. Ripple is also used to evaluate the performance in terms of execution time and memory usage overheads. Additional features of Ripple aim to minimise the burden on the user, with one example being its ability to accept programs written in a simpler version of the syntax.

Ripple can also be used as a debugger. The step-by-step executions it offers are ideal for reverse debugging [16], as completely reversing an execution experiencing a bug is typically not useful (as the bug may occur at any point within an execution). We show several examples of commonly experienced software bugs, and detail the process of using Ripple (and therefore our approach) to perform reverse debugging. Additional features of Ripple focus solely on debugging, including record mode that saves enough information to ensure an execution is reproducible. This mode maintains two execution traces, namely detailing all interleaving decisions (which statements were possible and which was executed) and the semantics history

(an ordered list of transition rules applied). When used with manual interleaving mode, a user can correctly reproduce a recorded execution (traditional debugging approach). In each intermediate state of the simulation, the entire program state including any information saved for reversal can be viewed. This allows the user to investigate an execution to find the cause of a bug (defect).

1.4 Outline of the Thesis

Chapter 2 describes an imperative while language that contains an interleaving form of parallel composition. The traditional semantics of this language are shown, detailing the behaviour of programs written in this language and the effects on the program state.

Chapter 3 summarises the challenges of reversing a concurrent programming language. These challenges are overcome via the two further semantics introduced that perform annotated forward execution and inverted execution respectively. This follows an approach outlined in [33, 34] and is shown in full later.

This language is extended with blocks, local variables and potentially recursive procedures in Chapter 4. The modification required to the program state and additional semantics are defined, supporting traditional, forward and reverse execution.

Prior to proving our proposed method of reversibility to be correct, Chapter 5 addresses three main challenges this presented. We introduce partial executions, describe how to generate the inverted version of a partially executed program, and finally how to stop such an execution at the desired position via atomic statements.

Chapter 6 contains the proof of correctness of our approach, where two key properties are proven to hold showing the approach to implement correct reversal. This proof is outlined in [32] and shown in full later.

The simulation tool Ripple is introduced in Chapter 7, with details of the software architecture, user interface and performance evaluation displayed. Ripple was originally introduced in [32] and is described later.

The link between our approach to reversibility and debugging is explored in Chapter 8. Two examples of common software bugs are used here to explain how Ripple can aid the process of finding the underlying cause, building on work presented in [32, 31].

Finally Chapter 9 contains a summary of the contributions of this work, along with a description of possible future work.

The following is a list of our four publications that contain work also presented here in this thesis.

- J. Hoey, I. Ulidowski, and S. Yuen. Reversing imperative parallel programs. In *EXPRESS/SOS*, volume 255 of *Electronic Proceedings in Theoretical Computer Science*, pages 51–66, 2017
- J. Hoey, I. Ulidowski, and S. Yuen. Reversing parallel programs with blocks and procedures. In *EXPRESS/SOS*, volume 276 of *Electronic Proceedings in Theoretical Computer Science*, pages 69–86, 2018
- J. Hoey and I. Ulidowski. Reversible imperative parallel programs and debugging. In *Reversible Computation*, volume 11497 of *Lecture Notes in Computer Science*, pages 108–127. Springer, 2019
- J. Hoey, I. Lanese, N. Nishida, I. Ulidowski, and G. Vidal. A case study for reversible computing: Reversible debugging of concurrent programs. In *Reversible Computation: Theory and Applications*, volume 12070 of *Lecture Notes in Computer Science*. Springer, 2020. To appear.

Chapter 2

Concurrent While Language

In this chapter we describe the imperative programming language that will be used throughout the following chapters. This is much like any while language, with an example being that described by Hüttel [35]. This language contains assignments, conditional statements and while loops. Additionally, this also supports a form of parallel composition, namely interleaving.

We describe the environments used to represent the program state, and the semantics defining traditional (irreversible) execution of programs written in this language. Finally several examples of program execution are given.

2.1 Imperative While Language

We begin with the introduction of our concurrent while language. This programming language supports global integer variables that can be used within or manipulated by arithmetic expressions, as well as boolean conditions containing logic operators. Much like any imperative while language, the programming language introduced here supports assignments, conditional statements (*guarding* or *branching*) and while loop statements (*iteration*). In contrast to other while languages, we also support a form of parallel composition where program executions can be *interleaved*. The challenges introduced by interleaving are discussed in Section 3.1 of Chapter 3.

2.1.1 Syntax

The syntax of our language is now shown. Let \mathbb{P} be the set of programs P where each program P is generated by the Backus-Naur Form (BNF) in Figure 2.1. Let \mathbb{S} be the set of statements S where each statement S is defined by the BNF in Figure 2.1. Further let X denote a variable, E an arithmetic expression and B a boolean condition. Finally, let \mathbb{Z} be the set of integers, where a number n is such that $n \in \mathbb{Z}$, and both In and Wn be *construct identifiers* for conditionals and loops respectively. The need

$$\begin{aligned}
P &::= \varepsilon \mid S \mid P;P \mid P \text{ par } P \\
S &::= \text{skip} \mid X = E \mid \text{if In } B \text{ then } P \text{ else } P \text{ end} \mid \\
&\quad \text{while Wn } B \text{ do } P \text{ end} \\
E &::= X \mid n \mid (E) \mid E \text{ Op } E \\
B &::= T \mid F \mid \neg B \mid (B) \mid E == E \mid E > E \mid B \wedge B
\end{aligned}$$

Figure 2.1: Syntax of the imperative concurrent while language

for and use of construct identifiers will be explained in Section 2.2. The BNF of the entire syntax of this language is shown in Figure 2.1. Parallel composition of the programs P and Q is written as either $P \text{ par } Q$ in the following syntax, or as $\text{Par } \{P\} \{Q\}$ in several examples. For use throughout our work, we introduce the notion of a *complete program*, the name given to a program written in the language defined above that has not yet been executed in any way (meaning no conditions/expressions are evaluated yet).

2.2 Program State

Program state is represented using three environments. We first consider the *variable state*, where variables are bound to memory locations and each location holds an integer value. We note that only global variables are supported, all of which are uniquely named. All global variables are assumed to exist prior to and throughout an execution, with no declaration statement within the syntax from Figure 2.1.

The *variable environment* γ is used to map each variable name to a unique memory location. Let **Var** be the set of variable names and **Loc** be the set of memory locations. The variable environment γ is such that $\gamma : \mathbf{Var} \rightarrow \mathbf{Loc}$. A variable environment is manipulated using the following notation, where entries are not inserted or removed as all variables exist before and after an execution.

- $\gamma(X)$ - queries the variable environment γ and returns the memory location currently associated with the variable X .

The variable state is completed using the *data store* σ , where each required memory location is mapped to an integer value. Let **Loc** be as above and \mathbb{Z} be the set of integers. The data store σ is such that $\sigma : \mathbf{Loc} \rightarrow \mathbb{Z}$. The following notation allows the use of a data store (again there are no commands to add or remove entries).

- $\sigma(l)$ - returns the current integer value at memory location l .
- $\sigma[l \mapsto v]$ - updates the data store σ such that the memory location l is updated to hold the value v .

Given a variable environment γ and the corresponding data store σ , all variables of a given program can be evaluated. Consider the following example below.

Example 1. (Data store and variable environment) Let γ and σ exist as follows.

Variable environment γ		Data store σ	
Var name	Mem location	Mem location	Value
X	11	11	3
Y	12	12	7
Z	13	13	1

There exists three variables X, Y and Z, where each is mapped to a distinct memory location in γ , namely 11, 12 and 13 respectively. The initial values in σ are such that $X = 3$, $Y = 7$ and $Z = 1$. We now consider the evaluation of the variable Y. The variable environment γ is queried using Y to find the memory location 12 (written as $\gamma(Y) = 12$). This location is then used to query the data store σ to retrieve the value 7 (written as $\sigma(12) = 7$). ■

The final environment used to represent the program state is the *while environment* β . Included for consistency with Chapter 4, a copy of a while loop statement is produced prior to its execution, storing a copy of the original condition and loop body. Each copy of a loop is maintained within the while environment β . To avoid ambiguity, each while loop (and each subsequent copy) is given a unique name termed a *construct identifier*:

Definition 2.2.1. (Construct identifier) A construct identifier is a unique name given to each conditional and loop statement, written as **In** and **Wn** in Figure 2.1. Each is a string with an integer version number (explained below), e.g. i2.0.

Returning to the while environment, let **Wn** be the set of while loop construct identifiers **Wn** and \mathbb{P} be the set of programs P. The while environment β is such that $\beta : \mathbf{Wn} \rightarrow \mathbb{P}$. The following notation exists to access the while environment.

- $\beta[\mathbf{Wn} \Rightarrow \mathbf{P}]$ - inserts a mapping into β for the loop uniquely named **Wn**, to the copy P of the loop body.
- $\beta[\mathbf{Wn}]$ - removes the mapping for the while loop unique name **Wn** from β .

Prior to giving an example of a while environment, we consider *code reuse*. While loops allow iteration of a sub-program (the loop body) where the same code is executed multiple times and termed code reuse. Re-execution of the same code violates the uniqueness of construct identifiers, where nested constructs are executed several times using the same name. This problem can be avoided using a process called *renaming*. Construct identifiers are now extended to each have a *version*

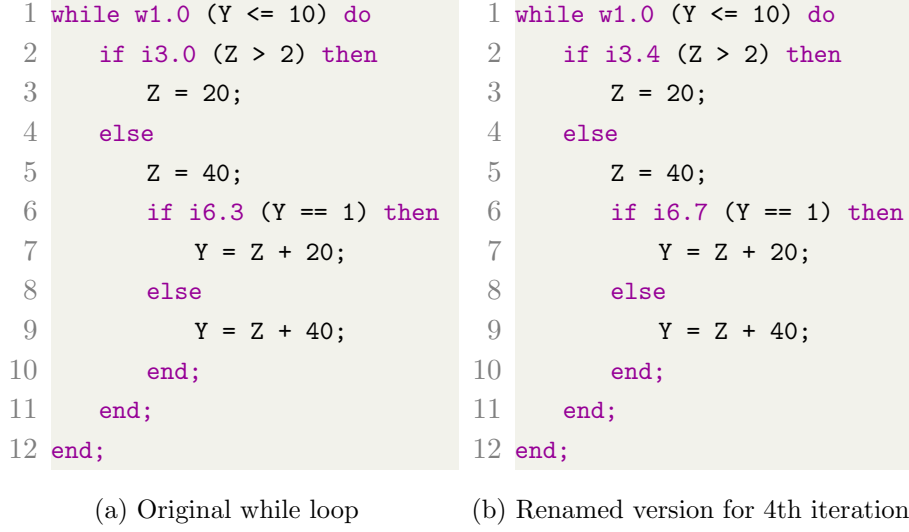


Figure 2.2: An example loop and its renaming

number, initially set to 0. For example a conditional statement named *i1* becomes *i1.0*. Before a copy of a loop is executed, all construct identifiers used within the body are renamed by incrementing the version number (*i1.0* becomes *i1.1*). This is implemented using the function $reL: \mathbb{P} \rightarrow \mathbb{P}$ that takes the original loop body. All construct identifiers are now unique and not have yet been used. Consider Example 2 that demonstrates the renaming applied to an example while loop body.

Example 2. (Renaming a while loop) A while loop is shown in Figure 2.2(a). For each iteration, all nested construct identifiers are updated to their next version number (with the other left unchanged). The renamed version of this loop for the fourth iteration is shown in Figure 2.2(b). ■

We are now ready to state an example of a while environment, namely Example 3.

Example 3. (While environment) Consider the program shown in Figure 2.3(a) which contains a while loop statement. The very first step of the execution of this loop is to create the corresponding mapping within the while environment β . This while environment entry is shown in Figure 2.3(b), indexed using the unique construct identifier given to this while loop. ■

The three environments introduced above are necessary to represent program state. The full program state, containing all three environments, is represented using \square , such that $\square = (\sigma, \gamma, \beta)$.

2.3 Traditional Operational Semantics

This section contains the operational semantics of the traditional execution of programs written in the previously defined programming language. This is termed traditional as the resulting execution loses information and is therefore irreversible. As explained by Hüttelin [35], we use *Structural Operational Semantics* (SOS) [69] to define a *transition system* that describes the behaviour of each of our programming language constructs (Figure 2.1).

A transition system is a directed graph, with each vertex a *configuration* and each edge a *transition*. A *configuration* represents a snapshot of an execution and is composed of a program P and program state \square , written as $(P \mid \square)$. Two configurations can be linked together as a transition, where the second configuration can be reached from the first via a single step of an execution. Each such transition is a member of the *transition relation* \hookrightarrow (traditional execution), which is defined as the least relation generated by the set of *transition rules* contained within this section. Each transition rule defines a possible transition from the relation \hookrightarrow , and consists of a *conclusion* (the transition this rule represents), potentially many *premises* (other transitions or claims that must hold for the conclusion to be valid) and an optional *side condition* (constraints placed on components of the rule). We write transition rules in the following way.

$$\frac{\text{premises}}{\text{conclusion}} \text{ side condition}$$

All of our transition rules are *compositional*, where all premises of a rule use only entities contained within the conclusion. This allows a sequence of instances of transition rules to prove a transition, forming the so-called *inference tree*. An inference tree contains a *leaf*, the name given to a rule that does not contain any further transitions as a premise. This encapsulates all *axioms* (transition rules that have no premises) such as [S2], and those that contain premises that are not transitions such as [W6]. Examples of inference trees are given in Section 2.4.

We are now ready to give a syntax-directed, *small step semantics* that defines traditional execution. Each transition rule represents a single step of a larger execution, meaning the resulting configuration need not be *terminal* (no further transitions from this configuration are possible), and instead can be *intermediate*. We use \hookrightarrow_a and \hookrightarrow_b to represent a step of arithmetic and boolean expression evaluation respectively, and \hookrightarrow^* be the reflective and transitive closure by an arbitrary number of steps. In the following rules, we use \square to represent all elements of the tuple (σ, γ, β) that are not explicitly stated with the rule. For example, \square in the following rule [A1] represents (β) as both σ and γ are explicitly stated.

	Loop name	Code
<pre> 1 while w1 (X > 4) do 2 X = X - 1; 3 Y = X + Y; 4 end </pre>	w1	<pre> while w1 (X > 4) do X = X - 1; Y = X + Y; end; </pre>
(a) While loop		(b) Corresponding while environment

Figure 2.3: Example loop code and matching while environment entry

Arithmetic Expressions

Arithmetic expressions are evaluated in potentially many steps. Let a_i be an arithmetic expression, X be a variable, and v, m and $n \in \mathbb{Z}$. Finally let the set of arithmetic expression operators $\mathbf{Op} = \{+, -, *, /\}$, such that $op \in \mathbf{Op}$.

$$\begin{array}{ll}
[A1] & \overline{(X \mid \sigma, \gamma, \square) \hookrightarrow_a (\sigma(\gamma(X)) \mid \sigma, \gamma, \square)} \\
[A2] & \overline{\frac{v = n \text{ op } m}{(n \text{ op } m \mid \square) \hookrightarrow_a (v \mid \square)}} \\
[A3] & \overline{((v) \mid \square) \hookrightarrow_a (v \mid \square)} \\
[A4] & \overline{\frac{(a_0 \mid \square) \hookrightarrow_a (a'_0 \mid \square)}{((a_0) \mid \square) \hookrightarrow_a ((a'_0) \mid \square)}} \\
[A5] & \overline{\frac{(a_0 \mid \square) \hookrightarrow_a (a'_0 \mid \square)}{(a_0 \text{ op } a_1 \mid \square) \hookrightarrow_a (a'_0 \text{ op } a_1 \mid \square)}} \\
[A6] & \overline{\frac{(a_1 \mid \square) \hookrightarrow_a (a'_1 \mid \square)}{(a_0 \text{ op } a_1 \mid \square) \hookrightarrow_a (a_0 \text{ op } a'_1 \mid \square)}}
\end{array}$$

Boolean Expressions

Let b be a boolean expression and ba_i be either a boolean or arithmetic expression. Let the set of boolean expression operators $\mathbf{Bop} = \{>, >=, <, <=, ==\}$ if used between two arithmetic expressions and $\mathbf{Bop} = \{\&, ||, ==\}$ if used between two boolean expressions. Finally $bop \in \mathbf{Bop}$.

$$\begin{array}{ll}
[B1] & \overline{(\neg T \mid \square) \hookrightarrow_b (F \mid \square)} \\
[B2] & \overline{(\neg F \mid \square) \hookrightarrow_b (T, \square)} \\
[B3] & \overline{\frac{(b \mid \square) \hookrightarrow_b (b' \mid \square)}{(\neg b \mid \square) \hookrightarrow_b (\neg b' \mid \square)}} \\
[B4] & \overline{\frac{ba_2 = ba_0 \text{ bop } ba_1}{(ba_0 \text{ bop } ba_1 \mid \square) \hookrightarrow_b (ba_2 \mid \square)}} \\
[B5] & \overline{\frac{(ba_0 \mid \square) \hookrightarrow_b (ba'_0 \mid \square)}{(ba_0 \text{ bop } ba_1 \mid \square) \hookrightarrow_b (ba'_0 \text{ bop } ba_1 \mid \square)}} \\
[B6] & \overline{\frac{(ba_1 \mid \square) \hookrightarrow_b (ba'_1 \mid \square)}{(ba_0 \text{ bop } ba_1 \mid \square) \hookrightarrow_b (ba_0 \text{ bop } ba'_1 \mid \square)}}
\end{array}$$

Sequential Composition

A statement S sequentially composed with a program P must execute until it reaches skip ([S1]). The skip is then removed and the execution continues with P ([S2]).

$$\begin{array}{ll}
[S1] & \overline{\frac{(S \mid \square) \hookrightarrow (S' \mid \square')}{(S; P \mid \square) \hookrightarrow (S'; P \mid \square')}} \\
[S2] & \overline{(\text{skip}; P \mid \square) \hookrightarrow (P \mid \square)}
\end{array}$$

Parallel Composition

A step of the parallel composition of two programs P and Q can originate from either of the programs ($[P1]/[P2]$) (provided each is valid). Only once both sides have completed (reached skip) can the parallel statement itself finish ($[P3]$).

$$\begin{array}{l}
 [P1] \quad \frac{(P \mid \Box) \hookrightarrow (P' \mid \Box')}{(P \text{ par } Q \mid \Box) \hookrightarrow (P' \text{ par } Q \mid \Box')} \quad [P2] \quad \frac{(Q \mid \Box) \hookrightarrow (Q' \mid \Box')}{(P \text{ par } Q \mid \Box) \hookrightarrow (P \text{ par } Q' \mid \Box')} \\
 [P3] \quad \frac{}{(\text{skip par skip} \mid \Box) \hookrightarrow (\text{skip} \mid \Box)}
 \end{array}$$

Assignment

Assignments complete in a single step ($[D1]$). The arithmetic expression e is (atomically) evaluated to a value v (via $(e \mid \sigma, \gamma, \Box) \hookrightarrow_a^* (v \mid \sigma, \gamma, \Box)$). The variable itself is evaluated to its memory location l ($\gamma(X) = l$). The location l is updated to hold the new value v (overwriting its previous value via $\sigma[l \mapsto v]$).

$$[D1] \quad \frac{(e \mid \sigma, \gamma, \Box) \hookrightarrow_a^* (v \mid \sigma, \gamma, \Box) \quad \gamma(X) = l}{(X = e \mid \sigma, \gamma, \Box) \hookrightarrow (\text{skip} \mid \sigma[l \mapsto v], \gamma, \Box)}$$

Conditional Statements

The boolean condition is first (atomically) evaluated to true ($[I1T]$) or false ($[I1F]$). Depending on this evaluation, either the true ($[I2]$) or false ($[I3]$) branch is executed. Note that the conditional is static following the general approach used in [67, 64], where the overall structure and the non-executed branch remain in place necessary for reversal. This is different to traditional semantics of conditionals including those of Hüttel[35], and is necessary for reversal purposes explained later in Chapter 3. Further, the rules $[I1T]$ and $[I1F]$ could be combined into a single rule, though we choose not to for simplicity later on. Once the true ($[I4]$) or false ($[I5]$) branch has executed completely, the conditional statement itself can then complete.

$$\begin{array}{l}
 [I1T] \quad \frac{(b \mid \Box) \hookrightarrow_b^* (T \mid \Box)}{(\text{if In } b \text{ then } P \text{ else } Q \text{ end} \mid \Box) \hookrightarrow (\text{if In } T \text{ then } P \text{ else } Q \text{ end} \mid \Box)} \\
 [I1F] \quad \frac{(b \mid \Box) \hookrightarrow_b^* (F \mid \Box)}{(\text{if In } b \text{ then } P \text{ else } Q \text{ end} \mid \Box) \hookrightarrow (\text{if In } F \text{ then } P \text{ else } Q \text{ end} \mid \Box)} \\
 [I2] \quad \frac{(P \mid \Box) \hookrightarrow (P' \mid \Box')}{(\text{if In } T \text{ then } P \text{ else } Q \text{ end} \mid \Box) \hookrightarrow (\text{if In } T \text{ then } P' \text{ else } Q \text{ end} \mid \Box')} \\
 [I3] \quad \frac{(Q \mid \Box) \hookrightarrow (Q' \mid \Box')}{(\text{if In } F \text{ then } P \text{ else } Q \text{ end} \mid \Box) \hookrightarrow (\text{if In } F \text{ then } P \text{ else } Q' \text{ end} \mid \Box')} \\
 [I4] \quad \frac{}{(\text{if In } T \text{ then skip else } Q \text{ end} \mid \Box) \hookrightarrow (\text{skip} \mid \Box)} \\
 [I5] \quad \frac{}{(\text{if In } F \text{ then } P \text{ else skip end} \mid \Box) \hookrightarrow (\text{skip} \mid \Box)}
 \end{array}$$

While Loops

A loop with zero iterations ([W1]) is undefined within the while environment ($\beta(\mathbf{Wn}) = \text{und}$) and has a condition that evaluates to false (via $(\mathbf{b} \mid \beta, \square) \hookrightarrow_{\mathbf{b}}^* (\mathbf{F} \mid \beta, \square)$). Such a loop terminates in a single step.

$$[\text{W1}] \quad \frac{\beta(\mathbf{Wn}) = \text{und} \quad (\mathbf{b} \mid \beta, \square) \hookrightarrow_{\mathbf{b}}^* (\mathbf{F} \mid \beta, \square)}{(\mathbf{while} \ \mathbf{Wn} \ \mathbf{b} \ \mathbf{do} \ \mathbf{P} \ \mathbf{end} \mid \beta, \square) \hookrightarrow (\mathbf{skip} \mid \beta, \square)}$$

The last step of a loop with at least one iteration ([W2]) must be defined within the while environment ($\beta(\mathbf{Wn}) = \text{def}$) and have a condition that evaluates to false (via $(\mathbf{b} \mid \beta, \square) \hookrightarrow_{\mathbf{b}}^* (\mathbf{F} \mid \beta, \square)$). Such a loop terminates, removing the entry from the while environment ($\beta[\mathbf{Wn}]$).

$$[\text{W2}] \quad \frac{\beta(\mathbf{Wn}) = \text{def} \quad (\mathbf{b} \mid \beta, \square) \hookrightarrow_{\mathbf{b}}^* (\mathbf{F} \mid \beta, \square)}{(\mathbf{while} \ \mathbf{Wn} \ \mathbf{b} \ \mathbf{do} \ \mathbf{P} \ \mathbf{end} \mid \beta, \square) \hookrightarrow (\mathbf{skip} \mid \beta[\mathbf{Wn}], \square)}$$

The first step of a loop with at least one iteration ([W3]) must initially be undefined within the while environment ($\beta(\mathbf{Wn}) = \text{und}$) and have a condition that evaluates to true (via $(\mathbf{b} \mid \beta, \square) \hookrightarrow_{\mathbf{b}}^* (\mathbf{T} \mid \beta, \square)$). Then a renamed version of the loop, named \mathbf{R} with body $\text{reL}(\mathbf{P})$, is inserted into the while environment ($\beta[\mathbf{Wn} \Rightarrow \mathbf{R}]$).

$$[\text{W3}] \quad \frac{\beta(\mathbf{Wn}) = \text{und} \quad (\mathbf{b} \mid \beta, \square) \hookrightarrow_{\mathbf{b}}^* (\mathbf{T} \mid \beta, \square)}{(\mathbf{while} \ \mathbf{Wn} \ \mathbf{b} \ \mathbf{do} \ \mathbf{P} \ \mathbf{end} \mid \beta, \square) \hookrightarrow (\mathbf{while} \ \mathbf{Wn} \ \mathbf{T} \ \mathbf{do} \ \text{reL}(\mathbf{P}) \ \mathbf{end} \mid \beta[\mathbf{Wn} \Rightarrow \mathbf{R}], \square)}$$

where $\mathbf{R} = \mathbf{while} \ \mathbf{Wn} \ \mathbf{b} \ \mathbf{do} \ \text{reL}(\mathbf{P}) \ \mathbf{end}$

Any condition evaluation that is not the first or the last ([W4]) requires the loop to be defined within the while environment ($\beta(\mathbf{Wn}) = \text{def}$) and to have a condition that evaluates to true (via $(\mathbf{b} \mid \beta, \square) \hookrightarrow_{\mathbf{b}}^* (\mathbf{T} \mid \beta, \square)$). The loop body is again renamed ($\text{reL}(\mathbf{P})$) and reflected into the procedure environment ($\beta[\mathbf{Wn} \Rightarrow \mathbf{R}]$).

$$[\text{W4}] \quad \frac{\beta(\mathbf{Wn}) = \text{def} \quad (\mathbf{b} \mid \beta, \square) \hookrightarrow_{\mathbf{b}}^* (\mathbf{T} \mid \beta, \square)}{(\mathbf{while} \ \mathbf{Wn} \ \mathbf{b} \ \mathbf{do} \ \mathbf{P} \ \mathbf{end} \mid \beta, \square) \hookrightarrow (\mathbf{while} \ \mathbf{Wn} \ \mathbf{T} \ \mathbf{do} \ \text{reL}(\mathbf{P}) \ \mathbf{end} \mid \beta[\mathbf{Wn} \Rightarrow \mathbf{R}], \square)}$$

where $\mathbf{R} = \mathbf{while} \ \mathbf{Wn} \ \mathbf{b} \ \mathbf{do} \ \text{reL}(\mathbf{P}) \ \mathbf{end}$

The loop body executes ([W5]) provided there is a valid execution step, until the body completes and reaches skip.

$$[\text{W5}] \quad \frac{\beta(\mathbf{Wn}) = \text{def} \quad (\mathbf{P} \mid \square) \hookrightarrow (\mathbf{P}' \mid \square')}{(\mathbf{while} \ \mathbf{Wn} \ \mathbf{T} \ \mathbf{do} \ \mathbf{P} \ \mathbf{end} \mid \square) \hookrightarrow (\mathbf{while} \ \mathbf{Wn} \ \mathbf{T} \ \mathbf{do} \ \mathbf{P}' \ \mathbf{end} \mid \square')}$$

After the completion of a loop body, the loop is reset to allow further execution ([W6]). If the loop body is skip, and the loop is defined within the while environment ($\beta(Wn) = \text{while } Wn \text{ b do } P \text{ end}$), the loop is reset to as in β .

$$[W6] \quad \frac{\beta(Wn) = \text{while } Wn \text{ b do } P \text{ end}}{(\text{while } Wn \text{ T do skip end} \mid \square) \hookrightarrow (\text{while } Wn \text{ b do } P \text{ end} \mid \square)}$$

2.4 Examples of Traditional Execution

We now show three small programs, and use inference trees (using transition rules defined above) to prove the existence of their first execution step. We begin with a conditional statement (with the true branch already executed) and a sequentially composed assignment. Let $P = \text{if } i1 \text{ T then skip else Q end; } X = 12$. The inference tree of the next execution step of P using the initial state \square follows.

$$\frac{\frac{}{(\text{if } i1 \text{ T then skip else Q end} \mid \square) \hookrightarrow (\text{skip} \mid \square)} [I4]}{(\text{if } i1 \text{ T then skip else Q end; } X = 12 \mid \square) \hookrightarrow (\text{skip; } X = 12 \mid \square)} [S1]$$

The composition of the transition rules [S1] and [I4] prove the conclusion to exist. Since the conditional statement is not skip (meaning [S2] does not apply), the only applicable transition rule is [S1]. With the condition evaluated to true and the true branch being skip, the only valid transition is via the axiom [I4], which closes the conditional statement with no change to the program state.

Consider the parallel composition of sequentially composed, racing assignments, namely where $P = \text{par } \{ X = X + 3; Y = 5; \} \{ X = 10; Y = 2; \}$. Let X initially be 2 ($\sigma(\gamma(X)) = 2$). Our semantics of parallel composition allows the first step of this execution to originate either from the left side of the parallel ([P1]), or from the right ([P2]). The inference trees of each are now given, beginning with a step of the left side. We note that the transition rule [D1] uses atomic evaluation of the expression $X + 3$ (like big step semantics of arithmetic expression evaluation), using the sequence of transition rules [A5];[A1];[A2].

$$\frac{\frac{\frac{}{(\mathbf{X} \mid \sigma, \gamma, \square) \hookrightarrow_a (2 \mid \sigma, \gamma, \square)} [A1]}{(\mathbf{X} + 3 \mid \sigma, \gamma, \square) \hookrightarrow_a^* (5 \mid \sigma, \gamma, \square)} [A5];[A2]}{\frac{}{(\mathbf{X} = \mathbf{X} + 3 \mid \sigma, \gamma, \square) \hookrightarrow (\text{skip} \mid \sigma[1 \mapsto 5], \gamma, \square)} [D1]}{\frac{}{(\mathbf{X} = \mathbf{X} + 3; Y = 5 \mid \sigma, \gamma, \square) \hookrightarrow (\text{skip; } Y = 5 \mid \sigma[1 \mapsto 5], \gamma, \square)} [S1]}{(\mathbf{P} \mid \sigma, \gamma, \square) \hookrightarrow (\text{par } \{ \text{skip; } Y = 5; \} \{ X = 10; Y = 2; \} \mid \sigma[1 \mapsto 5], \gamma, \square)} [P1]$$

The final data store is $\sigma[1 \mapsto 5]$, where the memory location 1 is evaluated via the premise $\gamma(X) = 1$ of [D1] (and the old value 2 of X has been lost).

The inference tree for executions starting with a step of the right side follows. Let Y initially be 6. The assignment $Y = 10$ is performed via [D1] which in this case is a leaf, due to the expression already being a value and requiring no evaluation. The final data store is $\sigma[1 \mapsto 10]$ (losing 6), where 1 is from the premise of [D1].

$$\frac{\frac{\frac{\gamma(X) = 1}{(X = 10 \mid \sigma, \gamma, \square) \hookrightarrow (\text{skip} \mid \sigma[1 \mapsto 10], \gamma, \square)} \text{ [D1]}}{(X = 10; Y = 2; \mid \sigma, \gamma, \square) \hookrightarrow (\text{skip}; Y = 2; \mid \sigma[1 \mapsto 10], \gamma, \square)} \text{ [S1]}}{(P \mid \sigma, \gamma, \square) \hookrightarrow (\text{par } \{ X = X + 3; Y = 5; \} \{ \text{skip}; Y = 2; \} \mid \sigma[1 \mapsto 10], \gamma, \square)} \text{ [P2]}$$

Our final example is of a while loop body, containing a conditional statement already evaluated to false. The first execution step performs an assignment from within the false branch of this nested conditional. Let the original program $P = \text{while } w1 \text{ } T \text{ do if } i1 \text{ } F \text{ then } Q \text{ else } X = 4; R \text{ end end}$, and the intermediate program $P' = \text{if } i1 \text{ } F \text{ then } Q \text{ else } X = 4; R \text{ end}$. Let X initially be 0. The inference tree proving there exists a transition from P that first performs the assignment $X = 4$ follows.

$$\frac{\frac{\frac{\frac{\gamma(X) = 1}{(X = 4 \mid \sigma, \gamma, \square) \hookrightarrow (\text{skip} \mid \sigma[1 \mapsto 4], \gamma, \square)} \text{ [D1]}}{(X = 4; R \mid \sigma, \gamma, \square) \hookrightarrow (\text{skip}; R \mid \sigma[1 \mapsto 4], \gamma, \square)} \text{ [S1]}}{(P' \mid \sigma, \gamma, \square) \hookrightarrow (\text{if } i1 \text{ } F \text{ then } Q \text{ else skip}; R \text{ end} \mid \sigma[1 \mapsto 4], \gamma, \square)} \text{ [I3]}}{(P \mid \sigma, \gamma, \square) \hookrightarrow (\text{while } w1 \text{ } T \text{ do if } i1 \text{ } F \text{ then } Q \text{ else skip}; R \text{ end end} \mid \sigma[1 \mapsto 4], \gamma, \square)} \text{ [W5]}$$

As in the previous example, the assignment contains a value and therefore requires no evaluation (hence [D1] is a leaf). The location 1 is retrieved from the premise $(\gamma(X) = 1)$ of [D1]. The data store produced is $\sigma[1 \mapsto 4]$ (losing 0).

2.5 Conclusion

In this chapter we have described an imperative while language that contains a form of parallel composition, known as interleaving. We have described how the program state is represented using a series of environments, and have defined the operational semantics that interact with these environments and perform traditional (irreversible) execution of this programming language.

Chapter 3

Reversing an Imperative Concurrent While Language

This chapter presents an approach to adding reversibility to the concurrent imperative programming language described in Chapter 2. Some key challenges introduced by our parallel composition operator are addressed, before the process of creating two versions of an original program is discussed. One of these versions performs forward execution alongside saving any information needed for reversal, while the second uses this saved information to simulate reverse execution. Two operational semantics are given defining the execution of each of these two versions of an original (irreversible) program.

3.1 Parallel Composition Challenges

The programming language in Chapter 2 contains a form of parallel composition, where the execution of two (or more) programs can be interleaved. The parallel composition of the programs P and Q is written as either $P \text{ par } Q$, or as $\text{par } \{P\} \{Q\}$ in some examples. Each component program of the parallel composition, namely P and Q , is referred to as a side or a thread from this point on. An example parallel program is shown later in Figure 3.2.

Interleaving introduces the following two key challenges that must be considered and addressed prior to the discussion of reversibility, namely

1. Multiple distinct execution order (also referred to as *interleaving order*),
2. Atomicity of statements.

The order in which executions interleave is arbitrary, meaning multiple distinct interleaving orders exist. All statements within the same side of a parallel execute in program order (order of statements from within the source code with interleaved

nested parallels). The execution order between statements from different sides of a parallel is not fixed, and is determined arbitrarily at runtime. This is a challenge as each distinct interleaving order is not guaranteed to have the same behaviour as all others, and can therefore produce different program states. This order is important when considering *races*, where two or more parallel steps of an execution each modify (or one accesses) the *same memory location*. In such cases the outcome of the program is directly affected by the ordering of those statements (result of the race). Example 4 shows how execution changes depending on the interleaving order.

Example 4. (Different interleaving orders) Consider a conditional statement in parallel with an assignment, with a race on X . Let $X = 0$ and $Y = 0$ initially.

```

if i1 ( $X < 10$ ) then            $X = 12$ ;
     $Y = X + 3$ ;                par
else
     $Y = X - 1$ ;
end;
```

An execution (or interleaving) order that first performs the assignment $X = 12$ before the conditional results in a state such that $X = 12$ and $Y = 11$. In contrast, an execution order where the conditional statement executes first results in a state such that $X = 12$ and $Y = 3$. The value of Y is directly affected by the outcome of the race on the reading of and writing to X . ■

Ambiguous interleaving order impacts reversibility. Statements should be inverted in the reverse order in which they were performed forwards in order for the initial program state to be restored correctly. The single execution order of sequential programs means this is easy, a property not shared by parallel programs. An inverse interleaving order cannot be arbitrary as in the forward execution, and instead must be determined correctly. The reverse execution of the program from Example 4, must determine the correct branch to reverse. Section 3.2.2 contains examples of misbehaviour due to incorrect inverse interleaving orders.

Now consider the atomicity of statements. In high level programming languages, each statement execution is typically composed of several smaller steps. An assignment actually contains the evaluation of the arithmetic expression, the retrieval of the memory location bound to this variable and finally the assignment. Parallel programs allow interleaving at any point within the execution of a statement. Returning to the racing program in Example 4, let the evaluation of the conditional statement begin by reading the value of X . Consider an execution that first reads the value of X , namely 0, as part of the conditional statement evaluation, before

interleaving and performing the assignment $X = 12$. Then the execution returns to the conditional statement and completes by executing the true branch. At a high level it appears as though $X = 12$ was executed first, while the true branch of the conditional was then executed (which implies that $X = 12$ was not executed first). Therefore interleaving can lead to old values being used (result of a race).

3.2 Annotation

We now add reversibility to our concurrent while language introduced above. As described in Chapter 1, one possible approach to this is to simply save the entire program state at each intermediate point (after each step). This approach, named full checkpointing, can reverse a given program by restoring to the snapshot of the desired position. This typically requires a large amount of information to be saved, for example the contents of each memory location. Some such data may be redundant, as the previous step of an execution is unlikely to have modified every memory location. A possible relaxation is periodic checkpointing, where snapshots are saved less frequently and forward re-execution used to reach positions in between checkpoints. Incremental checkpointing reduces the amount of information that must be saved by recording only the changes made by each step, which are then accumulated to restore a desired program state (usually starting from the initial program state).

We choose not to use a checkpointing approach due to our desire to execute programs in reverse, and the fact that simulating step-by-step reversal of a program using checkpointing would require many restorations of snapshots and forward re-executions. Secondly, in a concurrent language any forward re-execution is not guaranteed to behave identically and would therefore need to be addressed.

A second potential approach is to use a reversible programming language such as Janus [97]. In order to provide a basis that can be used to add reversibility to frequently used irreversible languages such as C++, we begin with the irreversible language defined in the previous sections. Due to difficulties in automatically converting an irreversible program into an equivalent reversible program (written in a reversible language), we do not choose this approach.

As outlined by Perumalla in [63], and used within the Reverse C Compiler (RCC) [63, 11], we choose to use a *source to source translator* approach. The basis of this is to take an original (irreversible) program and automatically produce two modified versions of it. The first, named the *annotated version* and generated via a process called *annotation*, performs the expected forward execution and saves only information that would otherwise have been lost. Any lost data vital for reversal, named *reversal information*, is saved as each statement executes. The second ver-

sion, named the *inverted version* and produced using a process called *inversion*, uses the saved reversal information to undo the effects of each statement and to restore the original program state.

Both modified versions are forward executing programs, performed on traditional irreversible computers. The inverted version simulates reverse execution by beginning in the final program state and performing an inverted version of each statement (in reverse order). The reversal information saved during the annotated execution allows the effects of each statement to be reversed. Work including RCC [63], Backstroke [75] and others [12], use program instrumentation to insert commands into an original program that save or use the reversal information. As a result of difficulties associated with parallel composition and the interleaving of such statements, and the complications added to the proof of correctness, we do not instrument a program with such commands and instead defer the saving and using of all information to the operational semantics.

We aim to save as small amount of information as is needed, as reasonable memory usage is crucial for scalability. We note here that we could reduce this information further by allowing only constructive assignments as in Janus [97]. This would remove the need to save any old values of variables, but would require all expressions to be re-evaluated during the reverse execution (with a possible further time penalty incurred when converting an original program using destructive assignments into only increments or decrements). Some information lost is already not saved as it can be recovered from the program code. For example, we do not save the copy of a while loop prior to its removal from the while environment, as the entry can be recreated easily from the inverted version of the program.

We now describe the type of reversal information required, consisting of the following two parts.

1. Information lost via each type of statement as it executes, including crucial values that are overwritten etc. (see Section 3.2.1).
2. The *interleaving order* of the execution needed to correctly reverse the statements in the corresponding order (see Section 3.2.2).

3.2.1 Saving Lost Information

Each type of statement from the syntax in Figure 2.1 of Chapter 2 loses information as it executes. Prior to detailing the information lost by each, we first introduce the *auxiliary store* δ as the environment used to store it. As a result, we now update \square to be the tuple $(\sigma, \gamma, \beta, \delta)$ by abuse of notation. One design choice made here is to use an environment separate to the program state, as this aides the proof of

correctness by ensuring the act of saving reversal information does not change the behaviour of the program w.r.t the program state. The second design choice made is to store all information into stacks, a data structure ideal for reversibility due to their Last-In, First-Out (LIFO) nature. The reversal information saved for the final statement should be accessible immediately as this is the first to be inverted (top of the stack). A side effect of this is we must consider the order in which reversal information is saved for nested constructs, as is highlighted below. An example of an auxiliary store and the population of it is shown later in Example 8. We now consider each type of statement in turn.

Assignment

Assignments are destructive in nature, as the current value of the variable is lost when it is overwritten. Consider the statement $X = 2 + 8$ and a program state where X is initially 4. After this assignment, the final program state will be such that X is equal to 10. For correct reversal, the value of X must be restored from 10 to 4. However the value 4 has been lost and cannot be re-calculated from the statement itself. This can be avoided by saving the current value of the variable prior to an assignment, meaning the old value can now be retrieved during reversal. This old value is pushed onto a stack named X (named after the specific variable) within δ . All old values of the same variable are saved onto a single stack, ensuring all races can be replayed correctly.

Conditional Statement

Conditional statements allow branching, where evaluation of a condition determines which branch is executed. Correct reversal requires only the branch that was executed to be reversed, introducing the challenge of determining this.

Boolean expressions are not guaranteed to be invariant, where the execution of the branch is capable of altering the value of the boolean condition. This means evaluation of the same condition after the execution of the branch (in the final program state) may differ to the original evaluation. Consider the conditional statement

```

if i1 (X > 4) then
    X = 2;
else
    X = 13;
end;
```

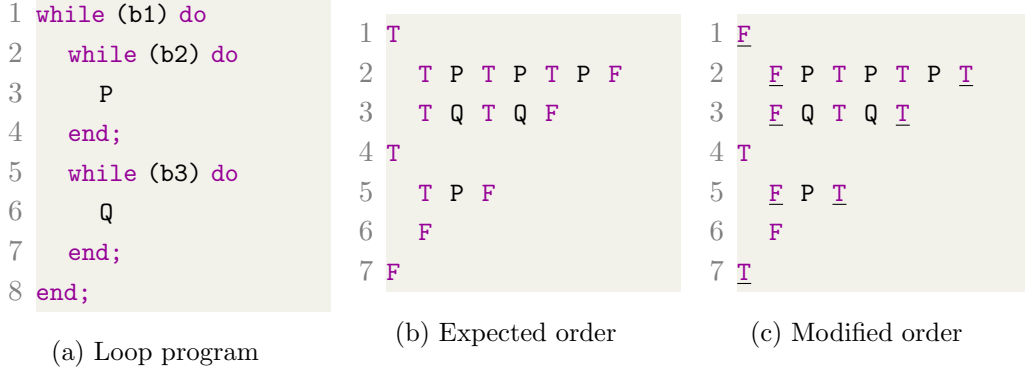


Figure 3.1: An example loop and both the expected and actual boolean sequence saved for reversal

and an initial program state where X is 6. During forward execution, the condition evaluates to true (since $6 > 4$), meaning the true branch is executed producing a final program state with X updated to 2. Now consider the inverse execution of this conditional statement. The condition $X > 4$ now evaluates to false (since $2 > 4$), leading to incorrect reversal as the false branch is now executed.

Therefore our approach is to save a boolean value indicating the branch that was executed. All boolean values for all conditional statements are pushed to the single stack named B on δ . The use of a stack here requires the boolean value for a conditional statement to be saved *after* any boolean values required for nested conditionals, ensuring these values are accessed from the stack in the correct order.

While Loop

While loops allow a varying number of iterations over a loop body. As for conditional statements, all conditions are not guaranteed to be invariant meaning the challenge of determining the number of iterations to reverse exists.

We choose not to implement a loop counter variable that increments for each iteration as this complicates the proof of correctness. Such an approach leads to an annotated execution having different behaviour w.r.t the program state when compared to the original. We therefore save a boolean value indicating the result of evaluating the condition each time. This builds a boolean sequence for each loop, capturing the number of iterations. Each element of this sequence is pushed to the stack W on δ . Consider the while loop

```

while w1 ( $X < 4$ ) do
   $X = X + 1$ ;
end;

```

and an initial program state where $X = 0$. This loop performs four iterations, and evaluates the condition five times (the four successful checks and the final unsuccessful one). The boolean sequence $T; T; T; T; F$ is then used during reversal to determine whether to iterate again.

We must again consider the order in which we save this sequence onto the stack named W . Figure 3.1(a) shows a loop containing two sequentially composed nested loops. Assume the outer loop performs two iterations, with the first containing three and two iterations of each nested loop respectively, and the second containing one and zero iterations of each nested loop respectively. First let the boolean sequence be generated in the intuitive manner of pushing a T for each successful iteration and finally an F . The sequence of booleans this generates is shown in the trace in Figure 3.1(b), where the outer vertical sequence represents the outer while loop, and each horizontal sequence represents each of the inner loops respectively. Assume these values have been pushed in this order to a stack, where the bottom of the trace is the top of the stack. When accessing this in reverse, the first boolean value retrieved is F . This causes confusion, as we know the outer loop performs two iterations, and ambiguity, as we would need to determine whether an F indicates the beginning of a loop with further iterations or a loop with zero iterations (as on line 6 of Figure 3.1(b)). In addition to this, the final element received is T which again leads to confusion as the loop must finish here.

This can be avoided by modifying the order in which the boolean values are saved. Specifically the opening T and the closing F are switched, meaning the first iteration of a loop therefore saves an F , while the final unsuccessful iteration saves a T . These two elements are then accessed in the expected order $T \dots F$ during reversal. Figure 3.1(c) shows the trace using the modified order, with all elements that have changed underlined. Finally, each element of the boolean sequence (except the beginning F) is saved after the execution of the loop body to ensure correct ordering between nested loops.

3.2.2 Interleaving Order and Identifiers

The specific execution path of a forward concurrent program is crucial for correct reversal, as the reverse execution must follow the corresponding inverted version. Interleaving during a reverse execution means a decision must be made about which statement to execute (invert) next. Problems may arise when the reverse execution follows a different path, as shown in Examples 5 and 6.

Example 5. (Incorrect statement order) Consider the program in Figure 3.2(a) that contains a race between two assignments to the same variable X (initially = 15. Assume an initial program state where $X = 15$. The forward execution of

<pre> 1 2 X = 10; par X = 5; 3 </pre>	<pre> 1 if i1 (X > 3) then if i2 (Y > 3) then 2 X = 10; Y = 12; 3 else par else 4 Y = 20; X = 3; 5 end end </pre>
(a) Racing assignments	(b) Conditional statements

Figure 3.2: Programs that reverse incorrectly under specific interleavings

this program follows the interleaving order where the left hand side assignment executes first, followed by the assignment on the right hand side. The left assignment overwrites the value 15, before the right assignment overwrites the value 10.

Now assume an inverse execution that does not follow the inverted order of the forward execution, and instead inverts the left followed by the right. Using stacks on the auxiliary store is sufficient to handle races, as the overwritten values can only ever be accessed in the correct order restoring X to 15. However the intermediate program structure, namely where $X = 10$ has not been performed (as its been reversed) and $X = 5$ has (waiting to be reversed), is incorrect since it did not occur during the forward execution (regardless of the value of X being correct). ■

Example 6. (Incorrect control flow) Another potential problem relates to control flow. Consider the program shown in Figure 3.2(b), and an execution where the left conditional executes the true branch (overwriting the value of X) and the right conditional executes the false branch (also overwriting the value of X). Let the forward execution first close the left conditional (pushing T to the stack B) and then close the right conditional (pushing F to the stack B).

Now consider the inverse execution that opens (inverse of closing) the left conditional first. Note that this is not in the order we expect. This means the top element of the stack B is used for the left conditional instead of the right, indicating the false branch of the left conditional must be inverted. This also means that T is used for the right conditional, indicating the true branch must be inverted. Each conditional is now trying to invert the incorrect branch. Note that both incorrect branches are assignments to Y that did not occur during the forward execution. This means there is no reversal information saved for these statements and the execution will halt (problems still arise even if reversal information does exist on the stack Y as it was not saved for these statements). ■

Each of these issues can be overcome by recording the order in which statements executed during the forward execution. This is achieved using *identifiers*, much like in the work [64, 68]. Identifiers, which are natural numbers, are used to capture the order in which statements execute. At each point that a decision is made during

```

1 while w1 (X > 3) do           Y = 20 [3];
2   X = X - 1 [1,4,7];         par   Z = Z + Y [6];
3 end [0,2,5,8];               Y = 12 [9];

```

Figure 3.3: Example use of identifiers to capture interleaving order

the forward execution, the next available identifier is associated with the statement that is executed. An annotated version of a program must therefore be capable of storing potentially many identifiers associated to each statement.

Identifiers are used in ascending order, typically beginning from 0. The next available identifier m is determined using the globally accessible and atomic function *next()*. Each subsequent call of this function returns the next available, namely $m+1$ and so on. This is essentially a counter and can be implemented as such. This single counter is shared by all concurrent sub-programs, ensuring we can capture the global interleaving order. Example 7 shows a concurrent program and the identifiers that capture the interleaving order.

Example 7. (Forward identifier use) Consider the annotated program shown in Figure 3.3, which contains a while loop in parallel with three assignment statements. We note here that `[]` represents a stack used to store identifiers for each statement (see Section 3.2). To aid this example, we represent all identifier stacks as part of the syntax. The actual implementation of these stacks is discussed in more detail in Section 3.2.3. Let this execution follow an interleaving order such that the first iteration and second condition evaluation of the loop happens first using identifiers 0, 1 and 2, before the first assignment of the right hand side is interleaved using identifier 3. Then the second iteration and third condition evaluation of the loop happens using identifiers 4 and 5, followed by the interleaved second assignment using identifier 6. Finally the while loop completes using identifiers 7 and 8, before the third and final assignment statement is interleaved using identifier 9. The order of these ten identifiers captures the interleaving order. ■

Each type of statement requires the use of identifiers. An assignment completes in a single step and requires the use of a single identifier each time. Conditional statements require two identifiers directly per execution, with one capturing the point at which the conditional was evaluated (referred to as opening) and one capturing when the conditional is closed. Any number of identifiers may be required during the execution of the appropriate branch. While loops require multiple identifiers per execution, with one to capture each time the condition is evaluated. Recall the while environment β and the fact that each loop is executed via a copy from Section 2.2 of Chapter 2. All identifiers are assigned to the specific copy within β , which we highlight in the following semantics using the function $refW(Wn, P)$. While

```

1 while w1 (X > 0) do
2   X = X - 1 [1,3,5];
3 end [0,2,4,6];
4 if i1 (Y > 4) then
5   Y = 10 [8];
6 else
7   Y = 20 [];
8 end [7,9];

```

(a) Executed annotated program

X	Y	B	W	WI
			(6,T)	
(5,1)			(4,T)	
(3,2)			(2,T)	
(1,3)	(8,6)	(9,T)	(0,F)	(6,seq)

(b) Auxiliary store where $\text{seq} = (1,3,5 \mid 0,2,4,6)$

Figure 3.4: Example program execution and its populated auxiliary store

loops ‘clean’ up after themselves by removing the specific copy from β , losing all identifiers associated with it. In order for reversal to be correct, all of these identifiers must be saved. Let \mathbb{C} be the set of sequences seq of identifiers. The function $\text{getAI}: \mathbb{P} \rightarrow \mathbb{C}$ takes a program P and returns all identifiers associated with statements within it. Note that this program must actually be an annotated program, written as $\mathbb{A}P$ and introduced in the coming sections. Written seq , a sequence contains groups of identifiers, where each group (a stack) contains identifiers associated to a specific statement and are separated using the symbol ‘|’. To save this sequence, we extend the auxiliary store to contain the stack WI , where such identifiers are recorded. Example 8 illustrates this.

The issue highlighted in Example 6 is addressed by using the identifiers to create a link between the reversal information and the statement to which it applies. This means that each step of an execution that saves any information must also use an identifier. The necessary reversal information for that statement is then indexed with this identifier, before being saved onto δ . This means all stacks on the auxiliary store δ are updated to contain pairs of the form (m, v) , where m is the identifier given to the statement and v is some information lost during that execution.

We now give an example of an auxiliary store. Let \mathbb{X} be the set of all variable names, \mathbb{K} be the set of identifiers, \mathbb{B} be the set $\{T, F\}$ of boolean values, \mathbb{Z} be the set of integers and \mathbb{C} be the set of identifier sequences retrieved from a loop copy prior to its removal. Then $\delta \in (\mathbb{X} \rightarrow (\mathbb{K} \times \mathbb{Z})) \cup (\mathbb{B} \rightarrow (\mathbb{K} \times \mathbb{B})) \cup (\mathbb{W} \rightarrow (\mathbb{K} \times \mathbb{B})) \cup (\mathbb{WI} \rightarrow (\mathbb{K} \times \mathbb{C}))$. We use the notation $\delta[e1 \rightarrow St]$ to represent pushing the element $e1$ to the stack St , and $\delta[St/St']$ to represent popping the head of stack St , leaving the tail St' .

Example 8. (Auxiliary store) Consider the program shown in Figure 3.4(a). Assume an initial program state such that $X = 3$ and $Y = 6$. This sequential program consists of a while loop that performs three iterations (since X is initially 3 and is decremented until it is less than or equal to 0). Following this, there is a conditional

$$\begin{aligned}
AP &::= \varepsilon \mid AS \mid AP; AP \mid AP \text{ par } AP \\
AS &::= \text{skip } I \mid X = E \ A \mid \text{if } In \ B \ \text{then } AP \ \text{else } AP \ \text{end } A \mid \\
&\quad \text{while } Wn \ B \ \text{do } AP \ \text{end } A \\
E &::= X \mid n \mid (E) \mid E \ Op \ E \\
B &::= T \mid F \mid \neg B \mid (B) \mid E == E \mid E > E \mid B \wedge B
\end{aligned}$$

Figure 3.5: Syntax of annotated versions of original programs

statement that will execute the true branch (as $Y = 6$). The identifiers 0 to 9 are assigned to the statements in the order in which they execute (sequential only here).

The auxiliary store containing all necessary reversal information for this execution is shown in Figure 3.4(b). The five necessary stacks are X and Y (a stack for each variable name), B (for conditionals), W (for loops) and WI (for loop identifiers). Each stack is now briefly explained, where elements are always pushed to and popped from the top. The variable X is assigned a new value three times (once for each iteration of the loop), meaning three corresponding pairs have been pushed to stack X . For example, the pair $(3, 2)$ means the statement with identifier 3 overwrote the value 2 (while setting it to 1). The variable Y is assigned a new value once, namely during the true branch via the statement with identifier 8. Therefore the stack Y contains the pair $(8, 6)$ capturing the overwritten value 6.

There is a single conditional statement that performs the true branch. This conditional is closed via the statement with identifier 9 (recall no information is saved for the opening of a conditional - identifier 7), meaning the stack B contains the pair $(9, T)$. There is a single while loop statement that performs three iterations. Therefore the sequence T, T, T, F is saved (alongside identifiers). For example the pair $(2, T)$ indicates the statement with identifier 2 evaluated the condition to T . The identifiers associated to the while loop are saved prior to its removal, represented using the sequence `seq` (in the form as explained above). ■

3.2.3 Annotation Function

In this section we introduce the process named annotation, which takes an original program and returns the annotated version. An annotated version must be capable of storing identifiers used to capture the interleaving order, meaning modifications to the syntax are required. These modifications concern only the interleaving order as all reversal information saving is deferred to the semantics (Section 3.3).

We take this opportunity to discuss the identifier stacks in more detail, and specifically focus on how these stacks can be implemented.

The majority of statements, excluding only block statements, require an identifier stack. Each initially empty identifier stack must be linked uniquely with a distinct

$$\begin{aligned}
ann(\varepsilon) &= \varepsilon \\
ann(S;P) &= a(S); ann(P) \\
ann(P \text{ par } Q) &= ann(P) \text{ par } ann(Q) \\
a(\text{skip}) &= \text{skip } A \\
a(X = e) &= X = e \ A \\
a(\text{if In } b \text{ then } P \text{ else } Q \text{ end}) &= \\
&\quad \text{if In } b \text{ then } ann(P) \text{ else } ann(Q) \text{ end } A \\
a(\text{while Wn } b \text{ do } P \text{ end}) &= \text{while Wn } b \text{ do } ann(P) \text{ end } A
\end{aligned}$$

Figure 3.6: Annotation function

statement, requiring a unique name. Let a stack name be of the form A_i , where i is an integer. Each statement can then be assigned an identifier stack, using the stack names in ascending order, such as A_1, A_2, A_3 etc. The syntax of all statements requiring an identifier stack is extended to now include a stack name. For example, the assignment statement $X = e$ is extended to include the stack name A_i such that $X = e \ A_i$.

In order for the identifier stacks to be persistent, and to outlive the execution of the forward program (as the semantics reach skip), the stacks must actually be stored separate to the program syntax. This can be achieved via a separate environment, where each uniquely named stack is globally accessible. This environment, named the identifier stack environment and represented as ψ , can be queried using a stack name. Given an identifier and the stack name to which it should be pushed, the stacks within this environment can be populated. We can use the notation $\psi(A_i)$ to access the stack A_i from ψ , and $\psi[k \rightarrow A_i]$ to represent pushing the identifier k to the stack A_i on ψ .

From this point, we aim to keep examples concise by having all stacks within the syntax of programs. This means that we no longer write just the stack name within the syntax, and instead write the entire stack. Example 7 displayed previously shows a program with identifier stacks within the syntax. We assume that all stacks are actually stored separately to the program code, and therefore all such stacks are persistent (by persistent, we mean that as a program is executed and each statement reaches skip, the stacks are not lost), which is crucial for reversal as we will explain in Section 3.5.1. However we omit the use of the stack identifier environment ψ , as this complicates the process of using identifiers that can be explained easier within the syntax. We will also return to this discussion when describing the operational semantics in Section 3.3, where we give two versions of the assignment rule, one that uses the identifier environment and one that does not.

<pre> 1 par { 2 while w1 (X < 3) do 3 X = X + 1; 4 Y = Y + X; 5 end; 6 } 7 { 8 if 11 (Y <= 1) then 9 Z = X + Y; 10 Y = Y - 1; 11 else 12 Z = Y - X; 13 Y = Y + 1; 14 end; 15 }</pre>	<pre> 1 par { 2 while w1 (X < 3) do 3 X = X + 1 []; 4 Y = Y + X []; 5 end []; 6 } 7 { 8 if 11 (Y <= 1) then 9 Z = X + Y []; 10 Y = Y - 1 []; 11 else 12 Z = Y - X []; 13 Y = Y + 1 []; 14 end []; 15 }</pre>
---	---

(a) Original program

(b) Annotated version

Figure 3.7: An original program and its corresponding annotated version

We use $[]$ to represent an empty stack, and $[2,1]$ to be a stack with the identifier 2 as its head. $A = m:A'$ represents a stack A that has m as its head and A' as its tail. Sequential and parallel composition do not require a stack, and instead use the stacks of their component statements. An annotated version of a program is of the syntax given in Figure 3.5, where \mathbb{AP} and \mathbb{AS} are the sets of annotated programs \mathbb{AP} and statements \mathbb{AS} respectively, and I indicates an optional identifier stack (e.g. hard-coded skip statements will not have one).

The function performing annotation, namely $ann: \mathbb{P} \rightarrow \mathbb{AP}$ (recalling \mathbb{P} is the set of original programs), is defined in Figure 3.6. This also contains the definition of the function $a: \mathbb{S} \rightarrow \mathbb{AS}$ (recalling \mathbb{S} is the set of original statements) that annotates each statement in turn.

Example 9 shows an original program and the corresponding annotated version. We return to this program when discussing inversion in a later section.

Example 9. (Annotated program) Consider the original program in Figure 3.7(a). The annotated version of this program contains all necessary identifier stacks associated with each appropriate statement which are initially empty. No saving commands are inserted as this is performed completely within the operational semantics. The annotated version of this original program is shown in Figure 3.7(b). ■

3.3 Forward Semantics of Annotated Programs

We now introduce the small step operational semantics of annotated forward execution, where both the forward execution and all reversal information saving is performed. As in Section 2.3 of Chapter 2, the least relation defined by the following set of transition rules is the transition relation \rightarrow . Each configuration used here is updated to consist of an annotated program AP or statement AS and state \square (now containing the auxiliary store δ). The reflective and transitive closure \rightarrow^* of \rightarrow indicates one configuration is reached from another via an arbitrary number of steps.

In order to capture the interleaving order via the use of identifiers, some transition rules require the use of an identifier while others do not. Those using identifiers are named *identifier steps*, while those that do not are called *skip steps*.

Definition 3.3.1. (Identifier step) An identifier step (rule) is a transition rule that uses an identifier. Each identifier step (rule) of the forward execution using an identifier m is represented using \xrightarrow{m} .

An example of such an identifier step is the rule for assignments, named [DA1] and shown in full below. This requires the use of an identifier as such a statement could be a component of a data race, where the order in which statements execute can affect the outcome of the program.

Definition 3.3.2. (Skip step) A skip step (rule) is a transition rule that does not use an identifier. Each skip step (rule) of the forward execution is represented as \rightarrow_s .

An example of a skip step is the rule for sequential composition, named [S2a] and shown below. This does not need an identifier as such a rule does not alter the program state and is therefore not dependent on the outcome of data races. Therefore we do not need to ensure a specific order is followed in the reverse execution.

Given that each transition rule is of one of the two possible kinds, we use $\xrightarrow{\circ}$ to represent either an identifier or skip step. Each transition rule [R] from Section 2.3 of Chapter 2 has a matching rule [Ra] here. The difference is that the following transition rules additionally use the auxiliary store δ , as described below. The semantics of arithmetic and boolean evaluation are unchanged from Chapter 2 and so are omitted.

Sequential Composition

Unchanged to that of Chapter 2, but with an optional identifier stack I .

$$\begin{array}{c}
 \text{[S1a]} \quad \frac{(\text{AS} \mid \square) \xrightarrow{\circ} (\text{AS}' \mid \square')}{(\text{AS}; \text{AP} \mid \square) \xrightarrow{\circ} (\text{AS}'; \text{AP} \mid \square')} \qquad \text{[S2a]} \quad \frac{}{(\text{skip } I; \text{AP} \mid \square) \rightarrow_s (\text{AP} \mid \square)}
 \end{array}$$

Parallel Composition

Unchanged to that of Chapter 2, but with an optional identifier stack I .

$$\begin{array}{l}
 \text{[P1a]} \quad \frac{(\text{AP} \mid \square) \xrightarrow{\circ} (\text{AP}' \mid \square')}{(\text{AP} \text{ par } \text{AQ} \mid \square) \xrightarrow{\circ} (\text{AP}' \text{ par } \text{AQ} \mid \square')} \qquad \text{[P2a]} \quad \frac{(\text{AQ} \mid \square) \xrightarrow{\circ} (\text{AQ}' \mid \square')}{(\text{AP} \text{ par } \text{AQ} \mid \square) \xrightarrow{\circ} (\text{AP} \text{ par } \text{AQ}' \mid \square')} \\
 \text{[P3a]} \quad \frac{}{(\text{skip } I_1 \text{ par skip } I_2 \mid \square) \rightarrow_s (\text{skip} \mid \square)}
 \end{array}$$

Assignment

Prior to defining the semantics of an assignment statement, we return to our discussion of identifier stacks from Section 3.2.3. As this is the first rule that will use an identifier and store this in an identifier stack, we first explain how identifier stacks are stored separately to the program code and how this is implemented. We then show a modified version of the rule that displays the stack within the code. Our subsequent rules will all follow this presentation style. Recall the introduction of the identifier stack environment ψ and the necessary notation to interact with it (Section 3.2.3). When using this environment, the syntax of an assignment statement contains a unique stack name. Consider the assignment statement $\mathbf{X} = \mathbf{e} \ \mathbf{Ai}$, where we assume the stack \mathbf{Ai} exists within the identifier stack environment ψ . The rule [D1aFull] below defines the semantics of an assignment.

$$\text{[D1aFull]} \quad \frac{\mathbf{m} = \text{next()} \quad (\mathbf{e} \mid \delta, \sigma, \gamma, \psi, \square) \hookrightarrow_{\mathbf{a}}^* (\mathbf{v} \mid \delta, \sigma, \psi, \gamma, \square) \quad \gamma(\mathbf{X}) = 1}{(\mathbf{X} = \mathbf{e} \ \mathbf{Ai} \mid \delta, \sigma, \gamma, \psi, \square) \xrightarrow{m} (\text{skip} \mid \delta[(\mathbf{m}, \sigma(1)) \rightarrow \mathbf{X}], \sigma[1 \mapsto \mathbf{v}], \gamma, \psi[\mathbf{m} \rightarrow \mathbf{Ai}], \square)}$$

This rule has demonstrated how the next identifier \mathbf{m} is stored into the identifier stack environment ψ (written $\psi[\mathbf{m} \rightarrow \mathbf{Ai}]$), and how \mathbf{m} is also stored alongside the old value of the variable (that will be lost) onto δ (written $\delta[(\mathbf{m}, \sigma(1)) \rightarrow \mathbf{X}]$). As in the corresponding rule in Chapter 2, the variable in question is evaluated to its memory location 1 which is updated to hold the new value \mathbf{v} within σ . We now display a second version of this rule, named [D1a], that uses our aforementioned simplification of representing stacks within the syntax (and ignores the identifier environment ψ). Note that the syntax of this statement now contains the stack \mathbf{Ai} (instead of just a stack name), which is treated as persistent by remaining after execution (alongside the `skip` statement). We therefore use \mathbf{A} to represent a stack from this point.

$$\text{[D1a]} \quad \frac{\mathbf{m} = \text{next()} \quad (\mathbf{e} \mid \delta, \sigma, \gamma, \square) \hookrightarrow_{\mathbf{a}}^* (\mathbf{v} \mid \delta, \sigma, \gamma, \square) \quad \gamma(\mathbf{X}) = 1}{(\mathbf{X} = \mathbf{e} \ \mathbf{A} \mid \delta, \sigma, \gamma, \square) \xrightarrow{m} (\text{skip } \mathbf{m}:\mathbf{A} \mid \delta[(\mathbf{m}, \sigma(1)) \rightarrow \mathbf{X}], \sigma[1 \mapsto \mathbf{v}], \gamma, \square)}$$

From this point on, all rules we define will follow this simplified format. We will write all stacks within the syntax of statements, and will omit all use of the identifier environment ψ .

Conditional Statements

Opening a conditional statement ([I1aT]/[I1aF]) inserts the identifier m (returned by $next()$) into the identifier stack ($m:A$) and saves no reversal information.

$$[I1aT] \frac{m = next() \quad (b \mid \square) \hookrightarrow_b^* (T \mid \square)}{(if \ In \ b \ then \ AP \ else \ AQ \ end \ A \mid \square) \xrightarrow{m} (if \ In \ T \ then \ AP \ else \ AQ \ end \ m:A \mid \square)}$$

$$[I1aF] \frac{m = next() \quad (b \mid \square) \hookrightarrow_b^* (F \mid \square)}{(if \ In \ b \ then \ AP \ else \ AQ \ end \ A \mid \square) \xrightarrow{m} (if \ In \ F \ then \ AP \ else \ AQ \ end \ m:A \mid \square)}$$

The true ([I2a]) or false branch ([I3a]) executes unchanged from Chapter 2. Recall from Section 3.2.1 that the result of condition evaluation is saved after the execution of the branch, meaning conditionals are static. Therefore the non-chosen branch and overall statement structure is retained throughout the execution.

$$[I2a] \frac{(AP \mid \square) \xrightarrow{\circ} (AP' \mid \square')}{(if \ In \ T \ then \ AP \ else \ AQ \ end \ A \mid \square) \xrightarrow{\circ} (if \ In \ T \ then \ AP' \ else \ AQ \ end \ A \mid \square')}$$

$$[I3a] \frac{(AQ \mid \square) \xrightarrow{\circ} (AQ' \mid \square')}{(if \ In \ F \ then \ AP \ else \ AQ \ end \ A \mid \square) \xrightarrow{\circ} (if \ In \ F \ then \ AP \ else \ AQ' \ end \ A \mid \square')}$$

Closing a conditional statement ([I4a]/[I5a]) behaves like in Chapter 2, but with the next identifier m (from $next()$) pushed into the identifier stack ($m:A$) and saved alongside a boolean value B that indicates which branch was executed ($\delta[(m,B) \rightarrow B]$).

$$[I4a] \frac{m = next()}{(if \ In \ T \ then \ skip \ I \ else \ AQ \ end \ A \mid \delta, \square) \xrightarrow{m} (skip \ m:A \mid \delta[(m,T) \rightarrow B], \square)}$$

$$[I5a] \frac{m = next()}{(if \ In \ F \ then \ AP \ else \ skip \ I \ end \ A \mid \delta, \square) \xrightarrow{m} (skip \ m:A \mid \delta[(m,F) \rightarrow B], \square)}$$

While Loops

A loop with zero iterations ([W1a]) uses the next identifier m , pushing this into the identifier stack ($m:A$) and saving it alongside the only element F of the boolean sequence of this loop ($\delta[(m,F) \rightarrow W]$).

$$[W1a] \frac{m = next() \quad \beta(Wn) = und \quad (b \mid \beta, \square) \hookrightarrow_b^* (F \mid \beta, \square)}{(while \ Wn \ b \ do \ AP \ end \ A \mid \delta, \beta, \square) \xrightarrow{m} (skip \ m:A \mid \delta[(m,F) \rightarrow W], \beta, \square)}$$

The last step of a loop ([W2a]) pushes the next identifier m to the identifier stack ($m:A$), and saves it alongside the final T of the boolean sequence ($\delta[(m,T) \rightarrow W]$). This identifier is also saved alongside the sequence of identifiers retrieved from this loop copy within β prior to its removal ($\delta[(m,C) \rightarrow WI]$ where $C = getAI(\beta(Wn))$).

$$[W2a] \frac{m = next() \quad \beta(Wn) = def \quad (b \mid \beta, \square) \hookrightarrow_b^* (F \mid \beta, \square)}{(while \ Wn \ b \ do \ AP \ end \ A \mid \delta, \beta, \square) \xrightarrow{m} (skip \ m:A \mid \delta[(m, T) \rightarrow W, (m, C) \rightarrow WI], \beta[Wn], \square)}$$

where $C = getAI(\beta(Wn))$

The first step of a loop ([W3a]) inserts the next identifier m into the identifier stack $(m:A)$ and saves it with the first F of the boolean sequence $(\delta[(m, F) \rightarrow W])$.

$$[W3a] \frac{m = next() \quad \beta(Wn) = und \quad (b \mid \beta, \square) \hookrightarrow_b^* (T \mid \beta, \square)}{(S \mid \delta, \beta, \square) \xrightarrow{m} (while \ Wn \ T \ do \ reL(AP) \ end \ m:A \mid \delta[(m, F) \rightarrow W], \beta[Wn \Rightarrow AR], \square)}$$

where $S = while \ Wn \ b \ do \ AP \ end \ A$ and $AR = while \ Wn \ b \ do \ reL(AP) \ end \ m:A$

The beginning of any middle iteration ([W4a]) uses the next identifier m and both inserts this into the identifier stack $(m:A)$ and saves it alongside the next element (must be a T) into the boolean sequence for this loop $(\delta[(m, T) \rightarrow W])$.

$$[W4a] \frac{m = next() \quad \beta(Wn) = S \quad (b \mid \beta, \square) \hookrightarrow_b^* (T \mid \beta, \square)}{(S \mid \delta, \beta, \square) \xrightarrow{m} (while \ Wn \ T \ do \ reL(AP) \ end \ m:A \mid \delta[(m, T) \rightarrow W], \beta[Wn \Rightarrow AR], \square)}$$

where $S = while \ Wn \ b \ do \ AP \ end \ A$ and $AR = while \ Wn \ b \ do \ reL(AP) \ end \ m:A$

Loop body execution ([W5a]) and loop resetting ([W6a]) match those from Chapter 2, since neither directly uses an identifier or saves reversal information. β'' is used to represent $\beta'[refW(Wn, AP')]$, which simply highlights that any identifiers assigned are reflected into the while environment.

$$[W5a] \frac{\beta(Wn) = def \quad (AP \mid \delta, \beta, \square) \xrightarrow{\circ} (AP' \mid \delta', \beta', \square')}{(while \ Wn \ T \ do \ AP \ end \ A \mid \delta, \beta, \square) \xrightarrow{\circ} (while \ Wn \ T \ do \ AP' \ end \ A \mid \delta', \beta'', \square')}$$

$$[W6a] \frac{\beta(Wn) = while \ Wn \ b \ do \ AP \ end \ A}{(while \ Wn \ T \ do \ skip \ I \ end \ A \mid \delta, \beta, \square) \rightarrow_s (while \ Wn \ b \ do \ AP \ end \ A \mid \delta, \beta, \square)}$$

3.4 Examples of Annotated Execution

We now return to the examples of traditional execution from Section 2.4 of Chapter 2 and highlight the differences of the matching annotated execution (noting the correspondence between each rule used). We begin with the sequential composition of a conditional statement ready to close and an assignment. The annotated version of this program is produced, such that $AP = \text{if } i1 \ T \ \text{then } skip \ I \ \text{else } Q \ \text{end } A;$ $X = 12 \ A'$. Let $AR = \text{if } i1 \ T \ \text{then } skip \ I \ \text{else } Q \ \text{end } A;$. Note the presence of the identifier stacks I (optional), A and A' , and recall the additional environment δ . The corresponding annotated execution is shown in the following inference tree.

$$\begin{array}{c}
\frac{\text{m} = \text{next}()}{(\text{AR} \mid \delta, \square) \xrightarrow{m} (\text{skip } \text{m}:\text{A} \mid \delta[(\text{m}, \text{T}) \rightarrow \text{B}], \square)} \text{[I4a]} \\
\frac{}{(\text{AP} \mid \delta, \square) \xrightarrow{m} (\text{skip } \text{m}:\text{A}; \text{X} = 12 \text{ A}' \mid \delta[(\text{m}, \text{T}) \rightarrow \text{B}], \square)} \text{[S1a]}
\end{array}$$

As can be seen above, this execution still does not alter the program state (\square), but does populate the auxiliary store δ , saving m and the boolean value T (the true branch was executed) onto the stack B . The final auxiliary store is $\delta[(\text{m}, \text{T}) \rightarrow \text{B}]$.

We return to our example of parallel composition, specifically the execution originating from the left side (the right follows accordingly). Recalling $\text{X} = 2$, let the annotated version $\text{AP} = \text{par } \{ \text{X} = \text{X} + 3 \text{ A}_1; \text{Y} = 5 \text{ A}_2; \} \{ \text{X} = 10 \text{ A}_3; \text{Y} = 2 \text{ A}_4; \}$, where each statement now contains an identifier stack. Further let $\text{AP}' = \text{par } \{ \text{skip } \text{m}:\text{A}_1; \text{Y} = 5 \text{ A}_2; \} \{ \text{Y} = 10 \text{ A}_3; \text{X} = 2 \text{ A}_4; \}$. The annotated execution of a step from the left side is shown in the inference tree below.

$$\begin{array}{c}
\frac{}{(\text{X} \mid \delta, \sigma, \gamma, \square) \hookrightarrow_{\text{a}} (2 \mid \delta, \sigma, \gamma, \square)} \text{[A1a]} \\
\frac{}{(\text{X} + 3 \mid \delta, \sigma, \gamma, \square) \hookrightarrow_{\text{a}}^* (5 \mid \delta, \sigma, \gamma, \square)} \text{[A5a];[A2a]} \\
\frac{}{(\text{X} = \text{X} + 3 \text{ A}_1; \mid \delta, \sigma, \gamma, \square) \xrightarrow{m} (\text{skip } \text{m}:\text{A}_1; \mid \delta', \sigma', \gamma, \square)} \text{[D1a]} \\
\frac{}{(\text{X} = \text{X} + 3 \text{ A}_1; \text{Y} = 5 \text{ A}_2; \mid \delta, \sigma, \gamma, \square) \xrightarrow{m} (\text{skip } \text{m}:\text{A}_1; \text{Y} = 5 \text{ A}_2; \mid \delta', \sigma', \gamma, \square)} \text{[S1a]} \\
\frac{}{(\text{AP} \mid \delta, \sigma, \gamma, \square) \xrightarrow{m} (\text{AP}' \mid \delta', \sigma', \gamma, \square)} \text{[P1a]}
\end{array}$$

The effect on the program state is unchanged to that of Section 2.4 of Chapter 2, namely $\sigma' = \sigma[1 \mapsto 5]$. One exception is the auxiliary store δ . The next identifier m is used by [D1a], and saved by [D1a] alongside the old value 2 of the variable X . Notice all evaluation still occurs, including of the variable X to the memory location 1 (via the premise $\gamma(\text{X}) = 1$ of [D1a] that is omitted). As such, $\delta' = \delta[(\text{m}, 2) \mapsto \text{X}]$.

Next we return to the example of an assignment statement nested within both a conditional statement and while loop, where X is initially 0. The annotated version of the original program is $\text{AP} = \text{while } \text{w1 } \text{T} \text{ do if } \text{i1 } \text{F} \text{ then } \text{Q} \text{ else } \text{X} = 4 \text{ A}_3; \text{R end } \text{A}_2 \text{ end } \text{A}_1$. The inference tree showing the corresponding annotated is now shown below, where we use $\text{AT} = \text{if } \text{i1 } \text{F} \text{ then } \text{Q} \text{ else } \text{X} = 4 \text{ A}_3; \text{R end } \text{A}_2$ and we use $\text{AT}' = \text{if } \text{i1 } \text{F} \text{ then } \text{Q} \text{ else skip } \text{m}:\text{A}_3; \text{R end } \text{A}_2$.

$$\begin{array}{c}
\frac{\text{m} = \text{next()} \quad \gamma(\text{X}) = 1}{(\text{X} = 4 \text{ A}_3 \mid \delta, \sigma, \gamma, \beta) \xrightarrow{m} (\text{skip } \text{m}:\text{A}_3 \mid \delta', \sigma', \gamma, \beta')} \text{[D1a]} \\
\frac{}{(\text{X} = 4 \text{ A}_3; \text{R} \mid \delta, \sigma, \gamma, \beta) \xrightarrow{m} (\text{skip } \text{m}:\text{A}_3; \text{R} \mid \delta', \sigma', \gamma, \beta')} \text{[S1a]} \\
\frac{}{(\text{AT} \mid \delta, \sigma, \gamma, \beta) \xrightarrow{m} (\text{if } \text{i1 } \text{F} \text{ then } \text{Q} \text{ else skip } \text{m}:\text{A}_3; \text{R end } \text{A}_2 \mid \delta', \sigma', \gamma, \beta')} \text{[I3a]} \\
\frac{}{(\text{AP} \mid \delta, \sigma, \gamma, \beta) \xrightarrow{m} (\text{while } \text{w } \text{T} \text{ do } \text{AT}' \text{ end } \text{A}_1 \mid \delta', \sigma', \gamma, \beta')} \text{[W5a]}
\end{array}$$

[D1a] is an axiom here as the expression requires no evaluation. The effect on the program state is equal to before, namely $\sigma' = \sigma[1 \mapsto 4]$ (with $\gamma(X) = 1$ evaluating X to memory location 1). The difference is that δ (and the identifier stack) is populated such that $\delta' = \delta[(m,0) \mapsto X]$. The identifier m is assigned to a statement of a copy within β such that $\beta' = \beta[\text{ref}W(Wn,AT')]$.

Example 10 briefly describes the forward execution of an example program using the semantics from Section 3.3. The executed version of this and the populated auxiliary store is also given, before the inverse version is generated.

Example 10. (Full annotated execution) Recall the original program and its annotated version from Figure 3.7 (page 38). We now consider an execution of this annotated program captured using the identifiers as assigned in Figure 3.10 (page 48). Assuming the initial program state where $X = 0$, $Y = 0$ and $Z = 0$, and an initial auxiliary store δ where all stacks are empty, we now briefly describe the forward annotated execution. Recall that stacks contains pairs of the form (identifier, value).

We begin with several steps of the left side of the parallel composition (the while loop) via [P1a]. We first evaluate the loop condition via [W3a], storing the identifier 0 and first element of the boolean sequence $F(0,F)$ to the stack W . We note the renamed copy of the loop body is generated here and inserted into β . Identifiers 1 and 2 are used to capture the execution of both assignments (X and Y updated to 1 via [D1a], both overwriting 0) within the loop body, each saving (1,0) and (2,0) to the stacks X and Y respectively. The while loop is then reset using [W6a], before re-evaluating the condition again to true via [W4a] using identifier 3. This inserts the pair (3,T) into the stack W . Next the entire loop body is executed again, saving identifiers 4 and 5 with the overwritten values 1 onto stacks X and Y respectively. The loop is again reset via [W6a], and the condition re-evaluated to true via [W4a] (saving the pair (6,T) to the stack W).

Interleaving occurs and we now perform a step of the right side via [P2a]. The condition is evaluated to true using identifier 7, with the reversal information saving deferred to after the execution of the branch. The first assignment of the true branch is then executed using identifier 8, updating Z to 5 and storing the pair (8,0) to the stack Z . We interleave again here and now perform the first assignment of the loop body on the left side, updating X to 3 and saving the pair (9,2) to the stack X . Interleaving occurs again as we now perform the final assignment from the true branch of the conditional, using identifier 10. This updates Y to 2 and saves the pair (10,3) to stack Y . The following step is of the while loop (left side) and performs the final assignment of the body using identifier 11. This sets Y to 5 and saves the pair (11,2) to the stack Y . The while loop is then reset via [W6a] and re-evaluated to false (as $X = 3$). This concludes the loop and saves the final element of the boolean sequence, namely T to the stack W (12,T). Further to this, all identifiers assigned to

the loop body (including its nested statements) are saved prior to its removal from β . This is a copy of each identifier stack used by the loop, represented in this work as a string. Therefore the stack **WI** contains the sequence **seq** = ‘1,4,9 | 2,5,11 | 0,3,6,12’ (where | separates the contents of different identifier stacks). At this point, the left side of the parallel is complete meaning execution can only continue via the right. The next step is to close the conditional statement via [I4a], which uses the identifier 13 and saves the pair (13,T) to indicate the true branch was executed. Finally, both sides are now complete (at skip), meaning [P3a] concludes the parallel and therefore the execution.

This execution produces a final state such that $X = 3$, $Y = 5$ and $Z = 5$. The final auxiliary store produced by this execution is now given, where **seq** is as above.

X	Y	Z	B	W	WI
	(11,2)			(12,T)	
(9,2)	(10,3)			(6,T)	
(4,1)	(5,1)			(3,T)	
(1,0)	(2,0)	(8,0)	(13,T)	(0,F)	(12,seq)

Figure 3.10(a) (page 48) displays the executed annotated program, where all identifier stacks are populated as explained above. We return to this inverted version in Section 3.7 where, after the introduction of the reverse semantics to follow, the reverse execution of this program is explained.

This example has been executed in our simulator (introduced in Chapter 7), demonstrating that the execution and auxiliary store described above is correct. ■

3.5 Inversion

With the annotated version and both the reversal information and identifiers required for reversal introduced, the next step is to define the process named inversion that is used to generate the second of our modified versions, namely the inverted version. This inverted version must support the use of identifiers to determine the inverse interleaving order, and reversal information to revert each statement. The use of each is deferred to the operational semantics of reverse execution (Section 3.6).

3.5.1 Inverse Interleaving Order

All interleaving decisions within a reverse execution are made using the identifiers assigned to statements during the matching forward execution. Recall the discussion from Section 3.2.3, where all identifier stacks can be stored separately, and are therefore available after the execution of the forward execution (where statements

reach skip). Under our simplification of including stacks within the syntax, we assume that these stacks are persistent in the same way as when using the identifier environment. This is crucial for correct reversal as the identifiers saved during the forward execution must be accessible during the reverse execution. Starting with the final identifier used during the forward execution, these are now used in descending order via the globally accessible and atomic function *previous()*. Whenever there is choice of next statement to execute (invert), only the statement with the appropriate identifier within its stack can be executed. Returning to the program in Example 7, we describe the process of using the identifiers in descending order in Example 11.

Example 11. (Inverse identifier use) Recall Example 7 of identifier use. The reverse execution begins with *previous()* = 9 (the final identifier used during forward execution). The first step is to decide which of the available steps to perform first, the inverted while loop (identifier 8) or the assignment $Y = 12$ (identifier 9). As our approach uses identifiers to determine the correct order in which to reverse the program, the only statement that should be executed next is that with identifier 9. At the corresponding position within the forward execution (namely at the end), the previously used identifier was 9. This must therefore be the first identifier consumed during the reversal. This process repeats until all statements have been inverted. ■

Using identifiers in this way follows *backtracking order*, where statements are reversed in exactly the inverted order of the forward execution [9]. An example of a possible relaxation of backtracking is *causal-consistent reversibility*, where independent steps of an execution can be reversed in any order [46, 67, 64]. Our approach currently only supports this relaxation for skip steps that have no effect on the program state. In the reverse semantics to follow, the three examples of such rules are [S2r], [P3r] and [W6r]. The matching rule of each from Section 3.3 does not use an identifier, meaning the reverse does not either. Such inverse steps may happen as soon as possible, or after the execution of other appropriate (identifier or skip) rules. This is acceptable as all skip steps do not modify the program state (Lemma 1 from Section 3.8). An inverse interleaving decision is now between the statement with the previous identifier (if possible) and any number of skip steps.

3.5.2 Using Reversal Information

With the decision of which statement to invert next having been made via identifiers, the next stage of reversal is to revert this statement. As shown in Section 3.2.1, each statement requires specific reversal information.

An assignment no longer evaluates the given arithmetic expression, and instead restores the given variable to the old value retrieved from the stack for this variable on δ . An inverse of a conditional statement does not evaluate the condition, and

$$\begin{aligned}
\text{IP} &::= \varepsilon \mid \text{IS} \mid \text{IP}; \text{IP} \mid \text{IP} \text{ par } \text{IP} \\
\text{IS} &::= \text{skip } I \mid X = E \text{ A} \mid \text{if In } B \text{ then IP else IP end A} \mid \\
&\quad \text{while Wn } B \text{ do IP end A} \\
E &::= X \mid n \mid (E) \mid E \text{ Op } E \\
B &::= T \mid F \mid \neg B \mid (B) \mid E == E \mid E > E \mid B \wedge B
\end{aligned}$$

Figure 3.8: Syntax of inverted versions of original programs

$$\begin{aligned}
\text{inv}(\varepsilon) &= \varepsilon \\
\text{inv}(\text{AS}; \text{AP}) &= \text{inv}(\text{AP}); i(\text{AS}) \\
\text{inv}(\text{AP} \text{ par } \text{AQ}) &= \text{inv}(\text{AP}) \text{ par } \text{inv}(\text{AQ}) \\
i(\text{skip } I) &= \text{skip } I \\
i(X = e \text{ A}) &= X = e \text{ A} \\
i(\text{if In } b \text{ then AP else AQ end A}) &= \\
&\quad \text{if In } b \text{ then } \text{inv}(\text{AP}) \text{ else } \text{inv}(\text{AQ}) \text{ end A} \\
i(\text{while Wn } b \text{ do AP end A}) &= \text{while Wn } b \text{ do } \text{inv}(\text{AP}) \text{ end A}
\end{aligned}$$

Figure 3.9: Inversion function

instead retrieves the next boolean value from the stack B on δ , typically saving time and space. This determines which branch to reverse. An inverse while loop begins by recreating the copy of it within the while environment β , where all identifier stacks of the copy of the body are re-populated using the sequence from stack WI on δ . This re-population is performed by the function $\text{setAI}: (\mathbb{IP} \times \mathbb{C}) \rightarrow \mathbb{IP}$, which takes the loop copy IP and sequence C , and inserts all identifiers of C into the statement stacks. As in the forward execution of a loop, all construct identifiers used within an inverted loop body are renamed. This is achieved using the matching function $\text{IreL}: \mathbb{IP} \rightarrow \mathbb{IP}$ that takes a program IP . This behaves correspondingly to $\text{reL}()$, but uses the version numbers in descending order, beginning with the values last used during the forward execution. The control flow of a loop is determined via the boolean values from W on δ , with the condition no longer evaluated and loops iterating until an F is retrieved.

3.5.3 Inversion Function

The inverted version of a given annotated program has the syntax shown in Figure 3.8, much like that of annotated programs but with \mathbb{IP} and \mathbb{IS} being the sets of inverted programs IP and statements IS respectively.

The inverted version is generated using the function $\text{inv}: \mathbb{AP} \rightarrow \mathbb{IP}$ (recalling that \mathbb{AP} is the set of annotated programs AP), shown in Figure 3.9. This also contains the definition of the function $i: \mathbb{AS} \rightarrow \mathbb{IS}$ (recalling that \mathbb{AS} is the set

<pre> 1 par { 2 while w1 (X < 3) do 3 X = X + 1 [1,4,9]; 4 Y = Y + X [2,5,11]; 5 end [0,3,6,12]; 6 } 7 { 8 if 11 (Y <= 1) then 9 Z = X + Y [8]; 10 Y = Y - 1 [10]; 11 else 12 Z = Y - X []; 13 Y = Y + 1 []; 14 end [7,13]; 15 }</pre>	<pre> 1 par { 2 while w1 (X > 3) do 3 Y = Y + X [2,5,11]; 4 X = X - 1 [1,4,9]; 5 end [0,3,6,12]; 6 } 7 { 8 if 11 (Y <= 1) then 9 Y = Y - 1 [10]; 10 Z = X + Y [8]; 11 else 12 Y = Y + 1 []; 13 Z = Y - X []; 14 end [7,13]; 15 }</pre>
---	---

(a) Executed annotated program

(b) Inverted version

Figure 3.10: An executed annotated program and its corresponding inverse version

of annotated statements **AS**), that inverts each statement in turn. This function is called on the statements of an executed annotated program in reverse order, with the results composed to invert the statement order. Note that the annotated execution is assumed to have been performed completely prior to inversion, meaning the auxiliary δ and all identifier stacks are populated accordingly.

3.6 Reverse Semantics of Inverted Programs

We now give our third small step operational semantics. As in Chapter 2, the least relation defined by the following set of transition rules is the transition relation \rightsquigarrow , representing reverse execution. Each configuration now consists of an inverted program **IP** and a state \square . The reflective and transitive closure \rightsquigarrow^* states one configuration can be reached from another in an arbitrary number of steps. The transition rules below that use an identifier are named identifier steps (represented as $\overset{m}{\rightsquigarrow}$), and those that do not are named skip steps (represented as \rightsquigarrow_s). A transition rule that may be either is written using $\overset{\circ}{\rightsquigarrow}$.

Each forward transition rule [Ra] from Section 3.3 has a corresponding reverse transition rule [Rr] defined below. We remark on the correlation between the rules of a forward and reverse execution in Section 3.8. No expressions or conditions are evaluated during a reverse execution and the transition rules of each are omitted.

Each identifier step defined below can only be applied if it is the next statement to reverse. This is enforced by retrieving the previously used identifier ($\mathbf{m} = \text{previous}()$)

and ensuring this is at the top of the appropriate identifier stack ($A = m:A'$). Each rule removes the identifier from this stack by continuing with A' .

Sequential Composition

Unchanged to that of Section 3.3.

$$\begin{array}{ll} \text{[S1r]} & \frac{(IS \mid \square) \overset{\circ}{\rightsquigarrow} (IS' \mid \square')}{(IS; IP \mid \square) \overset{\circ}{\rightsquigarrow} (IS'; IP \mid \square')} \quad \text{[S2r]} \quad \frac{}{(\text{skip } I; IP \mid \square) \rightsquigarrow_s (IP \mid \square)} \end{array}$$

Parallel Composition

Unchanged to that of Section 3.3.

$$\begin{array}{ll} \text{[P1r]} & \frac{(IP \mid \square) \overset{\circ}{\rightsquigarrow} (IP' \mid \square')}{(IP \text{ par } IQ \mid \square) \overset{\circ}{\rightsquigarrow} (IP' \text{ par } IQ \mid \square')} \quad \text{[P2r]} \quad \frac{(IQ \mid \square) \overset{\circ}{\rightsquigarrow} (IQ' \mid \square')}{(IP \text{ par } IQ \mid \square) \overset{\circ}{\rightsquigarrow} (IP \text{ par } IQ' \mid \square')} \\ \text{[P3r]} & \frac{}{(\text{skip } I_1 \text{ par skip } I_2 \mid \square) \rightsquigarrow_s (\text{skip} \mid \square)} \end{array}$$

Assignment

As in Section 3.3, we use this opportunity to discuss where the identifier stacks are actually stored. Recall our previous discussion of the identifier environment ψ , such that identifiers associated with statements are recorded into stacks within this environment. As before, we continue with our simplifying assumption of including all stacks within the syntax.

We start with the rule [D1rFull], corresponding to [D1aFull] shown in Section 3.3. Consider an assignment statement $X = e \text{ Ai}$, where Ai is a unique stack name. This assignment is reversed by restoring the variable to its previous value. Provided the previously used identifier m is in this statement's identifier stack ($\psi(Ai) = m:Ai'$), the assignment can be reversed by restoring the variable to the previous value v ($\sigma[1 \mapsto v]$) retrieved from stack X ($\delta(X) = (m, v):X'$). This reversal information is removed ($\delta[X/X']$), as is the identifier from Ai ($\psi[Ai/Ai']$).

$$\text{[D1rFull]} \quad \frac{A = m:A' \quad m = \text{previous}() \quad \delta(X) = (m, v):X' \quad \gamma(X) = 1 \quad \psi(Ai) = m:Ai'}{(X = e \text{ Ai} \mid \delta, \sigma, \gamma, \psi, \square) \overset{m}{\rightsquigarrow} (\text{skip} \mid \delta[X/X'], \sigma[1 \mapsto v], \gamma, \psi[Ai/Ai'], \square)}$$

We now show the simplified version of this rule named [D1r], which uses our assumption of stacks being within the syntax and omits the identifier environment ψ .

$$\text{[D1r]} \quad \frac{A = m:A' \quad m = \text{previous}() \quad \delta(X) = (m, v):X' \quad \gamma(X) = 1}{(X = e \text{ A} \mid \delta, \sigma, \gamma, \square) \overset{m}{\rightsquigarrow} (\text{skip } A' \mid \delta[X/X'], \sigma[1 \mapsto v], \gamma, \square)}$$

As was the case in our forward semantics, all further rules defined from this point will follow the simplified format, where all identifier stacks are contained within the syntax and the identifier stack environment is omitted.

Conditional Statements

Opening an inverted conditional statement (via [I1rT]/[I1rF]) reverses the closure, where the condition is not evaluated and instead the boolean result V retrieved from the stack B ($\delta(B) = (m, V) : B'$). This reversal information is removed ($\delta[B/B']$).

$$\begin{array}{c}
 \text{[I1rT]} \quad \frac{A = m : A' \quad m = \text{previous()} \quad \delta(B) = (m, T) : B'}{(S \mid \delta, \square) \xrightarrow{m} (\text{if In T then IP else IQ end } A' \mid \delta[B/B'], \square)} \\
 \text{where } S = \text{if In b then IP else IQ end A} \\
 \\
 \text{[I1rF]} \quad \frac{A = m : A' \quad m = \text{previous()} \quad \delta(B) = (m, F) : B'}{(S \mid \delta, \square) \xrightarrow{m} (\text{if In F then IP else IQ end } A' \mid \delta[B/B'], \square)} \\
 \text{where } S = \text{if In b then IP else IQ end A}
 \end{array}$$

The execution of the corresponding branch ([I2r]/[I3r]) behaves identically to those of Section 3.3 as neither directly use identifiers or reversal information.

$$\begin{array}{c}
 \text{[I2r]} \quad \frac{(IP \mid \square) \xrightarrow{\circ} (IP' \mid \square')}{(\text{if In T then IP else IQ end A} \mid \square) \xrightarrow{\circ} (\text{if In T then IP' else IQ end A} \mid \square')} \\
 \\
 \text{[I3r]} \quad \frac{(IQ \mid \square) \xrightarrow{\circ} (IQ' \mid \square')}{(\text{if In F then IP else IQ end A, } \square) \xrightarrow{\circ} (\text{if In F then IP else IQ' end A} \mid \square')}
 \end{array}$$

The closure of an inverse conditional statement ([I4r]/[I5r]) reverses the opening. The identifier m is used, but without any reversal information.

$$\begin{array}{c}
 \text{[I4r]} \quad \frac{A = m : A' \quad m = \text{previous()}}{(\text{if In T then skip I else IQ end A} \mid \square) \xrightarrow{m} (\text{skip } A' \mid \square)} \\
 \\
 \text{[I5r]} \quad \frac{A = m : A' \quad m = \text{previous()}}{(\text{if In F then IP else skip I end A} \mid \square) \xrightarrow{m} (\text{skip } A' \mid \square)}
 \end{array}$$

While Loops

A loop with zero iterations ([W1r]) is reversed in a single step. The stack W must contain a pair of the identifier m and an F ($\delta(W) = (m, F) : W'$) and the loop must be undefined within the while environment ($\beta(Wn) = \text{und}$). The loop terminates, removing the reversal information from stack W (via $\delta[W/W']$).

$$\text{[W1r]} \quad \frac{A = m : A' \quad m = \text{previous()} \quad \beta(Wn) = \text{und} \quad \delta(W) = (m, F) : W'}{(\text{while Wn b do IP end A} \mid \delta, \beta, \square) \xrightarrow{m} (\text{skip } A' \mid \delta[W/W'], \beta, \square)}$$

The final step of an inverted loop ([W2r]) is the reversal of its opening. If the loop is defined within the while environment ($\beta(\mathbf{Wn}) = \mathit{def}$) and the next element in the boolean sequence is an F ($\delta(\mathbf{W}) = (\mathbf{m}, \mathbf{F}) : \mathbf{W}'$), the loop statement terminates. The reversal information is removed ($\delta[\mathbf{W}/\mathbf{W}']$), as is the while environment entry ($\beta[\mathbf{Wn}]$).

$$[\mathbf{W2r}] \quad \frac{A = \mathbf{m} : \mathbf{A}' \quad \mathbf{m} = \mathit{previous}() \quad \beta(\mathbf{Wn}) = \mathit{def} \quad \delta(\mathbf{W}) = (\mathbf{m}, \mathbf{F}) : \mathbf{W}'}{(\mathbf{while} \ \mathbf{Wn} \ \mathbf{b} \ \mathbf{do} \ \mathbf{IP} \ \mathbf{end} \ \mathbf{A} \mid \delta, \beta, \square) \xrightarrow{m} (\mathbf{skip} \ \mathbf{A}' \mid \delta[\mathbf{W}/\mathbf{W}'], \beta[\mathbf{Wn}], \square)}$$

The first step of an inverted loop ([W3r]) reverses its closing. This loop must be undefined within the while environment ($\beta(\mathbf{Wn}) = \mathit{und}$), stack \mathbf{W} must contain \mathbf{T} as its head ($\delta(\mathbf{W}) = (\mathbf{m}, \mathbf{T}) : \mathbf{W}'$) and the stack \mathbf{WI} must have the identifier sequence \mathbf{C} as its head ($\delta(\mathbf{WI}) = (\mathbf{m}, \mathbf{C}) : \mathbf{WI}'$). These identifiers populate a renamed copy \mathbf{AR} of the loop ($\mathit{IreL}(\mathit{setAI}(\mathbf{IP}, \mathbf{C}))$), which is inserted into the while environment ($\beta[\mathbf{Wn} \Rightarrow \mathbf{AR}]$). This reversal information is removed ($\delta[\mathbf{W}/\mathbf{W}', \mathbf{WI}/\mathbf{WI}']$).

$$[\mathbf{W3r}] \quad \frac{A = \mathbf{m} : \mathbf{A}' \quad \mathbf{m} = \mathit{previous}() \quad \beta(\mathbf{Wn}) = \mathit{und} \quad \delta(\mathbf{W}) = (\mathbf{m}, \mathbf{T}) : \mathbf{W}' \quad \delta(\mathbf{WI}) = (\mathbf{m}, \mathbf{C}) : \mathbf{WI}'}{(\mathbf{S} \mid \delta, \beta, \square) \xrightarrow{m} (\mathbf{while} \ \mathbf{Wn} \ \mathbf{T} \ \mathbf{do} \ \mathbf{IP}' \ \mathbf{end} \ \mathbf{A}' \mid \delta[\mathbf{W}/\mathbf{W}', \mathbf{WI}/\mathbf{WI}'], \beta[\mathbf{Wn} \Rightarrow \mathbf{AR}], \square)}$$

where $\mathbf{S} = \mathbf{while} \ \mathbf{Wn} \ \mathbf{b} \ \mathbf{do} \ \mathbf{IP} \ \mathbf{end} \ \mathbf{A}$ and $\mathbf{IP}' = \mathit{IreL}(\mathit{setAI}(\mathbf{IP}, \mathbf{C}))$

and $\mathbf{AR} = \mathbf{while} \ \mathbf{Wn} \ \mathbf{T} \ \mathbf{do} \ \mathbf{IP}' \ \mathbf{end} \ \mathbf{A}'$

The beginning of any other iteration ([W4r]) requires the loop to be defined within the while environment ($\beta(\mathbf{Wn}) = \mathit{def}$) and the stack \mathbf{W} to contain \mathbf{T} as its head ($\delta(\mathbf{W}) = (\mathbf{m}, \mathbf{T}) : \mathbf{W}'$). The body is renamed ($\mathit{IreL}(\mathbf{IP})$) and reflected into the while environment ($\beta[\mathbf{Wn} \Rightarrow \mathbf{AR}]$), before all reversal information is removed ($\delta[\mathbf{W}/\mathbf{W}']$ and \mathbf{A}').

$$[\mathbf{W4r}] \quad \frac{A = \mathbf{m} : \mathbf{A}' \quad \mathbf{m} = \mathit{previous}() \quad \beta(\mathbf{Wn}) = \mathit{def} \quad \delta(\mathbf{W}) = (\mathbf{m}, \mathbf{T}) : \mathbf{W}'}{(\mathbf{S} \mid \delta, \beta, \square) \xrightarrow{m} (\mathbf{while} \ \mathbf{Wn} \ \mathbf{T} \ \mathbf{do} \ \mathit{IreL}(\mathbf{IP}) \ \mathbf{end} \ \mathbf{A}' \mid \delta[\mathbf{W}/\mathbf{W}'], \beta[\mathbf{Wn} \Rightarrow \mathbf{AR}], \square)}$$

where $\mathbf{S} = \mathbf{while} \ \mathbf{Wn} \ \mathbf{b} \ \mathbf{do} \ \mathbf{IP} \ \mathbf{end} \ \mathbf{A}$ and $\mathbf{AR} = \mathbf{while} \ \mathbf{Wn} \ \mathbf{b} \ \mathbf{do} \ \mathit{IreL}(\mathbf{IP}) \ \mathbf{end} \ \mathbf{A}'$

The execution of a loop body ([W5r]) and the resetting of a loop ([W6r]) corresponds to those for forward execution in Section 3.3. Again, β'' shows that identifiers are removed from the copy of the loop within β' , namely $\beta'' = \beta'[\mathit{refW}(\mathbf{Wn}, \mathbf{IP}')]$

$$[\mathbf{W5r}] \quad \frac{\beta(\mathbf{Wn}) = \mathit{def} \quad (\mathbf{IP} \mid \delta, \beta, \square) \xrightarrow{\circ} (\mathbf{IP}' \mid \delta', \beta', \square')}{(\mathbf{while} \ \mathbf{Wn} \ \mathbf{T} \ \mathbf{do} \ \mathbf{IP} \ \mathbf{end} \ \mathbf{A} \mid \delta, \beta, \square) \xrightarrow{\circ} (\mathbf{while} \ \mathbf{Wn} \ \mathbf{T} \ \mathbf{do} \ \mathbf{IP}' \ \mathbf{end} \ \mathbf{A} \mid \delta', \beta'', \square')}$$

$$[\mathbf{W6r}] \quad \frac{\beta(\mathbf{Wn}) = \mathbf{while} \ \mathbf{Wn} \ \mathbf{b} \ \mathbf{do} \ \mathbf{IP} \ \mathbf{end} \ \mathbf{A}}{(\mathbf{while} \ \mathbf{Wn} \ \mathbf{T} \ \mathbf{do} \ \mathbf{skip} \ \mathbf{I} \ \mathbf{end} \ \mathbf{A} \mid \delta, \square) \rightsquigarrow_s (\mathbf{while} \ \mathbf{Wn} \ \mathbf{b} \ \mathbf{do} \ \mathbf{IP} \ \mathbf{end} \ \mathbf{A} \mid \delta, \square)}$$

3.7 Examples of Reverse Execution

We return to our example executions from Section 2.4 (Chapter 2) and Section 3.4, and now briefly describe the reverse execution of each.

We begin with the sequentially composed conditional statement ready to close and an assignment statement. The inverted version contains only the conditional statement that must first be opened (condition evaluated). The assignment must have been previously reversed and therefore does not appear here. Let the inverted program $IP = \text{if } i1 \text{ } b \text{ then } R \text{ else } Q \text{ end } m:A$, where b and R are the original condition and true branch respectively (as the true branch is only reversed after the opening of the condition). The corresponding inverted execution is now shown.

$$\frac{A = m:A' \quad m = \text{previous}() \quad \delta(B) = (m, T) : B'}{(IP \mid \delta, \square) \xrightarrow{m} (\text{if } i1 \text{ } T \text{ then } R \text{ else } Q \text{ end } A \mid \delta[B/B'], \square)} \text{ [I1rT]}$$

The rule [I1rT] is a leaf here as there are no further transitions within its premises. The final auxiliary store is $\delta[B/B']$ (reversal information removed), while the state \square is unchanged. Note no evaluation of the condition, and no matching use of the rule [S1r] as in Section 3.4 as the conditional is no longer sequentially composed.

The parallel composition example is now considered, with only the example of a inference tree originating from the left side shown. With all subsequent statements (including all of those on the right side) reversed prior to the one that was executed first, let the inverted program $IP = \text{par } \{ X = X + 3 \ m:A_1; \} \{ \text{skip } I; \}$, recalling that (after the forward execution) $X = 5$ and stack X on the auxiliary delta δ has $(m, 2)$ as its head. The inference tree of this execution is shown below, again noting no matching use of [S1r] or any arithmetic expression evaluation rules.

$$\frac{\frac{A = m:A' \quad m = \text{previous}() \quad \delta(X) = (m, 2) : X' \quad \gamma(X) = 1}{(X = X + 3 \ m:A_1; \mid \delta, \sigma, \gamma, \square) \xrightarrow{m} (\text{skip } A_1; \mid \delta[X/X'], \sigma[1 \mapsto 2], \gamma, \square)} \text{ [D1r]}}{(IP \mid \delta, \sigma, \gamma, \square) \xrightarrow{m} (\text{par } \{ \text{skip } A_1; \} \{ \text{skip } I; \} \mid \delta[X/X'], \sigma[1 \mapsto 2], \gamma, \square)} \text{ [P1r]}$$

The transition rule [D1r] is always a leaf, as no evaluation of the expression is required. The variable X is still evaluated to the location 1, and the old value 2 is retrieved from stack X on δ . The final auxiliary store is $\delta[X/X']$ (reversal information removed) and the final data store is $\sigma[1 \mapsto 2]$ (as before the forward execution).

Recall the assignment nested within a conditional statement and while loop example from Section 3.4. Since the program R occurs after this forward step, it must be reversed first and so does not appear here. Let the respective inverted program $IP = \text{while } w1 \text{ } T \text{ do if } i1 \text{ } F \text{ then } IQ \text{ else } X = 4 \ m:A_3 \text{ end } A_2 \text{ end } A_1$. The inference tree follows, where $IP' = \text{while } w \text{ } T \text{ do if } i1 \text{ } F \text{ then } IQ \text{ else skip } A_3$

end A_2 end A_1 and with $IT = \text{if } i1 \text{ F then } IQ \text{ else } X = 4 \text{ m:A}_3 \text{ end } A_2$. The environments (after the forward execution) are such that $\sigma' = \sigma[1 \mapsto 4]$ and that stack X on δ has $(m, 0)$ as its head.

$$\begin{array}{c}
 \frac{A = m:A' \quad m = \text{previous()} \quad \delta(X) = (m, v):X' \quad \gamma(X) = 1}{(X = 4 \text{ m:A}_3 \mid \delta, \sigma, \gamma, \square) \xrightarrow{m} (\text{skip } A_3 \mid \delta[X/X'], \sigma[1 \mapsto 0], \gamma, \square)} \text{ [D1r]} \\
 \frac{}{(IT \mid \delta, \sigma, \gamma, \square) \xrightarrow{m} (\text{if } i1 \text{ F then } IQ \text{ else skip } A_3 \text{ end } A_2 \mid \delta[X/X'], \sigma[1 \mapsto 0], \gamma, \square)} \text{ [I3r]} \\
 \frac{}{(IP \mid \delta, \sigma, \gamma, \square) \xrightarrow{m} (IP' \mid \delta[X/X'], \sigma[1 \mapsto 0], \gamma, \square)} \text{ [W5r]}
 \end{array}$$

As above, [D1r] is a leaf as no evaluation of the expression is required. With reasoning like that of the previous example, the variable is restored to its previous value 0 and all reversal information (including the identifier) removed (hence the program state is correctly restored).

Finally, we return to Example 10 and the annotated execution and accompanying auxiliary store. Example 12 below briefly describes the respective reverse execution.

Example 12. (Full inverted execution) Recall the annotated execution explained in Example 10 and the inverted version given in Figure 3.10(b) (page 48). We now briefly describe the reverse execution. All interleaving decisions are determined using identifiers, where the initial value of $\text{previous}() = 13$. All reversal information is removed as it is used. The inverted version of our program is shown in Figure 3.10(b), where the overall statement order is inverted, and all identifier stacks are populated as in the executed version. Recall $X = 3$, $Y = 5$ and $Z = 5$.

The first step of the reverse execution must be of the conditional statement on the right side, as this contains the previously used identifier. This is via [I1rT], as the stack B contains the pair $(13, T)$ indicating the true branch must be reversed. Next we must interleave to find the identifier 12, which means we now open the while loop via [W3r] (as $(12, T)$ is at the top of the stack W). This re-creates the renamed version of the loop within β , using the sequence **seq** from the stack WI to populate the renamed copy with the appropriate identifiers (included in the loop within Figure 3.10(b) to help readability). The identifier 11 means we now perform the first assignment within the loop body, restoring Y to 2 (as $(11, 2)$ is at the top of the stack Y) via [D1r]. The next steps use identifiers 10, 9 and 8 to restore the variables Y , X and Z to old values retrieved from the appropriate stacks. Identifier 7 means we must close the conditional statement (which does not require any reversal information), resulting in the right side of the parallel reaching skip. From here, the reverse execution is essentially sequential, beginning with the reset of the loop via [W6r] (which could have occurred at any point after the statement using identifier 9), and then [W4r] to determine whether to iterate again (which we do as the stack

W has $(6, T)$ as its head). This process is then repeated to execute the loop body (identifiers 5 and 4), reset and begin another iteration (identifier 3) and finally the last loop body execution (identifiers 2 and 1). The final step here, using identifier 0, is to conclude the loop (and remove the entry from β) via $[W2r]$ as the stack W contains the pair $(0, F)$ as its head. Therefore the parallel then completes via $[P3r]$ (not using an identifier) and finishes the inverse execution.

Following the above execution closely shows that the final program state produced is such that $X = 0$, $Y = 0$ and $Z = 0$. Since no mappings exist in the while environment, we can conclude the program state has been correctly restored to as it was prior to the forward execution (Example 10). All reversal information has been used and removed from δ , meaning δ is also correctly restored to as it was prior to the forward execution, showing no garbage is produced.

As in Example 10, the inverted program has been executed in our simulator demonstrating the execution described above to be correct. ■

3.8 Correctness of Reversal

In this section we prove the approach described here to be correct. We name this *correct reversal* and formalise this in Definition 3.8.1, where a reverse execution is correct if the program state is correctly restored to as it was initially. We assume that from this point onwards, we consider only terminating programs.

Definition 3.8.1. (Correctness of reversal) Let AP be an annotated program, IP be the inverted version of AP and \square be the initial program state. Let AP execute on \square , namely $(AP \mid \square) \xrightarrow{\circ}^* (\text{skip } I \mid \square')$, producing some I and program state \square' . The reversal of AP is correct, with respect to \square' and \square , if there exists a reverse execution of IP such that $(IP \mid \square') \xrightarrow{\circ}^* (\text{skip } I' \mid \square)$, for some I' .

In order to prove this property for our simple concurrent while language programs, several lemmas are required. We first remark that requiring program states to be equal (in Definition 3.8.1) imposes unnecessary constraints. For example, underlying memory locations used within the variable state may differ provided all values of variables match, and a while environment from a reverse execution must contain all mappings of the corresponding forward environment but with one program the inverted version of the other. We therefore introduce *equivalence* between program states later in Chapter 6.

Our first smaller result relates to relaxing backtracking order of skip steps. This is dependent on no skip step altering the program state, as stated in Lemma 1.

Lemma 1. (Skip steps do not change program state) Let AP be an annotated program, IP be its inverted version and \square be the tuple $(\sigma, \gamma, \beta, \delta)$ of initial program

state environments. For all forward executions $(\mathbf{AP} \mid \square) \rightarrow_s (\mathbf{AP}'' \mid \square'')$ for some \mathbf{AP}'' and \square'' via a skip step [Ra], the program state is unchanged, namely $\square'' = \square$. For all reverse executions $(\mathbf{IP} \mid \square) \rightsquigarrow_s (\mathbf{IP}' \mid \square'')$, for some \mathbf{IP}'' and \square'' , via a skip step [Rr], the program state is unchanged, namely $\square'' = \square$.

Proof. This proof is by induction on the height of an inference tree of a skip transition. We begin by considering all bases cases, namely any inference trees of height 1. The three examples of skip steps present within our transition rules from Section 3.3 are [S2a], [P3a] and [W6a]. Each of these rules is a leaf (is either an axiom or contains no further transitions as premises), meaning they complete in a single step (height 1). As shown in Section 3.3, each of these three rules does not change the program state in any way, as required. With the three base cases shown to be valid, we now consider the six possible inductive cases. We assume Lemma 1 holds for an inference tree of height n such that $n \geq 1$, and next we prove it to be valid for an inference tree of height $n + 1$.

A skip transition (step) with the inference tree of height $n + 1$ must use at its root an instance of one of the six skip rules, namely [S1a], [P1a], [P2a], [I2a], [I3a] or [W5a]. As we have explained in Section 3.3, none of these rules change a program state. If we remove the root from this inference tree, we obtain a smaller inference tree of height n , proving valid a skip transition (step). By induction this transition does not change a program state, hence overall the original transition does not change program state.

Since skip transition rules are in one to one correspondence with reverse skip transition rules, the proof of the second half of Lemma 1 follows correspondingly to the proof of the first part. \square

To follow backtracking order in all other cases, identifiers must be used in ascending order throughout a forward execution, shown in Lemma 2, and in descending order within the reverse execution, shown in Lemma 3.

Lemma 2. (Forward order of identifiers) Let \mathbf{AP} and \mathbf{AQ} be annotated programs, \square be the set of environments and δ be an auxiliary store. If $(\mathbf{AP} \mid \square, \delta) \xrightarrow{\circ}^* (\mathbf{AP}' \mid \square', \delta') \xrightarrow{n} (\mathbf{AQ} \mid \square'', \delta'') \rightarrow^* (\mathbf{AQ}' \mid \square''', \delta''') \xrightarrow{m} (\mathbf{AQ}'' \mid \square''', \delta''')$, and the computation $(\mathbf{AQ} \mid \square'', \delta'') \rightarrow^* (\mathbf{AQ}' \mid \square''', \delta''')$ has no identifier steps, then $m = n + 1$.

Proof. The order of identifiers used during execution is maintained using the globally accessible and atomic function $next()$. The program \mathbf{AP} begins with any number of steps, before an identifier step occurs using the next available identifier n , while simultaneously incrementing $next()$ by one to $n + 1$. At some point (after any number of skip steps that do not use $next()$) the next identifier step occurs using the current value of $next()$ which is $n + 1$. Hence, $m = n + 1$. \square

Table 3.1: Each forward and matching inverse identifier rule

Fwd	Inv	Fwd	Inv
[S1a]	[S1r]	[I4a]	[I1rT]
[P1a]	[P1r]	[I5a]	[I1rF]
[P2a]	[P2r]	[W1a]	[W1r]
[D1a]	[D1r]	[W2a]	[W3r]
[I1aT]	[I4r]	[W3a]	[W2r]
[I1aF]	[I5r]	[W4a]	[W4r]
[I2a]	[I2r]	[W5a]	[W5r]
[I3a]	[I3r]		

Lemma 3. (Reverse order of identifiers) Let \mathbf{AP} and \mathbf{AQ} be the annotated versions producing the executed versions \mathbf{AP}' and \mathbf{AQ}' respectively, and \mathbf{IP} and \mathbf{IQ} be the inverted versions $inv(\mathbf{AP}')$ and $inv(\mathbf{AQ}')$ respectively. Further let \square be the set of all environments and δ be the auxiliary store. If $(\mathbf{IP} \mid \square, \delta) \xrightarrow{\circ}^* (\mathbf{IP}' \mid \square', \delta') \xrightarrow{n} (\mathbf{IQ} \mid \square'', \delta'') \rightsquigarrow^* (\mathbf{IQ}' \mid \square''', \delta''') \xrightarrow{m} (\mathbf{IQ}'' \mid \square''', \delta''')$, and provided the computation $(\mathbf{IQ} \mid \square'', \delta'') \rightsquigarrow^* (\mathbf{IQ}' \mid \square''', \delta''')$ has no identifier steps, then $m = n - 1$.

Proof. Correspondingly to that of Lemma 2, using the function *previous()* and the transition relation \rightsquigarrow in place of \rightarrow . \square

In order to match a forward and reverse execution at each stage, we state that each forward identifier rule (step) has a *corresponding inverse rule* (step) in Definition 3.8.2. We remark on the possible mismatch of skip steps in Chapter 5.

Definition 3.8.2. (Corresponding inverse identifier step) Each forward identifier rule (Section 3.3) has a matching inverse identifier rule (Section 3.6). Each forward rule $[\mathbf{Ra}]$ has the corresponding inverse rule $[\mathbf{Rr}]$, written as $[\mathbf{Ra}]^{-1} = [\mathbf{Rr}]$, where the effects of $[\mathbf{Ra}]$ on the program state is inverted by $[\mathbf{Rr}]$. For example, rule $[\mathbf{D1r}]$ is the corresponding inverse rule of $[\mathbf{D1a}]$. Each matching pair is shown in Table 3.1.

With the appropriate smaller results shown above, we now return to the discussion of proving correct reversal. This focuses on the following two properties:

1. The process of saving reversal information does not change the behaviour of the program (w.r.t the program state), while saving all required information.
2. The inverted version restores the program state to exactly as it was prior to the forward execution. The auxiliary store is also restored to as it was originally, meaning no *garbage* is produced.

The proof of these properties is omitted here as it is given in Chapter 6 for a programming language that contains this concurrent while language.

3.9 Conclusion

We have introduced a method of reversing programs written in the imperative concurrent language shown in Chapter 2. We have described the reversal information required and the auxiliary store used to record it, and the use of identifiers to capture an interleaving order. Annotation has been introduced as the process that generates the annotated version, a modified forwards version that is capable of storing identifiers. An annotated version executes as defined by the first operational semantics, where both the forward execution and all necessary reversal information saving occurs. We have introduced inversion as the function used to generate the inverted version of a program, that uses both the identifiers and reversal information to perform reverse execution. The second semantics define this behaviour, before the proof of both annotation and inversion is outlined. A more general version of these proofs are shown in Chapter 6.

Chapter 4

Reversing an Imperative Concurrent Language with Blocks, Local Variables and Procedures

In Chapter 2 we have introduced our imperative concurrent while language. The syntax of this language and the operational semantics of traditional (forward-only) execution have been stated. Reversibility has then been added to this while language in Chapter 3, using one modified version of a program to save all lost information and to use identifiers to capture the interleaving order, while another modified version uses this saved information to reverse the original execution.

In this chapter, we increase the complexity of the language through a series of additions to the programming language syntax. Block statements are introduced, along with variable and procedure declaration statements and recursion-enabled procedure call statements. Each addition results in necessary extensions to the notion of program state, as well as to the reversal information saved and the use of identifiers to capture the interleaving order. The three operational semantics defined in Chapters 2 and 3 (traditional, forwards with reversal information saving and reverse) are now extended to include all of the additional program constructs.

4.1 Motivating Example

The language introduced in Chapter 2 restricts the complexity of programs that can be written in it. There is no easy way to reuse code (outside of a while loop) which results in any program requiring code reuse to contain multiple copies of it. This increases the length of such programs and decreases the readability. There is no capability for different parts of the program to have different scopes, meaning all variables are global and must be freshly named (cannot reuse names).

```

1 start = 23;
2 output = -1;
3 begin b1.0
4   var initial = -1;
5   var result = -1;
6   proc p1.0 odd is
7     if i1.0 (initial == 0) then
8       result = 0;
9     else
10      initial = initial - 1;
11      call c1.0 even;
12    end;
13  end;
14  proc p2.0 even is
15    if i2.0 (initial == 0) then
16      result = 1;
17    else
18      initial = initial - 1;
19      call c2.0 odd;
20    end;
21  end;
22  initial = start;
23  call c3.0 even;
24  output = result;
25 end;

```

Figure 4.1: Program determining whether a number is odd or even using mutually recursive procedure calls. Note that variable and procedure removal statements are introduced later and so are omitted here.

Consider the program shown in Figure 4.1 that makes use of code reuse via procedures. Note the extended syntax of this is explained in the coming sections, where procedures are declared (lines 6–13 and 14–21) and then called (for example line 23). This example uses the *mutually recursive* (each recursively calls the other) procedures **even** and **odd** to determine whether a given number (**start** from line 1) is odd or even. Reproducing the same example (using the same underlying approach of conditional statements only) in our original language would require multiple copies of the same pieces of code, namely the conditional statements. In order to support such programs, we now introduce an extended version of our programming language.

4.2 Concurrent Language and Program State

Beginning with the imperative concurrent while language from Chapter 2, this section describes the constructs introduced to this basis language in order to produce our extended imperative concurrent language. We move away from the while language and towards a more complex, real-world programming language. This is achieved by adding support for block statements, each of which can contain the declaration of local variables and procedures, potentially recursive procedure call statements and finally removal statements.

4.2.1 Syntax of Programs

The syntax of our extended language is shown in Figure 4.2. Let \mathbb{P} be the set of programs P , and \mathbb{S} be the set of statements S . Further let X denote a variable, E be an arithmetic expression and B be a boolean condition. Finally, let \mathbb{Z} be the set

$$\begin{aligned}
P &::= \varepsilon \mid S \mid P; P \mid P \text{ par } P \\
S &::= \text{skip} \mid X = E \text{ pa} \mid \text{if } In \ B \text{ then } P \text{ else } Q \text{ end pa} \mid \\
&\quad \text{while } Wn \ B \text{ do } P \text{ end pa} \mid \text{begin } Bn \ BB \text{ end} \mid \\
&\quad \text{call } Cn \ n \text{ pa} \mid \text{runc } Cn \ P \text{ end} \\
BB &::= DV; DP; P; RP; RV \\
E &::= X \mid n \mid (E) \mid E \text{ Op } E \\
B &::= T \mid F \mid \neg B \mid (B) \mid E == E \mid E > E \mid B \wedge B \\
\\
DV &::= \varepsilon \mid \text{var } X = E \text{ pa}; DV & DP &::= \varepsilon \mid \text{proc } Pn \ n \text{ is } P \text{ end pa}; DP \\
RV &::= \varepsilon \mid \text{remove } X = E \text{ pa}; RV & RP &::= \varepsilon \mid \text{remove } Pn \ n \text{ is } P \text{ end pa}; RP
\end{aligned}$$

Figure 4.2: Syntax of our extended imperative concurrent language

of integers, where a number n is such that $n \in \mathbb{Z}$, pa be a statement path (that is introduced in Section 4.2.2), and In , Wn , Bn , Pn and Cn are construct identifiers.

In order for programs written in this syntax to be defined as valid, we make key assumptions regarding the syntax.

1. For each declaration statement within a block body, there is a corresponding removal statement at the end of the same block body (each block is ‘cleaned’ prior to its completion). Variables and procedures are removed in the inverse of the order in which they are declared.
2. Procedure call statements must only appear within block bodies.
3. A procedure must have been declared prior to it being called.
4. The **runc** construct cannot appear in original programs (and is only introduced via execution of a **call** statement).
5. All variables and procedures declared directly within the same block must be named uniquely with respect to each other.
6. All memory locations required for variable declarations exist and are initialised to 0. All global variables exist and are initialised to 0.

4.2.2 Program State Environments

The extended programming language requires an updated representation of program state. Modifications are needed to the variable environment in order to handle local variables, while the procedure environment is introduced to support procedures.

Variable State

Recall the variable environment γ and the data store σ introduced in Chapter 2, and how each is used to represent all variables and their values. A limitation of the implementation shown in the previous chapter is that only global variables are supported, meaning all variables are uniquely named. Returning to our extended language, γ and σ as currently defined are insufficient to support local variables. Each version of a given variable name must be mapped unambiguously to a unique memory location within the variable environment γ . The addition of versions of variables breaks the uniqueness of variable names and therefore introduces ambiguity into the current definition of γ .

We observe from the syntax of our language that local variables can only be defined within block statements, and recall the assumption that all local variables within a specific block statement are uniquely named. This allows all local variables with the same name to be distinguished provided we are aware of the block in which it was declared. In order to use this approach each block must be uniquely identifiable, a feature supported by giving each block statement a construct identifier (written as **Bn**). The variable environment is therefore extended to support local variables by mapping a pair of a variable name and the block name in which it was declared to a memory location. This approach provides a one-to-one mapping between versions of variables to memory locations, as required. Let **Loc** be as above, **Var** be the set of variable names, **Cn** be the set of construct identifiers and **Bn** be the set of those assigned to blocks. Then the variable environment is such that $\gamma : (\mathbf{Var} \times \mathbf{Bn}) \rightarrow \mathbf{Loc}$. The final consideration is global variables, which are assumed to exist prior to an execution and are not declared. All global variables will use the empty block name λ . The variable environment γ is manipulated using the following commands.

- $\gamma[(X, \mathbf{Bn}) \Rightarrow 1]$ - creates an entry into the variable environment γ , mapping a variable **X** that declared within a block **Bn** to the memory location 1.
- $\gamma[(X, \mathbf{Bn})]$ - removes the mapping from γ between a variable **X** that was declared in a block **Bn** and some memory location.
- $evalV(X, \mathbf{pa}, \gamma)$ - the function $evalV: (\mathbf{Var} \times \mathbf{Pa} \times \gamma) \rightarrow \mathbf{Bn}$ takes a variable name **X**, path **pa** and a variable environment γ , and returns the memory location associated with this version of the variable within γ .

The function $evalV()$ requires the correct block name to be provided in order to behave correctly. In order for a variable to be evaluated, we must first be able to determine the appropriate block name. This is achieved using *statement paths*.

```

1 X = 10 ( $\lambda$ , []);
2 begin b1.0
3   var X = 13 (b1.0, []);
4   begin b2.0
5     var Y = 6 (b2.0:b1.0, []);
6     Y = X + 2 (b2.0:b1.0, []);
7     begin b3.0
8       var X = 100 (b3.0:b2.0:b1.0, []);
9       X = Y + Z (b3.0:b2.0:b1.0, []);
10    end;
11  end;
12 end;

```

(a) Sample program

Block name	Local names
b1	{X}
b2	{Y}
b3	{X}

(b) Scope information

Figure 4.3: A sample of a program and the scope information required for variable evaluation, where Z is a global variable.

Definition 4.2.1. (Statement path) A statement path is a representation of a statement's position within a program. A path is an ordered sequence of block names (unique construct identifiers) in which the statement resides.

Given a statement path, all of the possible block names that could have been used to index this version within γ are accessible. The correct name can be determined using *scope information*. The scope information contains all local names used within a given block. Starting with the name of the block in which a statement appears, the scope information is used to find the first block name within the statement path that uses the given variable name locally. If a match is not found and the end of the path is reached, this variable is global and so will be indexed using λ within the variable environment. In examples throughout this thesis, paths are often omitted as they are easily read from the code. Consider Example 13 below.

Example 13. (Variable evaluation) Consider the program in Figure 4.3(a) and the representation of the scope information for this execution in Figure 4.3(b). Line 1 contains an assignment of variable X with a path λ . This empty path indicates the variable is global, meaning this version of the variable is accessed from the variable environment using (X, λ) . Line 9 shows an assignment using three different variables using the path **b3.0:b2.0:b1.0**. This statement starts by evaluating the expression $Y + Z$. For each variable, the block names within the path are considered in order until the first is found that contains the variable name within its set of local names from the scope information (Figure 4.3(b)). For Y, **b3.0** does not contain a local version of Y meaning we move onto **b2.0**, which does contain Y. Therefore this is accessed from the variable environment using $(Y, \mathbf{b2.0})$. Each block name is checked for variable Z, where no match is found meaning this variable is global (accessed as

X above). Finally, the variable X is evaluated. Each block name is checked, with a match found for $b3.0$. Note that a match would have also been found for $b1.0$, but crucially was not the first. Therefore this X is accessed using $(X, b3.0)$. ■

The final component of the variable state is the data store σ . This environment linking each memory location to the value it holds is unchanged from Section 2.2 in Chapter 2, with identical notation used throughout.

While and Procedure Environments

Section 2.2 of Chapter 2 introduced the while environment as a solution to issues introduced by code reuse. In order to maintain the uniqueness of construct identifiers, all of those used within a loop body are renamed prior to each iteration. In order to maintain the copies of all while loops, the while environment β is used to assign a unique loop name (construct identifier) to a copy of that loop. The while environment and all necessary notation is now used unchanged from Chapter 3.

Support for procedures introduces another cause of code reuse. Consider the parallel composition of two calls to the same procedure. Let this procedure contain a block statement that declares a local variable. The correct behaviour of such a situation is that each call statement declares a separate local variable, where the execution of one of the calls does not affect the variable from the other. Currently this requirement is violated, as the block name in which the variable is declared will be the same on each side. This means both declare a variable indexed with the same block name (onto γ), and therefore both use the same version of the variable name.

We therefore extend the process of renaming to now include a procedure body prior to its execution. Each call statement uses a renamed version of the procedure body, with all copies maintained with the newly introduced *procedure environment* μ . From this point on, we refer to the entry made into the procedure environment via a procedure declaration statement as the *basis mapping*, from which all copies are produced. Each copy can be unambiguously indexed with this procedure environment using the construct identifier of the call statement (which are guaranteed to be unique). With \mathbf{N} being the set of procedure names \mathbf{n} used within the code, \mathbf{Pn} the set of construct identifiers associated with procedures and \mathbf{Pa} being the set of paths \mathbf{pa} , we introduce the function $evalP: (\mathbf{N} \times \mathbf{Pa}) \rightarrow \mathbf{Pn}$, which given a procedure name and a path, returns the unique construct identifier \mathbf{Pn} of the procedure to use (using the scope information like for variables that share names). The final consideration is how to make each copy unique, and specifically the block name unique in each separate procedure body copy. Recalling that each call statement is named uniquely, we therefore rename all nested construct identifiers to begin with the unique call name. For example, a block statement named $b1.0$ within a call

<pre> 1 proc p1.0 sellN is 2 begin b1.0 3 var num = 4; 4 if i1.0 (count >= num) then 5 seats = seats - num; 6 else 7 errorNum = errorNum - 1; 8 end; 9 while w1.0 (num > 0) then 10 call c1.0 issueTicket; 11 num = num - 1; 12 end; 13 end 14 end </pre>	<pre> 1 begin c2.0:b1.0 2 var num = 4; 3 if c2.0:i1.0 (count >= num) then 4 seats = seats - num; 5 else 6 errorNum = errorNum - 1; 7 end; 8 while c2.0:w1.0 (num > 0) then 9 call c2.0:c1.0 issueTicket; 10 num = num - 1; 11 end; 12 end </pre>
---	--

(a) Original sell procedure

(b) Renamed version for call c2.0

Figure 4.4: An original procedure and a renamed version

statement `c2.0` is renamed to `c2.0:b1.0` (including recursive calls). Therefore our two calls in parallel will now have uniquely named block statements and will now declare two separate local variables, as required. For consistency, all construct identifiers used within a procedure body are renamed accordingly, including nested call statements crucial to maintain the validity of this renaming. Consider Example 14 of a procedure and a renamed copy of it.

Example 14. (Renaming a procedure body) Consider the basis mapping of the procedure `sellN` from Figure 4.4(a), which sells and then issues four tickets (provided four are available). The call statement `call c2.0 sellN pa` produces the renamed copy shown in Figure 4.4(b), where all construct identifiers are unique (including between parallel calls to this procedure). ■

The function $reP: (\mathbb{P} \times \mathbf{Cn}) \rightarrow \mathbb{P}$ takes a program (copy of the procedure body) and its unique construct identifier, and returns the renamed version.

4.3 Traditional Operational Semantics

As in Chapter 2, we now use syntax-directed, small step operational semantics to define the traditional execution of this extended language (where information is lost and executions are irreversible). Each configuration now consists of a program P (over the extended syntax) and a program state \square that now includes the procedure environment μ . We represent the execution of programs on program states as a relation between configurations. This relation is called a transition relation and here is

defined as the least relation generated by the transition rules in this section. Recall that \hookrightarrow_a and \hookrightarrow_b represent a step of arithmetic or boolean evaluation respectively, and the reflective and transitive closure \hookrightarrow^* indicates an arbitrary number of execution steps. All rules are restated here, with all differences (including the use of the newly introduced procedure environment) discussed.

Arithmetic Expressions

Arithmetic expressions are evaluated as in Chapter 2, with the evaluation of a variable (rule [A1]) containing a path \mathbf{pa} , and using $evalV()$.

$$\begin{array}{ll}
[A1] \quad \frac{evalV(\gamma, \mathbf{pa}, \mathbf{X}) = 1}{(\mathbf{X} \ \mathbf{pa} \mid \sigma, \gamma, \square) \hookrightarrow_a (\sigma(1) \mid \sigma, \gamma, \square)} & [A2] \quad \frac{v = n \ \mathbf{op} \ m}{(n \ \mathbf{op} \ m \mid \square) \hookrightarrow_a (v \mid \square)} \\
[A3] \quad \frac{}{(\langle v \rangle \mid \square) \hookrightarrow_a (v \mid \square)} & [A4] \quad \frac{(a_0 \mid \square) \hookrightarrow_a (a'_0 \mid \square)}{(\langle a_0 \rangle \mid \square) \hookrightarrow_a (\langle a'_0 \rangle \mid \square)} \\
[A5] \quad \frac{(a_0 \mid \square) \hookrightarrow_a (a'_0 \mid \square)}{(a_0 \ \mathbf{op} \ a_1 \mid \square) \hookrightarrow_a (a'_0 \ \mathbf{op} \ a_1 \mid \square)} & [A6] \quad \frac{(a_1 \mid \square) \hookrightarrow_a (a'_1 \mid \square)}{(a_0 \ \mathbf{op} \ a_1 \mid \square) \hookrightarrow_a (a_0 \ \mathbf{op} \ a'_1 \mid \square)}
\end{array}$$

Boolean Expressions

Unchanged to that of Chapter 2.

$$\begin{array}{ll}
[B1] \quad \frac{}{(\neg T \mid \square) \hookrightarrow_b (F \mid \square)} & [B2] \quad \frac{}{(\neg F \mid \square) \hookrightarrow_b (T, \square)} \\
[B3] \quad \frac{(b \mid \square) \hookrightarrow_b (b' \mid \square)}{(\neg b \mid \square) \hookrightarrow_b (\neg b' \mid \square)} & [B4] \quad \frac{ba_2 = ba_0 \ \mathbf{bop} \ ba_1}{(ba_0 \ \mathbf{bop} \ ba_1 \mid \square) \hookrightarrow_b (ba_2 \mid \square)} \\
[B5] \quad \frac{(ba_0 \mid \square) \hookrightarrow_b (ba'_0 \mid \square)}{(ba_0 \ \mathbf{bop} \ ba_1 \mid \square) \hookrightarrow_b (ba'_0 \ \mathbf{bop} \ ba_1 \mid \square)} & [B6] \quad \frac{(ba_1 \mid \square) \hookrightarrow_b (ba'_1 \mid \square)}{(ba_0 \ \mathbf{bop} \ ba_1 \mid \square) \hookrightarrow_b (ba_0 \ \mathbf{bop} \ ba'_1 \mid \square)}
\end{array}$$

Sequential Composition

Unchanged to that of Chapter 2.

$$\begin{array}{ll}
[S1] \quad \frac{(S \mid \square) \hookrightarrow (S' \mid \square')}{(S; P \mid \square) \hookrightarrow (S'; P \mid \square')} & [S2] \quad \frac{}{(\mathbf{skip}; P \mid \square) \hookrightarrow (P \mid \square)}
\end{array}$$

Parallel Composition

Unchanged to that of Chapter 2.

$$\begin{array}{ll}
[P1] \quad \frac{(P \mid \square) \hookrightarrow (P' \mid \square')}{(P \ \mathbf{par} \ Q \mid \square) \hookrightarrow (P' \ \mathbf{par} \ Q \mid \square')} & [P2] \quad \frac{(Q \mid \square) \hookrightarrow (Q' \mid \square')}{(P \ \mathbf{par} \ Q \mid \square) \hookrightarrow (P \ \mathbf{par} \ Q' \mid \square')} \\
[P3] \quad \frac{}{(\mathbf{skip} \ \mathbf{par} \ \mathbf{skip} \mid \square) \hookrightarrow (\mathbf{skip} \mid \square)} &
\end{array}$$

Assignment

Assignment statements behave much like those in Chapter 2, but with a path pa and use of the function $\text{evalV}()$ that supports local variables.

$$[D1] \quad \frac{(\text{e pa} \mid \sigma, \gamma, \square) \hookrightarrow_a^* (\text{v} \mid \sigma, \gamma, \square) \quad \text{evalV}(\text{X}, \text{pa}, \gamma) = 1}{(\text{X} = \text{e pa} \mid \sigma, \gamma, \square) \hookrightarrow (\text{skip} \mid \sigma[1 \mapsto \text{v}], \gamma, \square)}$$

Conditional Statements

With the addition of a path pa , the semantics is unchanged from Chapter 2.

$$[I1T] \quad \frac{(\text{b pa} \mid \square) \hookrightarrow_b^* (\text{T} \mid \square)}{(\text{if In b then P else Q end pa} \mid \square) \hookrightarrow (\text{if In T then P else Q end pa} \mid \square)}$$

$$[I1F] \quad \frac{(\text{b pa} \mid \square) \hookrightarrow_b^* (\text{F} \mid \square)}{(\text{if In b then P else Q end pa} \mid \square) \hookrightarrow (\text{if In F then P else Q end pa} \mid \square)}$$

$$[I2] \quad \frac{(\text{P} \mid \square) \hookrightarrow (\text{P}' \mid \square')}{(\text{if In T then P else Q end pa} \mid \square) \hookrightarrow (\text{if In T then P}' \text{ else Q end pa} \mid \square')}$$

$$[I3] \quad \frac{(\text{Q} \mid \square) \hookrightarrow (\text{Q}' \mid \square')}{(\text{if In F then P else Q end pa} \mid \square) \hookrightarrow (\text{if In F then P else Q}' \text{ end pa} \mid \square')}$$

$$[I4] \quad \frac{}{(\text{if In T then skip else Q end pa} \mid \square) \hookrightarrow (\text{skip} \mid \square)}$$

$$[I5] \quad \frac{}{(\text{if In F then P else skip end pa} \mid \square) \hookrightarrow (\text{skip} \mid \square)}$$

While Loops

With the addition of a path pa , the semantics is unchanged from Chapter 2.

$$[W1] \quad \frac{\beta(\text{Wn}) = \text{und} \quad (\text{b pa} \mid \beta, \square) \hookrightarrow_b^* (\text{F} \mid \beta, \square)}{(\text{while Wn b do P end pa} \mid \beta, \square) \hookrightarrow (\text{skip} \mid \beta, \square)}$$

$$[W2] \quad \frac{\beta(\text{Wn}) = \text{def} \quad (\text{b pa} \mid \beta, \square) \hookrightarrow_b^* (\text{F} \mid \beta, \square)}{(\text{while Wn b do P end pa} \mid \beta, \square) \hookrightarrow (\text{skip} \mid \beta[\text{Wn}], \square)}$$

$$[W3] \quad \frac{\beta(\text{Wn}) = \text{und} \quad (\text{b pa} \mid \beta, \square) \hookrightarrow_b^* (\text{T} \mid \beta, \square)}{(\text{S} \mid \beta, \square) \hookrightarrow (\text{while Wn T do reL(P) end pa} \mid \beta[\text{Wn} \Rightarrow \text{AR}], \square)}$$

where $\text{S} = \text{while Wn b do P end pa}$ and $\text{AR} = \text{while Wn b do reL(P) end pa}$

$$[W4] \quad \frac{\beta(\text{Wn}) = \text{def} \quad (\text{b pa} \mid \beta, \square) \hookrightarrow_b^* (\text{T} \mid \beta, \square)}{(\text{S} \mid \beta, \square) \hookrightarrow (\text{while Wn T do reL(P) end pa} \mid \beta[\text{Wn} \Rightarrow \text{AR}], \square)}$$

where $\text{S} = \text{while Wn b do P end pa}$ and $\text{AR} = \text{while Wn b do reL(P) end pa}$

$$[W5] \quad \frac{\beta(\text{Wn}) = \text{def} \quad (\text{P} \mid \square) \hookrightarrow (\text{P}' \mid \square')}{(\text{while Wn T do P end pa} \mid \square) \hookrightarrow (\text{while Wn T do P}' \text{ end pa} \mid \square')}$$

$$[W6] \quad \frac{\beta(\text{Wn}) = \text{while Wn b do P end pa}}{(\text{while Wn T do skip end pa} \mid \beta, \square) \hookrightarrow (\text{while Wn b do P end pa} \mid \beta, \square)}$$

Block

The block body executes via [B1]. The repeated use of this transition rule executes the entire block body (provided the statement eventually terminates), at which point the block can close via [B2].

$$[B1] \quad \frac{(P \mid \square) \hookrightarrow (P' \mid \square')}{(\text{begin } b1 \ P \ \text{end} \mid \square) \hookrightarrow (\text{begin } b1 \ P' \ \text{end} \mid \square')}$$

$$[B2] \quad \frac{}{(\text{begin } b1 \ \text{skip} \ \text{end} \mid \square) \hookrightarrow (\text{skip} \mid \square)}$$

Variable and Procedure Declaration

A local variable declaration ([L1]) gets a fresh memory location l ($nextLoc(\sigma) = l$) and takes the block name Bn from the path pa ($pa = Bn:pa'$). The entry is made into the variable environment between the pair containing the variable and block name, and the fresh memory location ($\gamma[(X, Bn) \Rightarrow l]$). The memory location is initialised to the value v produced by evaluating e ($\sigma[l \mapsto v]$).

$$[L1] \quad \frac{(e \ pa \mid \sigma, \gamma, \square) \hookrightarrow_a^* (v \mid \sigma, \gamma, \square) \quad nextLoc(\sigma) = l \quad pa = Bn:pa'}{(\text{var } X = e \ pa \mid \sigma, \gamma, \square) \hookrightarrow (\text{skip} \mid \sigma[l \mapsto v], \gamma[(X, Bn) \Rightarrow l], \square)}$$

A procedure is declared via [L2]. The basis mapping is inserted into the procedure environment μ , linking the unique procedure name Pn to a pair containing the procedure name n used in the code and the procedure body P ($\mu[Pn \Rightarrow (n, P)]$).

$$[L2] \quad \frac{}{(\text{proc } Pn \ n \ \text{is } P \ pa \mid \mu, \square) \hookrightarrow (\text{skip} \mid \mu[Pn \Rightarrow (n, P)], \square)}$$

Variable and Procedure Removals

Local variables are removed via [H1]. Using the block name Bn from the statements path pa ($pa = Bn:pa'$), this version of the variable is evaluated to the memory location l ($\gamma(X, Bn) = l$). This memory location is then restored to 0 since it is guaranteed to have been fresh ($\sigma[l \mapsto 0]$) and then the entry within the variable environment is removed ($\gamma[(X, Bn)]$).

A procedure is removed via [H2]. Provided the entry is defined within the procedure environment ($\mu(Pn) = def$), this mapping is removed from μ ($\mu[Pn]$).

$$[H1] \quad \frac{pa = Bn:pa' \quad \gamma(X, Bn) = l}{(\text{remove } X = e \ pa \mid \sigma, \gamma, \square) \hookrightarrow (\text{skip} \mid \sigma[l \mapsto 0], \gamma[(X, Bn)], \square)}$$

$$[H2] \quad \frac{\mu(Pn) = def}{(\text{remove } Pn \ n \ \text{is } P \ pa \mid \mu, \square) \hookrightarrow (\text{skip} \mid \mu[Pn], \square)}$$

Procedure Call

A procedure call opens via [G1]. The given procedure name n and the statement path pa is used to evaluate the construct identifier of the procedure declaration statement ($evalP(n, pa) = Pn$). The procedure name Pn is used to retrieve the basis mapping ($\mu(Pn) = (n, P)$), which is renamed ($reP(P, Cn) = P'$) producing a copy of the procedure that is inserted into the procedure environment ($\mu[Cn \Rightarrow (n, P')]$). A **runc** instance is produced to allow the renamed copy to be executed.

$$[G1] \quad \frac{evalP(n, pa) = Pn \quad \mu(Pn) = (n, P) \quad reP(P, Cn) = P'}{(call \ Cn \ n \ pa \mid \mu, \square) \hookrightarrow (runc \ Cn \ P' \ end \mid \mu[Cn \Rightarrow (n, P')], \square)}$$

The entire procedure body then executes via the rule [G2].

$$[G2] \quad \frac{(P \mid \square) \hookrightarrow (P' \mid \square')}{(runc \ Cn \ P \ end \mid \square) \hookrightarrow (runc \ Cn \ P' \ end \mid \square')}$$

Once the body of the **runc** construct has reached skip, and provided the mapping is defined within the procedure environment ($\mu(Cn) = def$), the call statement closes via [G3] by removing the copy made for this call ($\mu[Cn]$).

$$[G3] \quad \frac{\mu(Cn) = def}{(runc \ Cn \ skip \ end \mid \mu, \square) \hookrightarrow (skip \mid \mu[Cn], \square)}$$

4.4 Examples of Traditional Execution

This section contains examples of program execution and the corresponding inference trees. Examples of assignments, conditionals and loops are available in Chapter 3, whereas we focus here on the additional programming language constructs.

First consider a procedure call statement that is currently executing the body (via **runc**). This body currently contains a block statement that requires the removal of a local variable. Let P be the program

```

runc c1
  begin b1
    remove X = 20 pa; Q
  end
end
```

and let $T = \text{begin } b1 \text{ remove } X = 20 \text{ pa; } Q \text{ end}$. Note that Q is the sequential composition of any number of removal statements. Let the local version of X initially

be 3. The inference tree showing the transition exists to perform this local variable declaration is now displayed.

$$\begin{array}{c}
\frac{\text{pa} = \text{b1:pa}' \quad \gamma(\text{X}, \text{b1}) = 1}{(\text{remove } \text{X} = 20 \text{ pa} \mid \sigma, \gamma, \mu, \square) \hookrightarrow (\text{skip} \mid \sigma[1 \mapsto 0], \gamma[(\text{X}, \text{b1})], \mu, \square)} \text{ [H1]} \\
\frac{(\text{remove } \text{X} = 20 \text{ pa}; \text{Q} \mid \sigma, \gamma, \mu, \square) \hookrightarrow (\text{skip}; \text{Q} \mid \sigma[1 \mapsto 0], \gamma[(\text{X}, \text{b1})], \mu, \square)}{(\text{T} \mid \sigma, \gamma, \mu, \square) \hookrightarrow (\text{begin b1 skip}; \text{Q} \text{ end} \mid \sigma[1 \mapsto 0], \gamma[(\text{X}, \text{b1})], \mu, \square)} \text{ [S1]} \\
\frac{(\text{T} \mid \sigma, \gamma, \mu, \square) \hookrightarrow (\text{begin b1 skip}; \text{Q} \text{ end} \mid \sigma[1 \mapsto 0], \gamma[(\text{X}, \text{b1})], \mu, \square)}{(\text{P} \mid \sigma, \gamma, \mu, \square) \hookrightarrow (\text{runc c1 begin b1 skip}; \text{Q} \text{ end end} \mid \sigma[1 \mapsto 0], \gamma[(\text{X}, \text{b1})], \mu, \square)} \text{ [B1]} \\
\frac{(\text{P} \mid \sigma, \gamma, \mu, \square) \hookrightarrow (\text{runc c1 begin b1 skip}; \text{Q} \text{ end end} \mid \sigma[1 \mapsto 0], \gamma[(\text{X}, \text{b1})], \mu, \square)}{(\text{P} \mid \sigma, \gamma, \mu, \square) \hookrightarrow (\text{runc c1 begin b1 skip}; \text{Q} \text{ end end} \mid \sigma[1 \mapsto 0], \gamma[(\text{X}, \text{b1})], \mu, \square)} \text{ [G2]}
\end{array}$$

[H1] is a leaf here as no further transitions appear within its premises. Instead they retrieve the name of the block in which this statement resides from the path, and evaluate the variable to its memory location 1. This location is reset to 0 ($\sigma' = \sigma[1 \mapsto 0]$) and the local variable removed ($\gamma' = \gamma[(\text{X}, \text{b1})]$). Therefore the old value X, namely 3, is overwritten and lost.

Our second example is of a program containing a block nested within a block, with the next step of its execution to close a call statement. Let P be the program

```

begin b1
  begin b2
    runc c1 skip end; Q
  end; R
end

```

The following inference tree shows that the desired transition exists, namely to close the call statement with name c1.

$$\begin{array}{c}
\frac{\mu(\text{Cn}) = \text{def}}{(\text{runc c1 skip end} \mid \mu, \square) \hookrightarrow (\text{skip} \mid \mu[\text{c1}], \square)} \text{ [G3]} \\
\frac{(\text{runc c1 skip end} \mid \mu, \square) \hookrightarrow (\text{skip} \mid \mu[\text{c1}], \square)}{(\text{runc c1 skip end}; \text{Q} \mid \mu, \square) \hookrightarrow (\text{skip}; \text{Q} \mid \mu[\text{c1}], \square)} \text{ [S1]} \\
\frac{(\text{runc c1 skip end}; \text{Q} \mid \mu, \square) \hookrightarrow (\text{skip}; \text{Q} \mid \mu[\text{c1}], \square)}{(\text{begin b2 runc c1 skip end}; \text{Q} \text{ end} \mid \mu, \square) \hookrightarrow (\text{begin b2 skip}; \text{Q} \text{ end} \mid \mu[\text{c1}], \square)} \text{ [B1]} \\
\frac{(\text{begin b2 runc c1 skip end}; \text{Q} \text{ end} \mid \mu, \square) \hookrightarrow (\text{begin b2 skip}; \text{Q} \text{ end} \mid \mu[\text{c1}], \square)}{(\text{begin b2 runc c1 skip end}; \text{Q} \text{ end}; \text{R} \mid \mu, \square) \hookrightarrow (\text{begin b2 skip}; \text{Q} \text{ end}; \text{R} \mid \mu[\text{c1}], \square)} \text{ [S1]} \\
\frac{(\text{begin b2 runc c1 skip end}; \text{Q} \text{ end}; \text{R} \mid \mu, \square) \hookrightarrow (\text{begin b2 skip}; \text{Q} \text{ end}; \text{R} \mid \mu[\text{c1}], \square)}{(\text{P} \mid \mu, \square) \hookrightarrow (\text{begin b1 begin b2 skip}; \text{Q} \text{ end}; \text{R} \text{ end} \mid \mu[\text{c1}], \square)} \text{ [B1]}
\end{array}$$

The repeated use of [B1] and [S1] reduces the program to the procedure call statement (runc construct). [G3] here is a leaf, as its only premise requires only for a mapping to exist within μ . As a result, the procedure is closed by removing the entry for this call statement from the procedure environment (hence $\mu' = \mu[\text{c1}]$).

4.5 Annotation

This section begins the process of adding reversibility to the extended version of our programming language. As in Chapter 3, our approach is to save all information lost during the traditional execution, similarly to the Reverse C Compiler (RCC) by Perumalla et al [63, 11], Backstroke [93] and work by Schordan et al [75, 76, 77]. Given an original program, we again produce a forward annotated version and an inverted version. In this section we produce the annotated version and define its execution. This includes all reversal information and identifier use required for the additional program constructs introduced in this language. Inversion and reverse execution is introduced in Section 4.8.

4.5.1 Extended Reversal Information

For all constructs that appear unchanged here from Chapter 3, the reversal information required for each is as described there. We now consider each type of additional statement and describe the type of information required for the correct reversal of each. This subsection concludes with a description of the updates required to the auxiliary store δ in order to save this extra reversal information.

Block Statement

A block statement does not require any reversal information directly, as a step of a block body does not lose information (other than the information lost via the statement within the body) as it executes, and neither does a block closure.

Variable Declaration and Removal

The declaration of a local variable does not lose any information that cannot be recovered, and instead creates a mapping within the variable environment (to a fresh memory location retrieved via $nextLoc(\sigma)$) and initialises this local variable to the given value (evaluated expression). Since all memory locations are initialised to 0 (Section 4.2.1) and the specific location used must be fresh, the value 0 is always lost. The inversion of a declaration statement can therefore also use 0 and so is not required to be saved.

A local variable removal statement does however lose information that cannot be recalculated during inversion. Removing a variable requires the deletion of the corresponding mapping within the variable environment (which can be recalculated during inversion as this maps the variable and block name to a potentially different memory location), and the overwriting of the value held at that memory location

(back to 0). Crucially this old value cannot be restored during the inverse execution and is therefore required to be saved.

Procedure Declaration and Removal

A procedure declaration statement does not overwrite any information and instead only creates an entry within the procedure environment. Since all local names within a specific block are unique, no mapping can currently exist and therefore cannot be overwritten. Therefore no reversal information is required.

The removal of a procedure deletes this mapping from the procedure environment. Each of the procedure name and procedure body can be recovered from the inverted version of the program, and therefore can be restored without the use of any reversal information. Since this is the basis mapping of a procedure, this will not be directly executed and therefore no identifiers are associated to statements here. This means no information needs to be saved.

Procedure Call

A call begins with the evaluation of the given name to access the basis mapping of this procedure. A copy of this procedure body is inserted into the procedure environment, indexed with the call statement construct identifier. Since call names are unique, there cannot be an existing mapping and so no information is lost.

The execution of a procedure body does not lose information directly, and instead only that of the individual statements. The final step of a procedure call is to remove the copy of the procedure body made for this call statement. This mapping can be recovered from the inverse program (the procedure name and inverted procedure body) and therefore does not require any reversal information. The exception is the loss of any identifiers assigned to statements within this copy (Section 4.5.2).

Extended Auxiliary Store

Recall the auxiliary store δ from Chapter 3, where this is introduced as the environment to store all reversal information separately to the program state. This is currently a collection of stacks with one for each variable name (storing old values), one for all conditional statements (named **B**), one for all while loops (named **W**) and one for identifiers associated to copies of while loops (named **WI**).

Two key extensions are required to the auxiliary store in order to store the reversal information needed for the additional programming language constructs. The first concerns each stack that is for a specific variable. Each such stack will now also be used to save the final value held by each local version of that variable name (again to handle races). The second extension is the introduction of the stack

Pr , used to store the sequence of identifiers associated with a copy of a procedure body prior to its removal. This can be viewed as the equivalent of the stack WI but for procedures. From here, a reference to δ represents our extended auxiliary store. Recall that \mathbb{X} is the set of all variable names, \mathbb{K} is the set of identifiers, \mathbb{B} is the set $\{\text{T}, \text{F}\}$ of boolean values, \mathbb{Z} is the set of integers and \mathbb{C} is the set of identifier sequences retrieved from a loop or procedure copy prior to its removal. Then δ is defined as $\delta : (\mathbb{X} \rightarrow (\mathbb{K} \times \mathbb{Z})) \cup (\mathbb{B} \rightarrow (\mathbb{K} \times \mathbb{B})) \cup (\mathbb{W} \rightarrow (\mathbb{K} \times \mathbb{B})) \cup (\text{WI} \rightarrow (\mathbb{K} \times \mathbb{C})) \cup (\text{Pr} \rightarrow (\mathbb{K} \times \mathbb{C}))$.

4.5.2 Extended Identifier Use

Our approach to reversibility uses the notion of identifiers to capture the interleaving order of a program as it executes. As explained in Chapter 3, identifiers are used in ascending order and associated to statements as they execute. This is extended here to support the additional program constructs, namely blocks, declarations and removals, and procedure call statements. Each is now considered in turn.

Block Statement

A block statement is used to encapsulate a sub-program, with no rule to explicitly open it (Section 4.3). Execution of the block body begins immediately, with identifiers used depending on the type of statements. The single step to close a block does not alter the program state in any way and simply allows the block statement to reach skip. Therefore the closure of a block becomes another example of a statement that is not required to follow backtracking order. We note that the inverse of a block closure is the opening of the inverted block statement (which does not have an explicit rule and would lead to difficulty in using any identifier assigned).

Variable/Procedure Declaration and Removal

Both the declaration and removal of either a local variable or a procedure have a direct impact on the program state. Each could be part of a race between the declaration or removal of two variables/procedures that share the same name. These two factors mean the interleaving order of such statements is crucial for correct reversal, meaning all such statements (that complete in a single step) use an identifier.

Procedure Call

The first step of a call is to evaluate the given name and to create a unique version of this procedure body. This direct impact on the program state is reversed via the closure of the inverted call statement. The changes this makes require an identifier to ensure backtracking order is followed. The execution of the procedure body may


```

AP ::= ε | AS | AP; AP | AP par AP
AS ::= skip I | X = E (pa,A) | if In B then AP else AQ end (pa,A) |
      while Wn B do AP end (pa,A) | begin Bn ABB end |
      call Cn n (pa,A) | runc Cn AP end A
ABB ::= ADV; ADP; AP; ARP; ARV
E ::= X | n | (E) | E Op E
B ::= T | F | ¬B | (B) | E == E | E > E | B ∧ B

ADV ::= ε | var X = E (pa,A); ADV
ADP ::= ε | proc Pn n is AP end (pa,A); ADP
ARV ::= ε | remove X = E (pa,A); ARV
ARP ::= ε | remove Pn n is AP end (pa,A); ARP

```

Figure 4.5: Syntax of our annotated concurrent language

use identifiers throughout, depending on the type of statements. The final step of a call is to close it, where the copy of the procedure body for this call is removed. This step therefore requires the use of an identifier. Further to this, all identifiers associated with statements within the block body will be lost. As is the case for while loops, the sequence of all of these lost identifiers (retrieved via *getAI()*) is saved alongside the identifier used to close the call onto the stack *Pr* on δ .

4.5.3 Annotation Function

We now update the definition of annotated programs to reflect the extended programming language. The annotated version must be capable of storing identifiers that are assigned to specific statements. As in Chapter 3, the syntax of annotated programs differs to that of original programs, and is given here in Figure 4.5, where *AP* and *AS* represent annotated programs and statements respectively.

By abuse of notation, we extend the previously defined functions used to produce the annotated version. This is the function *ann*: $\mathbb{P} \rightarrow \mathbb{AP}$ (recalling \mathbb{P} and \mathbb{AP} are the sets of original and annotated programs respectively), that uses the function *a*: $\mathbb{S} \rightarrow \mathbb{AS}$ (recalling \mathbb{S} and \mathbb{AS} are the sets of original and annotated statements respectively) to annotate each statement. Figure 4.6 contains the definition of both of these functions. Recall our discussion from Chapter 3 regarding the identifier stacks. Note that throughout this chapter, we write identifier stacks within the syntax. As previously explained, these stacks can actually be implemented via a separate environment, allowing all such stacks to be persistent. Similarly to Chapter 3, we omit the explicit use of this environment and give all stacks within statements.

$$\begin{aligned}
ann(\varepsilon) &= \varepsilon \\
ann(S;P) &= a(S); ann(P) \\
ann(P \text{ par } Q) &= ann(P) \text{ par } ann(Q) \\
a(\text{skip}) &= \text{skip } I \\
a(X = e \text{ pa}) &= X = e \text{ (pa,A)} \\
a(\text{if In b then P else Q end pa}) &= \\
&\quad \text{if In b then } ann(P) \text{ else } ann(Q) \text{ end (pa,A)} \\
a(\text{while Wn b do P end pa}) &= \text{while Wn b do } ann(P) \text{ end (pa,A)} \\
a(\text{begin Bn P end}) &= \text{begin Bn } ann(P) \text{ end} \\
a(\text{var X = E pa}) &= \text{var X = E (pa,A)} \\
a(\text{proc Pn n is P end pa}) &= \text{proc Pn n is } ann(P) \text{ end (pa,A)} \\
a(\text{call Cn n pa}) &= \text{call Cn n (pa,A)} \\
a(\text{runc Cn P end}) &= \text{runc Cn AP end A} \\
a(\text{remove X = E pa}) &= \text{remove X = E (pa,A)} \\
a(\text{remove Pn n is P end pa}) &= \text{remove Pn n is } ann(P) \text{ end (pa,A)}
\end{aligned}$$

Figure 4.6: Annotation function

4.6 Forward Semantics of Annotated Programs

This section contains small step operational semantics defining the forward execution of annotated programs. By abuse of notation, the transition relation \rightarrow is the smallest relation generated by the transition rules given here. The reflective and transitive closure \rightarrow^* now represents an annotated execution. We re-use the definition of identifier and skip steps from Chapter 3 to represent transition rules that do and do not use identifiers respectively. All programs used within configurations from this point on are written using the annotated syntax in Figure 4.5. Each transition rule defined within the forward-only semantics in Section 4.3 has a corresponding rule here, where the effect on all program state environments is identical, except for the auxiliary store δ which is now populated.

Sequential Composition

Unchanged to that of Chapter 3.

$$\begin{array}{ll}
\text{[S1a]} \quad \frac{(AS \mid \square) \xrightarrow{\circ} (AS' \mid \square')}{(AS; AP \mid \square) \xrightarrow{\circ} (AS'; AP \mid \square')} & \text{[S2a]} \quad \frac{}{(\text{skip } I; AP \mid \square) \rightarrow_s (AP \mid \square)}
\end{array}$$

Parallel Composition

Unchanged to that of Chapter 3.

$$\begin{array}{c}
[P1a] \frac{(AP \mid \square) \overset{\circ}{\rightarrow} (AP' \mid \square')}{(AP \text{ par } AQ \mid \square) \overset{\circ}{\rightarrow} (AP' \text{ par } AQ \mid \square')} \qquad [P2a] \frac{(AQ \mid \square) \overset{\circ}{\rightarrow} (AQ' \mid \square')}{(AP \text{ par } AQ \mid \square) \overset{\circ}{\rightarrow} (AP \text{ par } AQ' \mid \square')} \\
[P3a] \frac{}{(\text{skip } I_1 \text{ par skip } I_2 \mid \square) \rightarrow_s (\text{skip} \mid \square)}
\end{array}$$

Assignment

As in Chapter 3 but using a path \mathbf{pa} and the function $evalV()$.

$$[D1a] \frac{\mathbf{m} = next() \quad (\mathbf{e} \text{ pa} \mid \sigma, \gamma, \square) \hookrightarrow_a^* (\mathbf{v} \mid \sigma, \gamma, \square) \quad evalV(\mathbf{X}, \mathbf{pa}, \gamma) = 1}{(\mathbf{X} = \mathbf{e} \text{ (pa, A)} \mid \delta, \sigma, \gamma, \square) \xrightarrow{m} (\text{skip } \mathbf{m}:\mathbf{A} \mid \delta[(\mathbf{m}, \sigma(1)) \rightarrow \mathbf{X}], \sigma[1 \mapsto \mathbf{v}], \gamma, \square)}$$

Conditional Statements

As in Chapter 3 but using a path \mathbf{pa} .

$$[I1aT] \frac{\mathbf{m} = next() \quad (\mathbf{b} \text{ pa} \mid \square) \hookrightarrow_b^* (\mathbf{T} \mid \square)}{(\text{if In } \mathbf{b} \text{ then AP else AQ end (pa, A)} \mid \square) \xrightarrow{m} (\text{if In } \mathbf{T} \text{ then AP else AQ end (pa, m:A)} \mid \square)}$$

$$[I1aF] \frac{\mathbf{m} = next() \quad (\mathbf{b} \text{ pa} \mid \square) \hookrightarrow_b^* (\mathbf{F} \mid \square)}{(\text{if In } \mathbf{b} \text{ then AP else AQ end (pa, A)} \mid \square) \xrightarrow{m} (\text{if In } \mathbf{F} \text{ then AP else AQ end (pa, m:A)} \mid \square)}$$

$$[I2a] \frac{(\mathbf{AP} \mid \square) \overset{\circ}{\rightarrow} (\mathbf{AP}' \mid \square')}{(\text{if In } \mathbf{T} \text{ then AP else AQ end (pa, A)} \mid \square) \overset{\circ}{\rightarrow} (\text{if In } \mathbf{T} \text{ then AP}' \text{ else AQ end (pa, A)} \mid \square')}$$

$$[I3a] \frac{(\mathbf{AQ} \mid \square) \overset{\circ}{\rightarrow} (\mathbf{AQ}' \mid \square')}{(\text{if In } \mathbf{F} \text{ then AP else AQ end (pa, A)} \mid \square) \overset{\circ}{\rightarrow} (\text{if In } \mathbf{F} \text{ then AP else AQ}' \text{ end (pa, A)} \mid \square')}$$

$$[I4a] \frac{\mathbf{m} = next()}{(\text{if In } \mathbf{T} \text{ then skip I else AQ end (pa, A)} \mid \delta, \square) \xrightarrow{m} (\text{skip } \mathbf{m}:\mathbf{A} \mid \delta[(\mathbf{m}, \mathbf{T}) \rightarrow \mathbf{B}], \square)}$$

$$[I5a] \frac{\mathbf{m} = next()}{(\text{if In } \mathbf{F} \text{ then AP else skip I end (pa, A)} \mid \delta, \square) \xrightarrow{m} (\text{skip } \mathbf{m}:\mathbf{A} \mid \delta[(\mathbf{m}, \mathbf{F}) \rightarrow \mathbf{B}], \square)}$$

While Loops

As in Chapter 3 but using a path \mathbf{pa} . The interaction with the while environment is exactly as described previously.

$$[W1a] \frac{\mathbf{m} = next() \quad \beta(\mathbf{Wn}) = und \quad (\mathbf{b} \text{ pa} \mid \beta, \square) \hookrightarrow_b^* (\mathbf{F} \mid \beta, \square)}{(\text{while } \mathbf{Wn} \text{ b do AP end (pa, A)} \mid \delta, \beta, \square) \xrightarrow{m} (\text{skip } \mathbf{m}:\mathbf{A} \mid \delta[(\mathbf{m}, \mathbf{F}) \rightarrow \mathbf{W}], \beta, \square)}$$

$$\begin{aligned}
[W2a] \quad & \frac{\mathbf{m} = next() \quad \beta(\mathbf{Wn}) = def \quad (\mathbf{b} \text{ pa} \mid \beta, \square) \hookrightarrow_b^* (\mathbf{F} \mid \beta, \square)}{(\text{while } \mathbf{Wn} \text{ b do AP end } (\mathbf{pa}, \mathbf{A}) \mid \delta, \beta, \square) \xrightarrow{m} (\text{skip } \mathbf{m}:\mathbf{A} \mid \delta', \beta[\mathbf{Wn}], \square)} \\
& \text{where } \delta' = \delta[(\mathbf{m}, \mathbf{T}) \rightarrow \mathbf{W}, (\mathbf{m}, \mathbf{C}) \rightarrow \mathbf{WI}] \text{ and } \mathbf{C} = getAI(\beta(\mathbf{Wn})) \\
[W3a] \quad & \frac{\mathbf{m} = next() \quad \beta(\mathbf{Wn}) = und \quad (\mathbf{b} \text{ pa} \mid \beta, \square) \hookrightarrow_b^* (\mathbf{T} \mid \beta, \square)}{(\mathbf{S} \mid \delta, \beta, \square) \xrightarrow{m} (\text{while } \mathbf{Wn} \text{ T do AP' end } (\mathbf{pa}, \mathbf{m}:\mathbf{A}) \mid \delta[(\mathbf{m}, \mathbf{F}) \rightarrow \mathbf{W}], \beta[\mathbf{Wn} \Rightarrow \mathbf{AR}], \square)} \\
& \text{where } \mathbf{S} = \text{while } \mathbf{Wn} \text{ b do AP end } (\mathbf{pa}, \mathbf{A}), \text{ and } \mathbf{AP}' = reL(\mathbf{AP}) \\
& \text{and } \mathbf{AR} = \text{while } \mathbf{Wn} \text{ b do AP' end } (\mathbf{pa}, \mathbf{m}:\mathbf{A}) \\
[W4a] \quad & \frac{\mathbf{m} = next() \quad \beta(\mathbf{Wn}) = def \quad (\mathbf{b} \text{ pa} \mid \beta, \square) \hookrightarrow_b^* (\mathbf{T} \mid \beta, \square)}{(\mathbf{S} \mid \delta, \beta, \square) \xrightarrow{m} (\text{while } \mathbf{Wn} \text{ T do AP' end } (\mathbf{pa}, \mathbf{m}:\mathbf{A}) \mid \delta[(\mathbf{m}, \mathbf{T}) \rightarrow \mathbf{W}], \beta[\mathbf{Wn} \Rightarrow \mathbf{AR}], \square)} \\
& \text{where } \mathbf{S} = \text{while } \mathbf{Wn} \text{ b do AP end } (\mathbf{pa}, \mathbf{A}), \text{ and } \mathbf{AP}' = reL(\mathbf{AP}) \\
& \text{and } \mathbf{AR} = \text{while } \mathbf{Wn} \text{ b do AP' end } (\mathbf{pa}, \mathbf{m}:\mathbf{A}) \\
[W5a] \quad & \frac{\beta(\mathbf{Wn}) = def \quad (\mathbf{AP} \mid \delta, \beta, \square) \xrightarrow{\circ} (\mathbf{AP}' \mid \delta', \beta', \square')}{(\text{while } \mathbf{Wn} \text{ T do AP end } (\mathbf{pa}, \mathbf{A}) \mid \delta, \beta, \square) \xrightarrow{\circ} (\text{while } \mathbf{Wn} \text{ T do AP' end } (\mathbf{pa}, \mathbf{A}) \mid \delta', \beta'', \square')} \\
& \text{where } \beta'' = \beta'[refW(\mathbf{Wn}, \mathbf{AP}')] \\
[W6a] \quad & \frac{\beta(\mathbf{Wn}) = \text{while } \mathbf{Wn} \text{ b do AP end } (\mathbf{pa}, \mathbf{A})}{(\text{while } \mathbf{Wn} \text{ T do skip I end } (\mathbf{pa}, \mathbf{A}) \mid \delta, \beta, \square) \rightarrow_s (\text{while } \mathbf{Wn} \text{ b do AP end } (\mathbf{pa}, \mathbf{A}) \mid \delta, \beta, \square)}
\end{aligned}$$

Block

No reversal information or identifiers are used directly by a block statement, meaning the semantics is unchanged from the traditional execution within Section 4.3.

$$\begin{aligned}
[B1a] \quad & \frac{(\mathbf{AP} \mid \square) \xrightarrow{\circ} (\mathbf{AP}' \mid \square')}{(\text{begin b1 AP end } \mid \square) \xrightarrow{\circ} (\text{begin b1 AP' end } \mid \square')} \\
[B2a] \quad & \frac{}{(\text{begin b1 skip I end } \mid \square) \rightarrow_s (\text{skip } \mid \square)}
\end{aligned}$$

Variable and Procedure Declaration

A variable declaration matches that in Section 4.3, except that [L1a] uses the next available identifier ($\mathbf{m} = next()$) and pushes this into the identifier stack for this statement $\mathbf{m}:\mathbf{A}$. With the same reasoning, procedure declarations via [L2a] are unchanged except for an identifier use. Note that neither require reversal information.

$$\begin{aligned}
[L1a] \quad & \frac{\mathbf{m} = next() \quad (\mathbf{e} \text{ pa} \mid \sigma, \gamma, \square) \hookrightarrow_a^* (\mathbf{v} \mid \sigma, \gamma, \square) \quad nextLoc(\sigma) = 1 \quad \mathbf{pa} = \mathbf{Bn}:\mathbf{pa}'}{(\text{var } \mathbf{X} = \mathbf{e} \text{ (pa, A)} \mid \sigma, \gamma, \square) \xrightarrow{m} (\text{skip } \mathbf{m}:\mathbf{A} \mid \sigma[1 \mapsto \mathbf{v}], \gamma[(\mathbf{X}, \mathbf{Bn}) \Rightarrow 1], \square)} \\
[L2a] \quad & \frac{\mathbf{m} = next()}{(\text{proc Pn n is AP (pa, A)} \mid \mu, \square) \xrightarrow{m} (\text{skip } \mathbf{m}:\mathbf{A} \mid \mu[\mathbf{Pn} \Rightarrow (\mathbf{n}, \mathbf{AP})], \square)}
\end{aligned}$$

Variable and Procedure Removals

A variable removal [H1a] matches that of Section 4.3, but that the next identifier ($\mathbf{m} = next()$) is pushed to the identifier stack ($\mathbf{m}:\mathbf{A}$) and stored alongside the final value held by the variable into stack \mathbf{X} on $\delta(\delta(\mathbf{m}, \sigma(1)) \rightarrow \mathbf{X})$.

A procedure removal via [H2a] also uses the next available identifier ($\mathbf{m} = next()$) and inserts this into the statements identifier stack ($\mathbf{m}:\mathbf{A}$). All other effects are equal to that in Section 4.3, with the basis mapping of this procedure removed from the procedure environment ($\mu[\mathbf{Pn}]$).

[H1a]

$$\frac{\mathbf{m} = next() \quad \mathbf{pa} = \mathbf{Bn}:\mathbf{pa}' \quad \gamma(\mathbf{X}, \mathbf{Bn}) = 1}{(\text{remove } \mathbf{X} = \mathbf{e} \ (\mathbf{pa}, \mathbf{A}) \mid \delta, \sigma, \gamma, \square) \xrightarrow{m} (\text{skip } \mathbf{m}:\mathbf{A} \mid \delta(\mathbf{m}, \sigma(1)) \rightarrow \mathbf{X}, \sigma[1 \mapsto 0], \gamma[(\mathbf{X}, \mathbf{Bn})], \square)}$$

[H2a]

$$\frac{\mathbf{m} = next() \quad \mu(\mathbf{Pn}) = def}{(\text{remove } \mathbf{Pn} \text{ n is } \mathbf{AP} \ (\mathbf{pa}, \mathbf{A}) \mid \mu, \square) \xrightarrow{m} (\text{skip } \mathbf{m}:\mathbf{A} \mid \mu[\mathbf{Pn}], \square)}$$

Procedure Call

A procedure call opens via [G1a]. This matches [G1] from Section 4.3, but that the next available identifier ($\mathbf{m} = next()$) is inserted into the identifier stack ($\mathbf{m}:\mathbf{A}$). The renamed copy of the procedure body \mathbf{AP}' is inserted into the procedure environment (that will not already exist due to the uniqueness of the construct identifier \mathbf{Cn}).

$$[\text{G1a}] \quad \frac{\mathbf{m} = next() \quad evalP(\mathbf{n}, \mathbf{pa}) = \mathbf{Pn} \quad \mu(\mathbf{Pn}) = (\mathbf{n}, \mathbf{AP}) \quad reP(\mathbf{AP}, \mathbf{Cn}) = \mathbf{AP}'}{(\text{call } \mathbf{Cn} \ \mathbf{n} \ (\mathbf{pa}, \mathbf{A}) \mid \mu, \square) \xrightarrow{m} (\text{runc } \mathbf{Cn} \ \mathbf{AP}' \ \text{end } \mathbf{m}:\mathbf{A} \mid \mu[\mathbf{Cn} \Rightarrow (\mathbf{n}, \mathbf{AP}')], \square)}$$

The execution of the procedure body via [G2a] is unchanged from [G2] in Section 4.3. The one exception is that any identifiers associated to the statements of the copy of the procedure are reflected into the procedure environment mapping, meaning β is updated ($\mu'[refC(\mathbf{Cn}, \mathbf{AP}')]$).

$$[\text{G2a}] \quad \frac{(\mathbf{AP} \mid \mu, \square) \xrightarrow{\circ} (\mathbf{AP}' \mid \mu', \square')}{(\text{runc } \mathbf{Cn} \ \mathbf{AP} \ \text{end } \mathbf{A} \mid \mu, \square) \xrightarrow{\circ} (\text{runc } \mathbf{Cn} \ \mathbf{AP}' \ \text{end } \mathbf{A} \mid \mu'[refC(\mathbf{Cn}, \mathbf{AP}')], \square')}$$

The closure of a procedure call [G3a] uses the next identifier ($\mathbf{m} = next()$) and pushes it to the identifier stack ($\mathbf{m}:\mathbf{A}$). The final difference from Section 4.3 is that all of the identifiers associated to statements within the copy of the procedure body specifically for this call statement must be saved prior to its removal ($\delta(\mathbf{m}, getAI(\mu(\mathbf{Cn}), \mathbf{F})) \rightarrow \mathbf{Pr}$).

$$[G3a] \frac{m = next() \quad \mu(Cn) = def}{(runc \ Cn \ skip \ I \ end \ A \mid \delta, \mu, \square) \xrightarrow{m} (skip \ m:A \mid \delta[m, getAI(\mu(Cn))]) \multimap Pr, \mu[Cn], \square)}$$

4.7 Examples of Annotated Execution

We now return to the examples shown in Section 4.4, and show the inference trees of the corresponding annotated execution. We begin with the example of a procedure call statement executing a local variable declaration within a nested block. Let the annotated version AP be the program

```

runc c1
  begin b1
    remove X = 20 (pa,A); AQ
  end
end A1

```

and AT = `begin b1 remove X = 20 (pa,A); AQ end`. The inference tree of the annotated execution is shown below. Recall that the local version of X is initially 3. [H1a] is an axiom here. The premises match those in Section 4.4, meaning $\sigma' = \sigma[1 \mapsto 0]$ and $\gamma' = \gamma[(X, b1)]$. The auxiliary store $\delta' = \delta[(m, 3) \multimap X]$ now contains the final value this variable held, namely 3. We highlight that the identifier m is assigned to the copy of the procedure by updating μ , namely $\mu' = \mu[refC(Cn, AP')]$, where $AP' = \text{begin b1 skip m:A; AQ}$.

$$\begin{array}{c}
\frac{m = next() \quad pa = b1:pa' \quad \gamma(X, b1) = 1}{[H1a]} \\
\frac{(\text{remove } X = 20 \ (pa,A) \mid \delta, \sigma, \gamma, \mu, \square) \xrightarrow{m} (\text{skip } m:A \mid \delta', \sigma', \gamma', \mu', \square)}{[S1a]} \\
\frac{(\text{remove } X = 20 \ (pa,A); \text{AQ} \mid \delta, \sigma, \gamma, \mu, \square) \xrightarrow{m} (\text{skip } m:A; \text{AQ} \mid \delta', \sigma', \gamma', \mu', \square)}{[B1a]} \\
\frac{(\text{AT} \mid \delta, \sigma, \gamma, \mu, \square) \xrightarrow{m} (\text{begin b1 skip m:A; AQ end} \mid \delta', \sigma', \gamma', \mu', \square)}{[G2a]} \\
(\text{AP} \mid \delta, \sigma, \gamma, \mu, \square) \xrightarrow{m} (\text{runc c1 begin b1 skip m:A; AQ end end } A_1 \mid \delta', \sigma', \gamma', \mu', \square)
\end{array}$$

Recall the closing call statement within two nested blocks. Let AP be the program

```

begin b1
  begin b2
    runc c1 skip I end A; AQ
  end; AR
end

```

and $AT = \text{begin } b2 \text{ runc } c1 \text{ skip } I \text{ end } A; AQ \text{ end}; AR$ and let the program $AT' = \text{begin } b2 \text{ runc } c1 \text{ skip } I \text{ end } A; AQ \text{ end}$. The inference tree follows.

$$\begin{array}{c}
\frac{m = next() \quad \mu(Cn) = def}{[G3a]} \\
\frac{(\text{runc } c1 \text{ skip } I \text{ end } A \mid \delta, \mu, \square) \xrightarrow{m} (\text{skip } m:A \mid \delta', \mu', \square)}{[S1a]} \\
\frac{(\text{runc } c1 \text{ skip } I \text{ end } A; AQ \mid \delta, \mu, \square) \xrightarrow{m} (\text{skip } m:A; AQ \mid \delta', \mu', \square)}{[B1a]} \\
\frac{(\text{AT}' \mid \delta, \mu, \square) \xrightarrow{m} (\text{begin } b2 \text{ skip } m:A; AQ \text{ end} \mid \delta', \mu', \square)}{[S1a]} \\
\frac{(\text{AT} \mid \delta, \mu, \square) \xrightarrow{m} (\text{begin } b2 \text{ skip } m:A; AQ \text{ end}; AR \mid \delta', \mu', \square)}{[B1a]} \\
\frac{(\text{AP} \mid \delta, \mu, \square) \xrightarrow{m} (\text{begin } b1 \text{ begin } b2 \text{ skip } m:A; AQ \text{ end}; AR \text{ end} \mid \delta', \mu', \square)}{[B1a]}
\end{array}$$

This follows identically to Section 4.4, with the corresponding annotated transition rule used. The final procedure environment is $\mu' = \mu[Cn]$, as this copy of the procedure body is removed. The difference here is [H1a], which is still a leaf but now also saves all identifiers used by the procedure to the stack Pr before this procedure is removed. Therefore $\delta' = \delta[(m, getAI(\mu(Cn))) \multimap Pr]$.

Example 15 considers the execution of the complete program shown in Figure 4.9(a) (on page 84).

Example 15. (Full annotated execution) Consider Figure 4.9(a) on page 84, noting the program is sequential (meaning identifiers only link reversal information as there is no interleaving) and contains only a block statement. All variables are initially 0, and the next identifier is 4. The transition rule [B1a] allows the block body to execute, beginning with two variable declarations via [L1a] using identifiers 4 and 5. Next, a procedure is declared via [L2a] using the identifier 6 (that inserts the basis mapping for this procedure). The assignment $X = X + Y$ is executed via [D1a] using identifier 7, saving the pair (7,2) (the overwritten value 2) onto stack X on δ .

The call statement executes next, opening via [G1a] using identifier 8. A renamed copy of the procedure `add` is produced (in this case this is identical) and inserted into μ . This body is then executed (via [G2r] and the `runc` construct), using identifier 9 to overwrite the current value 0 of `res` (saving the pair (9,0)). The procedure call statement then completes using identifier 10. The copy of the procedure is removed from μ , after all identifiers associated with its statements are stored onto stack Pr on δ (in this case, the sequence is simply 9). The assignment $Y = 5$ via [D1a] uses the identifier 11 and saves the pair (11,2). The execution concludes by removing the basis mapping of the procedure via [L2a] (which requires no reversal information to invert as the program lost can be restored from the source code), and finally the removal of the two local variables using identifiers 13 and 14. The pairs (13,5) and (14,4) are saved onto the stacks X and Y respectively. The block statement concludes via the skip step [B2a].

The final state produced is such that $\text{res} = 6$ (as X and Y are both local and have been removed). The final procedure environment is empty as all mappings are removed, however just before the closure of the call statement, it contained the single mapping $\text{c1.0} \mapsto \text{res} = X + Y$ [9]. The final auxiliary store produced by this execution is now given.

X	Y	res	B	W	WI	Pr
(14,4)	(13,5)					
(7,2)	(11,2)	(9,0)				(10,9)

All identifier stacks are populated as shown in Figure 4.9(a). We return to this inverted version in Section 4.10 where, after the introduction of the reverse semantics to follow, the reverse execution of this program is explained.

This execution has been performed using our simulator (Chapter 7), demonstrating the final auxiliary store and program state to be correct. ■

4.8 Inversion

After the definition of the forward execution of an annotated program with reversal information saving, the next consideration is of the reverse execution. Prior to defining the extended functions required to produce the inverse execution, we first describe the use of identifiers to determine the inverse interleaving order and the use of reversal information to invert each statement.

4.8.1 Inverse Interleaving Order

Backtracking order is followed for the majority of a reverse execution, with the only relaxations being skip steps, the closing of a loop iteration (not the loop itself) and the closing of a block statement. Each of these steps does not use identifiers during the forward or reverse execution. Recall the function *previous()* that returns the previously used identifier (the next to invert). Any statement that must be inverted in backtracking order must contain the identifier m such that $m = \text{previous}()$. Note that all identifier stacks are persistent, with Chapter 3 detailing a possible implementation.

The declaration or removal of either a variable or a procedure each used a single identifier during the forward execution (for each time it was executed), meaning the inverted version of each also requires the use of the corresponding identifier. A procedure call statement uses two identifiers directly (excluding those used within the block body). The first is to open the call statement and the second to close it.

During the inverse execution the second identifier is now used to open the inverted call statement, while the first is used to close it.

4.8.2 Statement Reversal

The execution of an inverted version of a statement must undo all effects of the corresponding forward statement. To do so, a number of statements need to use the reversal information saved during the matching forward execution. Assignments, conditionals and loops are reversed as described in Section 3.6 of Chapter 3.

Block statements do not require reversal information and therefore no interaction with δ is required. An inverted block statement simply allows the body to execute before closing. A variable declaration statement (reversing a forward removal statement) recreates a local variable initialised to the final value it held during the forward execution, which is retrieved from the stack for this variable on δ . A variable removal statement (reversing a forward declaration statement) does not use δ (and instead restores the memory location to 0).

A procedure declaration statement during an inverse execution reverses a procedure removal statement. Since this is the basis mapping, no identifiers are assigned to it and therefore are not lost. This means a procedure declaration statement recreates the mapping based on the inverted program contained within it, meaning no reversal is used. A inverse procedure removal statement (reversing a procedure declaration) simply removes a mapping from the procedure environment, requiring no reversal information to do so.

The opening of a inverse procedure call (reversing the closure of the forward call) uses the sequence of identifiers saved prior to its removal from stack Pr . This is used to recreate the copy of the procedure body into μ , using the inverse renaming function $\text{IreP}: (\mathbb{IP} \times \mathbf{Cn}) \rightarrow \mathbb{IP}$ (recalling that \mathbb{IP} is the set of inverted programs and \mathbf{Cn} is the set of construct identifiers) and the function $\text{setAI}()$ to re-populate all identifier stacks. All steps of the inverted procedure body are then executed using reversal information dependent on the type of statement. Finally, the inverse procedure call closes (reversal of the opening during the forward execution) simply by removing the copy from μ , without the use of any reversal information.

4.8.3 Inversion Function

An inverted version of an original program must be capable of using the reversal information saved by the annotated execution to reverse it. This concerns only the identifiers as all information saved to δ is accessed within the operational semantics to follow. We now introduce the syntax of inverted programs in Figure 4.7, where IP and IS are used to represent inverted programs and statements respectively.

```

IP ::= ε | IS | IP; IP | IP par IP
IS ::= skip I | X = E (pa,A) | if In B then IP else IQ end (pa,A) |
      while Wn B do IP end (pa,A) | begin Bn IBB end |
      call Cn n (pa,A) | runc Cn IP end A
IBB ::= IDV; IDP; IP; IRP; IRV
E ::= X | n | (E) | E Op E
B ::= T | F | ¬B | (B) | E == E | E > E | B ∧ B

IDV ::= ε | var X = E (pa,A); IDV
IDP ::= ε | proc Pn n is IP end (pa,A); IDP
IRV ::= ε | remove X = E (pa,A); IRV
IRP ::= ε | remove Pn n is IP end (pa,A); IRP

```

Figure 4.7: Syntax of inverted versions of original programs

By abuse of notation, we now provide updated definitions of the functions used to generate an inverted version from Chapter 3. These are $inv: \mathbb{AP} \rightarrow \mathbb{IP}$ (recalling that \mathbb{AP} and \mathbb{IP} are the sets of annotated programs **AP** and inverted programs **IP** respectively), and $i: \mathbb{AS} \rightarrow \mathbb{IS}$ (recalling that \mathbb{AS} and \mathbb{IS} are the sets of annotated statements **AS** and inverted statements **IS** respectively). Both of these functions are given in Figure 4.8. As before, $i()$ is called on each statement in reverse order, allowing $inv()$ to invert the statement order.

One major difference here is the inversion of a block statement (not contained within Chapter 3). Inverting the statement order of a block body produces a block that begins with removal statements (since these are executed last). To help readability and the semantics, the function $inv()$ (and $i()$) change each removal statement into a declaration (and vice versa). Example 16 discusses a program containing a block statement and its inverted version.

Example 16. (Inverted program) A program containing an executed block statement (with an arbitrary block body) is shown in Figure 4.9(a), with the inverted version of it shown in Figure 4.9(b). The statement order has been inverted, and all declaration statements are now removal statements (and vice versa). ■

4.9 Reverse Semantics of Inverted Programs

The small step operational semantics performing reverse execution of programs is now shown. By abuse of notation, the transition relation \rightsquigarrow is the least relation generated by the rules given here. The reflective and transitive closure \rightsquigarrow^* also represents an inverse execution of this extended language. All configurations from this point on contain an inverted program, written in the syntax in Figure 4.7. As

$$\begin{aligned}
& \text{inv}(\varepsilon) = \varepsilon \\
& \text{inv}(\text{AS}; \text{AP}) = \text{inv}(\text{AP}); i(\text{AS}) \\
& \text{inv}(\text{AP} \text{ par } \text{AQ}) = \text{inv}(\text{AP}) \text{ par } \text{inv}(\text{AQ}) \\
& i(\text{skip } I) = \text{skip } I \\
& i(\text{X} = e \text{ (pa, A)}) = \text{X} = e \text{ (pa, A)} \\
& i(\text{if In b then AP else AQ end (pa, A)}) = \\
& \quad \text{if In b then } \text{inv}(\text{AP}) \text{ else } \text{inv}(\text{AQ}) \text{ end (pa, A)} \\
& i(\text{while Wn b do AP end (pa, A)}) = \text{while Wn b do } \text{inv}(\text{AP}) \text{ end (pa, A)} \\
& i(\text{begin Bn AP end}) = \text{begin Bn } \text{inv}(\text{AP}) \text{ end} \\
& i(\text{var X} = e \text{ (pa, A)}) = \text{remove X} = e \text{ (pa, A)} \\
& i(\text{proc Pn n is AP end (pa, A)}) = \text{remove Pn n is } \text{inv}(\text{AP}) \text{ end (pa, A)} \\
& i(\text{call Cn n (pa, A)}) = \text{call Cn n (pa, A)} \\
& i(\text{runc Cn AP end A}) = \text{runc Cn } \text{inv}(\text{AP}) \text{ A} \\
& i(\text{remove Pn n is AP end (pa, A)}) = \text{proc Pn n is } \text{inv}(\text{AP}) \text{ end (pa, A)} \\
& i(\text{remove X} = e \text{ (pa, A)}) = \text{var X} = e \text{ (pa, A)}
\end{aligned}$$

Figure 4.8: Inversion function

in Section 4.6, transition rules using an identifier are named identifier steps, while those that do not are named skip steps. Each identifier step defined below contains the premises $\mathbf{m} = \text{previous}()$ and $\mathbf{A} = \mathbf{m}:\mathbf{A}'$ to ensure this is the next statement to invert. Each identifier step removes the identifier \mathbf{m} from \mathbf{A} (leaving \mathbf{A}').

An important observation is that an inverted execution does not perform any evaluation, meaning rules for arithmetic or boolean evaluation are omitted. This lack of evaluation during an inverse execution potentially saves time when compared to a reversible language (e.g. Janus [55]), which has to evaluate a post-condition.

Sequential Composition

Unchanged to that of Chapter 3.

$$\begin{aligned}
[\text{S1r}] \quad & \frac{(\text{IS} \mid \square) \overset{\circ}{\rightsquigarrow} (\text{IS}' \mid \square')}{(\text{IS}; \text{IP} \mid \square) \overset{\circ}{\rightsquigarrow} (\text{IS}'; \text{IP} \mid \square')} & [\text{S2r}] \quad \frac{}{(\text{skip } I; \text{IP} \mid \square) \rightsquigarrow_s (\text{IP} \mid \square)}
\end{aligned}$$

Parallel Composition

Unchanged to that of Chapter 3.

$$\begin{aligned}
[\text{P1r}] \quad & \frac{(\text{IP} \mid \square) \overset{\circ}{\rightsquigarrow} (\text{IP}' \mid \square')}{(\text{IP} \text{ par } \text{IQ} \mid \square) \overset{\circ}{\rightsquigarrow} (\text{IP}' \text{ par } \text{IQ} \mid \square')} & [\text{P2r}] \quad \frac{(\text{IQ} \mid \square) \overset{\circ}{\rightsquigarrow} (\text{IQ}' \mid \square')}{(\text{IP} \text{ par } \text{IQ} \mid \square) \overset{\circ}{\rightsquigarrow} (\text{IP} \text{ par } \text{IQ}' \mid \square')} \\
[\text{P3r}] \quad & \frac{}{(\text{skip } I_1 \text{ par skip } I_2 \mid \square) \rightsquigarrow_s (\text{skip } \mid \square)}
\end{aligned}$$

<pre> 1 begin b1.0 2 var X = 2 [4]; 3 var Y = 2 [5]; 4 proc p1.0 add is 5 res = X + Y [9]; 6 end [6]; 7 X = X + Y [7]; 8 call c1.0 add [8,10]; 9 Y = 5 [11]; 10 remove p1.0 add is 11 res = X + Y [9]; 12 end [12]; 13 remove Y = 2 [13]; 14 remove X = 2 [14]; 15 end; </pre>	<pre> 1 begin b1.0 2 var X = 2 [14]; 3 var Y = 2 [13]; 4 proc p1.0 add is 5 res = X + Y [9]; 6 end [12]; 7 Y = 5 [11]; 8 call c1.0 add [8,10]; 9 X = X + Y [7]; 10 remove p1.0 add is 11 res = X + Y [9]; 12 end [6]; 13 remove Y = 2 [5]; 14 remove X = 2 [4]; 15 end; </pre>
---	---

(a) Executed program

(b) Inverted version

Figure 4.9: An executed block statement and the corresponding inverted version

Assignment

As in Chapter 3 but using a path pa .

$$[D1r] \quad \frac{A = m:A' \quad m = previous() \quad \delta(X) = (m,v):X' \quad evalV(X,pa,\gamma) = 1}{(X = e \ (pa,A) \mid \delta, \sigma, \square) \xrightarrow{m} (skip \ A' \mid \delta[X/X'], \sigma[1 \mapsto v], \square)}$$

Conditional Statements

As in Chapter 3 but using a path pa .

$$[I1rT] \quad \frac{A = m:A' \quad m = previous() \quad \delta(B) = (m,T):B'}{(S \mid \delta, \square) \xrightarrow{m} (if \ In \ T \ then \ IP \ else \ IQ \ end \ (pa,A') \mid \delta[B/B'], \square)}$$

where $S = if \ In \ b \ then \ IP \ else \ IQ \ end \ (pa,A)$

$$[I1rF] \quad \frac{A = m:A' \quad m = previous() \quad \delta(B) = (m,F):B'}{(S \mid \delta, \square) \xrightarrow{m} (if \ In \ F \ then \ IP \ else \ IQ \ end \ (pa,A') \mid \delta[B/B'], \square)}$$

where $S = if \ In \ b \ then \ IP \ else \ IQ \ end \ (pa,A)$

$$[I2r] \quad \frac{(IP \mid \square) \xrightarrow{\circ} (IP' \mid \square')}{(if \ In \ T \ then \ IP \ else \ IQ \ end \ (pa,A) \mid \square) \xrightarrow{\circ} (if \ In \ T \ then \ IP' \ else \ IQ \ end \ (pa,A) \mid \square')}$$

[I3r]

$$\frac{(IQ \mid \square) \overset{\circ}{\rightsquigarrow} (IQ' \mid \square')}{(\text{if In F then IP else IQ end } (pa, A), \square) \overset{\circ}{\rightsquigarrow} (\text{if In F then IP else IQ' end } (pa, A) \mid \square')}$$

[I4r]

$$\frac{A = m:A' \quad m = \text{previous}()}{(\text{if In T then skip I else IQ end } (pa, A) \mid \square) \overset{m}{\rightsquigarrow} (\text{skip } A' \mid \square)}$$

[I5r]

$$\frac{A = m:A' \quad m = \text{previous}()}{(\text{if In F then IP else skip I end } (pa, A) \mid \square) \overset{m}{\rightsquigarrow} (\text{skip } A' \mid \square)}$$

While Loop

As in Chapter 3 but using a path *pa*.

$$[W1r] \quad \frac{m = \text{previous()} \quad A = m:A' \quad \beta(Wn) = \text{und} \quad \delta(W) = (m, F):W'}{(\text{while Wn b do IP end } (pa, A) \mid \delta, \beta, \square) \overset{m}{\rightsquigarrow} (\text{skip } A' \mid \delta[W/W'], \beta, \square)}$$

$$[W2r] \quad \frac{m = \text{previous()} \quad A = m:A' \quad \beta(Wn) = \text{def} \quad \delta(W) = (m, F):W'}{(\text{while Wn b do IP end } (pa, A) \mid \delta, \beta, \square) \overset{m}{\rightsquigarrow} (\text{skip } A' \mid \delta[W/W'], \beta[Wn], \square)}$$

$$[W3r] \quad \frac{m = \text{previous()} \quad A = m:A' \quad \beta(Wn) = \text{und} \quad \delta(W) = (m, T):W' \quad \delta(WI) = (m, C):WI'}{(S \mid \delta, \beta, \square) \overset{m}{\rightsquigarrow} (\text{while Wn T do IP' end } (pa, A') \mid \delta[W/W', WI/WI'], \beta[Wn \Rightarrow IR], \square)}$$

where $S = \text{while Wn b do IP end } (pa, A)$ and $IP' = \text{IreL}(\text{setAI}(\text{IP}, C))$
and $IR = \text{while Wn T do IP' end } (pa, A')$

$$[W4r] \quad \frac{m = \text{previous()} \quad A = m:A' \quad \beta(Wn) = \text{def} \quad \delta(W) = (m, T):W' \quad \delta(WI) = (m, C):WI'}{(S \mid \delta, \beta, \square) \overset{m}{\rightsquigarrow} (\text{while Wn T do IreL(IP) end } (pa, A') \mid \delta[W/W'], \beta[Wn \Rightarrow IR], \square)}$$

where $S = \text{while Wn b do IP end } (pa, A)$
and $IR = \text{while Wn b do IreL(IP) end } (pa, A')$

$$[W5r] \quad \frac{\beta(Wn) = \text{def} \quad (IP \mid \delta, \beta, \square) \overset{\circ}{\rightsquigarrow} (IP' \mid \delta', \beta', \square')}{(\text{while Wn T do IP end } (pa, A) \mid \delta, \beta, \square) \overset{\circ}{\rightsquigarrow} (\text{while Wn T do IP' end } (pa, A) \mid \delta', \beta'', \square')}$$

where $\beta'' = \beta'[\text{refW}(Wn, IP')]$

$$[W6r] \quad \frac{\beta(Wn) = \text{while Wn b do IP end } (pa, A)}{(\text{while Wn T do skip I end } (pa, A) \mid \delta, \beta, \square) \rightsquigarrow_s (\text{while Wn b do IP end } (pa, A) \mid \delta, \beta, \square)}$$

Block

Execution of an inverted block body via [B1r] and the closure of a block via [B2r] is identical to that of the forward execution from Section 4.6.

$$[B1r] \quad \frac{(IP \mid \square) \overset{\circ}{\rightsquigarrow} (IP' \mid \square')}{(\text{begin b1 IP end } \mid \square) \overset{\circ}{\rightsquigarrow} (\text{begin b1 IP' end } \mid \square')}$$

$$[B2r] \quad \overline{(\text{begin } b1 \text{ skip } I \text{ end } | \square) \rightsquigarrow_s (\text{skip } | \square)}$$

Variable and Procedure Declaration

An inverse declaration statement [L1r] (reversal of a forward removal statement) retrieves the final value v' that was held by this variable during the forward execution from the stack ($\delta(X) = (m, v') : X'$), while a fresh memory location l is retrieved ($nextLoc(\sigma) = l$). Using the most direct block name Bn ($pa = Bn : pa'$), the matching entry is created within the variable environment ($\gamma[(X, Bn) \Rightarrow l]$) and the matching memory location is initialised to the value v' ($\sigma[l \mapsto v']$). Finally, the reversal information is removed from δ ($\delta[X/X']$).

$$[L1r] \quad \frac{A = m : A' \quad m = previous() \quad \delta(X) = (m, v') : X' \quad nextLoc(\sigma) = l \quad pa = Bn : pa'}{(\text{var } X = e \text{ (pa, A) } | \delta, \sigma, \gamma, \square) \xrightarrow{m} (\text{skip } A' | \delta[X/X'], \sigma[l \mapsto v'], \gamma[(X, Bn) \Rightarrow l], \square)}$$

An inverse procedure declaration [L2r] modifies the program state by re-creating the basis mapping for this procedure within the procedure environment. No reversal information is required as this is based on the inverted program code.

$$[L2r] \quad \frac{A = m : A' \quad m = previous()}{(\text{proc } Pn \text{ n is IP (pa, A) } | \mu, \square) \xrightarrow{m} (\text{skip } A' | \mu[Pn \Rightarrow (n, IP)], \square)}$$

Variable and Procedure Removals

An inverse removal statement (reversing a declaration) via [H1r] restores the memory location to 0 ($\sigma[l \mapsto 0]$) and removes the entry from γ ($\gamma[(X, Bn)]$). Procedure removal statements (reversing a declaration) via [H2r] remove the mapping from the procedure environment ($\mu[Pn]$).

$$[H1r] \quad \frac{A = m : A' \quad m = previous() \quad \gamma(X, Bn) = l \quad pa = Bn : pa'}{(\text{remove } X = e \text{ (pa, A) } | \delta, \sigma, \gamma, \square) \xrightarrow{m} (\text{skip } A' | \delta, \sigma[l \mapsto 0], \gamma[(X, Bn)], \square)}$$

$$[H2r] \quad \frac{A = m : A' \quad m = previous() \quad \mu(Pn) = def}{(\text{remove } Pn \text{ n is IP (pa, A) } | \mu, \square) \xrightarrow{m} (\text{skip } A' | \mu[Pn], \square)}$$

Procedure Call

An inverted call statement opens via [G1r]. The behaviour is like that of Section 4.6, but with the identifier stacks of the renamed copy of the procedure body populated using the sequence C retrieved from the stack Pr on δ ($\delta(Pr) = (m, C) : Pr'$).

$$[G1r] \quad \frac{A = m : A' \quad m = previous() \quad evalP(n, pa) = Pn \quad \mu(Pn) = (n, IP) \quad \delta(Pr) = (m, C) : Pr'}{(\text{call } Cn \text{ n (pa, A) } | \delta, \mu, \square) \xrightarrow{m} (\text{runc } Cn \text{ IP' end } A' | \delta[Pr/Pr'], \mu[Cn \Rightarrow (n, IP')], \square)}$$

where $IP' = IreP(setAI(IP, C), Cn)$

The execution of a procedure body via [G2r] is unchanged to Section 4.3.

$$[G2r] \frac{(\text{IP} \mid \mu, \square) \overset{\circ}{\rightsquigarrow} (\text{IP}' \mid \mu', \square')}{(\text{runc Cn IP end A} \mid \mu, \square) \overset{\circ}{\rightsquigarrow} (\text{runc Cn IP}' \text{ end A} \mid \mu'[\text{refC}(\text{Cn}, \text{IP}')], \square')}$$

The closure of an inverted block statement (reversing the opening of the forward version) executes and removes the mapping from the procedure environment ($\mu[\text{Cn}]$).

$$[G3r] \frac{A = m:A' \quad m = \text{previous}() \quad \mu(\text{Cn}) = \text{def}}{(\text{runc Cn skip I end A} \mid \mu, \square) \overset{m}{\rightsquigarrow} (\text{skip } m:A \mid \mu[\text{Cn}], \square)}$$

4.10 Examples of Reverse Execution

We return for the final time to our examples from Section 4.4 and Section 4.7. We begin with the procedure call statement executing its body, containing a variable removal statement within a nested block. Note that for all examples in this section, we ignore any previous execution that occurred up to our starting point (and therefore do not show it in the inverted version).

The inverted version of this program is a **runc** construct (as the call is open), a block statement with a single declaration statement (inverse of the removal, and allowed to appear on its own as we omit all other statements of the block body). Recall that the final value held by the location during the forward execution was 0, and that stack **X** on δ had the pair $(m, 3)$ as its head. Let the inverted version **IP** be the program

```

runc c1
  begin b1
    var X = 20 (pa,m:A)
  end
end A1
    
```

and **IT** = **begin b1 var X = 20 (pa,m:A) end**. The inference tree follows.

$$\begin{array}{c}
 \frac{m = \text{previous()} \quad \delta(X) = (m, v') : X' \quad \text{nextLoc}(\sigma) = 1 \quad \text{pa} = \text{b1:pa}'}{(\text{var X} = 20 \text{ (pa,m:A)} \mid \delta, \sigma, \gamma, \mu, \square) \overset{m}{\rightsquigarrow} (\text{skip A} \mid \delta', \sigma', \gamma', \mu', \square)} \quad [L1r] \\
 \frac{\quad}{(\text{IT} \mid \delta, \sigma, \gamma, \mu, \square) \overset{m}{\rightsquigarrow} (\text{begin b1 skip m:A end} \mid \delta', \sigma', \gamma', \mu', \square)} \quad [B1r] \\
 \frac{\quad}{(\text{IP} \mid \delta, \sigma, \gamma, \mu, \square) \overset{m}{\rightsquigarrow} (\text{runc c1 begin b1 skip m:A end end A}_1 \mid \delta', \sigma', \gamma', \mu', \square)} \quad [G2r]
 \end{array}$$

The premise $A = m:A'$ is omitted from the leaf [L1r] above as we write this directly into the statement. The declaration of this variable recreates the variable within γ , namely $\gamma' = \gamma[(X, b1) \mapsto 1]$, and re-initialises this location to the final value 3 held by the variable during forward execution, namely $\sigma' = \sigma[1 \mapsto 3]$. Finally, this old value is used and therefore removed from the stack X on δ , namely $\delta' = \delta[X/X']$. Therefore the state is restored to as it was prior to the matching forward transition.

Now consider our program containing a call statement ready to close nested within two blocks. We note that **AQ** and **AR** from Section 4.7 must be inverted prior to this step and so are omitted. The inverse of closing a call statement is to open it, meaning we use **IT** to represent the procedure body that would be inverted after this step. Let **IP** be the program

```
begin b1
  begin b2
    call c1 n (pa,m:A)
  end
end
```

and $IT_1 = \text{begin b2 call c1 n (pa,m:A) end}$. The following inference tree shows the desired transition exists.

$$\begin{array}{c}
\frac{m = \text{previous()} \quad evalP(n, pa) = Pn \quad \mu(Pn) = (n, IT) \quad \delta(Pr) = (m, C) : Pr'}{[G1r]} \\
\frac{(\text{call c1 n (pa,m:A)} \mid \delta, \mu, \square) \xrightarrow{m} (\text{runc c1 IT' end A} \mid \delta', \mu', \square)}{[B1r]} \\
\frac{(\text{IT}_1 \mid \delta, \mu, \square) \xrightarrow{m} (\text{begin b2 runc c1 IT' end A end} \mid \delta', \mu', \square)}{[B1r]} \\
(\text{IP} \mid \delta, \mu, \square) \xrightarrow{m} (\text{begin b1 begin b2 runc c1 IT' end A end end} \mid \delta', \mu', \square)
\end{array}$$

The premise $A = m:A'$ is omitted from the leaf [G1r] as above. The copy of the procedure body unique for this call statement is recreated, namely $\mu' = \mu[Cn \Rightarrow (n, T')]$, where T' is the renamed version of the procedure body with all identifier stacks populated using the sequence C from the stack WI ($T' = IreP(setAI(IP, C), Cn)$). This reversal information is then removed from the stack Pr , meaning $\delta' = \delta[Pr/Pr']$. We also note this highlights the mismatch of rules during a forward and reverse execution, namely [S1r].

Example 17 returns to the complete annotated execution shown in Example 15 and describes the corresponding reverse execution.

Example 17. (Complete inverse execution) The inverted version of the program used in Example 15 is given in Figure 4.9(b) (page 84). Note the inverted statement order and that all declaration statements are now removals (and vice versa). The

block statement executes via [B1r], beginning with the re-declaration of two local variables. This is via [L1r], which ignores the value contained within the statement (namely 2), and instead initialises this to old values retrieved from the stacks **X** and **Y** respectively. Therefore **X** is initialised to 4, and **Y** to 5 (with both pieces of reversal information removed). The procedure is re-declared via [L2r] (recreating the basis mapping using identifier 12), before the assignment **Y** = 5 is reversed (not evaluating the expression and instead restoring it to the value 2 from the stack **Y**). The call statement then is then opened via [G1r], which recreates the copy of this procedure and populates all stacks with the identifiers from **Pr**. The procedure environment μ contains two mappings, namely the basis mapping $\mathbf{p1.0} \mapsto \mathbf{res} = \mathbf{X} + \mathbf{Y} \text{ []}$ and the call-specific copy $\mathbf{c1.0} \mapsto \mathbf{res} = \mathbf{X} + \mathbf{Y} \text{ [9]}$. Identifier 9 is then used to reverse the assignment to **res**, before the call closes via [G3r] that removes the mapping for **c1.0** from μ . The assignment **X** = **X** + **Y** is then reversed (restoring **X** to 2 from the stack **Y**). Finally, the procedure and both local variables are removed via [H2r] and [H1r] respectively, before the block itself closes via [B2r]. The execution is complete, producing the desired state such that **res** = 0 and that μ is empty. We have correctly restored both the program state, and the auxiliary store as all reversal information has been used, to as they were prior to the forward execution.

The inverted execution described here has been executed using our simulator, which shows the behaviour outlined above to be correct. ■

4.11 Correctness of Reversal

Maintaining the definition of correctness from the previous chapter, namely Definition 3.8.1, we again focus on two properties of our approach. The first being that annotation does not change the behaviour of the original program, and the second that the inverse execution correctly restores the program state and uses all reversal information (garbage free). We note that all lemmas from Section 3.8 of Chapter 3 are extended to hold for the updated programming language, with all references from this point on being to the extended versions. Chapter 6 details necessary prerequisites, before stating formally each of these properties and its proof.

4.12 Conclusion

We have introduced an extended version of our programming language, adding support for blocks statements containing local variables and procedures. The program state has been updated to reflect this, extending the renaming process to handle potentially recursive procedure calls. All reversal information required for these additional constructs has been described and the necessary stacks added to the

auxiliary store. Annotation has been extended, alongside the forward operational semantics. Inversion has also been updated, allowing the extension of the reverse operational semantics to be defined. All smaller results given in Chapter 3 are extended accordingly, with the proof of correctness deferred to Chapter 6.

Chapter 5

Challenges of Proving Correctness

Chapter 4 has introduced an approach to reversing executions of an imperative concurrent programming language. Two key processes have been used, namely annotation and inversion. It is important to prove that both of these processes, and therefore our method of reversibility, is correct. Before stating the properties that we will use to determine correctness, we first address multiple challenges. In this chapter, each of the three challenges is discussed in turn, with appropriate solutions described.

Each of these challenges concerns the reverse execution of programs, and specifically our Inversion Result (introduced later in Chapter 6). The three key challenges we will focus on are:

1. Parallel composition may allow different interleaving orders;
2. Partially executed programs may not reach skip when inverted;
3. Stopping the inverse execution at the point corresponding to the beginning of the forward execution.

Each of these issues is now discussed, alongside the proposed solutions.

5.1 Multiple Interleaving Orders

The semantics of parallel composition (see Chapter 4) means multiple distinct execution orders exist. Many of these executions can be described as equivalent, where the order of identifier steps is identical while the order of skip steps is different. Since no skip steps alter the program state (Lemma 1 of Chapter 3), each of these executions produces the same final program state (hence equivalent). Inverse skip steps can also be applied in several different orders while preserving an equivalent execution. This results in the potential for constructs to remain open during an

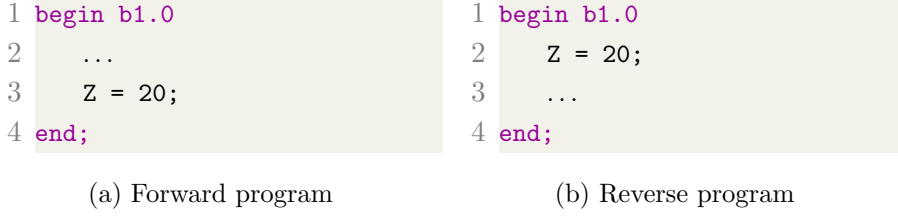


Figure 5.1: A program and its inverse with non-matching skip steps

inverse execution for longer than during the forward execution, adding difficulty to the task of matching the rules applications between a forward and reverse execution.

Skip steps used within a forward execution may not have the one-to-one correspondence with those of the matching reverse execution, as highlighted in Example 18.

Example 18. (Non-matching skip steps) Consider the forward program in Figure 5.1(a), whose execution concludes with the final assignment. The transition sequence of this is [D1a] (to perform the assignment that reaches skip), and then [B1a] to close the block. Now consider the inverted program from Figure 5.1(b). The transition sequence of this execution begins with [D1r] (to reverse the assignment) and then [S2r] (to continue on into the block body). Therefore there is a mismatch between [B1a] and [S2r]. Note that if the assignment is the only statement within the block, skip steps will match as required ([B2r]). ■

We avoid these difficulties by considering executions as a composition of segments. Firstly let any arbitrary execution be named a *standard execution*. We now introduce a *uniform execution*, where each identifier step is followed by all of the possible closing skip steps (named a segment). All skip steps within a segment are said to have been ‘caused’ by the preceding identifier step.

Definition 5.1.1. (Uniform execution) A uniform execution is a standard program execution where all skip steps (transitions that do not use identifiers) are performed as soon as they are available. A step of a uniform execution is represented using $U \xrightarrow{\circ}$ and $U \xrightarrow{\circ\sim}$ for forward and reverse execution respectively.

We remark that skip steps within a uniform execution are not interleaved across a parallel composition. This is due to identifier steps executed on one side not being able to make skip steps available on the other. Therefore any two equivalent standard executions (that is two transition sequences where the order of identifier steps is identical, with skip steps potentially different) will have the same uniform version. All skip steps from each sequence will be performed as soon as they are available, which will be identical in both cases. Consider Example 19 that shows a uniform version of a standard execution.

<pre> 1 par { 2 begin b1.0 3 x = 20 [1]; 4 end; 5 } { 6 y = 2 [0]; 7 z = 4 [2]; 8 } </pre>	Standard	Uniform
	[D1a] (0)	[D1a] (0)
	[D1a] (1)	[S2a] (of 0)
	[S2a] (of 0)	[D1a] (1)
	[D1a] (2)	[B2a] (of 1)
	[B1a] (of 1)	[D1a] (2)
	[P3a]	[P3a]

(a) Concurrent program (b) Standard and uniform execution

Figure 5.2: A program, a standard execution and its uniform version

Example 19. (Standard and uniform executions) Consider the concurrent program in Figure 5.2(a), where identifiers 0–2 capture one available standard execution. Figure 5.2(b) shows first the order of rule applications under this standard execution, with each rule shown alongside the identifier it used (or that caused it). The rules [S2a] and [B2a] are not executed as soon as available and instead deferred to later in this execution. The corresponding uniform execution is also shown in Figure 5.2(b), where the order of identifier rules is unchanged (meaning the final program states match) and the skip steps are now performed as soon as available. ■

One further consideration is of programs that begin with skip steps. This is a special case of a uniform execution, where all skip steps performed at the beginning have not been ‘caused’ by any identifier step. Consider Example 20 that shows a program beginning with several skip steps.

Example 20. (Program beginning with skip steps) The program in Figure 5.3 contains the parallel composition of two sub-programs, each beginning with skip steps. The left side starts with an empty block statement that can be closed via the skip step [B2a] (leaving the remaining program AP). The right side starts with a block that is not empty, with a block body beginning with a skip statement. The skip step [S2a] can remove this skip statement leaving the block with the remaining body AQ. ■

Such an execution performs all initial skip steps in a fixed order, namely all from the left side of a parallel before those of the right in a depth first manner. This ensures all equivalent executions have a single uniform version. Lemma 4 states that for any standard forward execution, there exists a uniform forward execution that has the same behaviour (with respect to the program state).

Lemma 4. (Equivalent uniform execution) Given an arbitrary standard forward or reverse program execution, there exists an equivalent uniform execution that

```

1 begin b1.0                begin b2.0
2   skip;                   skip;
3 end;                      AQ
4 AP                        end;

```

Figure 5.3: A complete program beginning with several skip steps

modifies the program state equivalently. Let P be an arbitrary program and \square be the tuple $(\sigma, \gamma, \mu, \beta, \delta)$ of initial program state environments.

1. If $(P \mid \square) \xrightarrow{\circ}^* (\text{skip } I \mid \square')$ for some I and final program state \square' , then there exists an equivalent uniform execution $(P \mid \square) \xrightarrow{U}^{\circ}^* (\text{skip } I' \mid \square'')$ for I' and program state \square'' such that $I' = I$ and $\square'' = \square'$.
2. If $(P \mid \square) \xrightarrow{\sim}^* (\text{skip } I \mid \square')$ for some I and final program state \square' , then there exists an equivalent uniform execution $(P \mid \square) \xrightarrow{U}^{\sim}^* (\text{skip } I' \mid \square'')$ for I' and program state \square'' such that $I' = I$ and $\square'' = \square'$.

Proof. The proof of Lemma 4 is by induction on the length of the execution using Lemma 5 below. All executions are assumed to be of complete programs. We consider each of the two possible cases of execution, beginning with the special case of those that start with any number of skip steps. Such a complete execution, written $(P \mid \square) \rightarrow_s^* (P' \mid \square) \xrightarrow{\circ}^* (\text{skip } I \mid \square')$, exists such that $(P \mid \square) \rightarrow_s^* (P' \mid \square)$ contains all possible skip steps and that $(P' \mid \square) \xrightarrow{\circ}^* (\text{skip } I \mid \square')$ therefore begins with an identifier step. Since this execution is complete, all of the initial skip steps do not have a preceding identifier step that caused them. Therefore the beginning skip steps can be reordered as shown in Part 1 of Lemma 5, such that all skip steps from the left side of a parallel occur before those from the right. With the execution $(P \mid \square) \rightarrow_s^* (P' \mid \square)$ reordered into a uniform equivalent, application of the induction hypothesis to the remaining shorter execution $(P' \mid \square) \xrightarrow{\circ}^* (\text{skip } I \mid \square')$ (guaranteed to begin with an identifier step) shows the existence of a uniform equivalent.

We now consider executions of the form $(P \mid \square) \xrightarrow{m} (P' \mid \square) \rightarrow_s^* (P'' \mid \square) \xrightarrow{m+1} (P''' \mid \square) \xrightarrow{\circ}^* (\text{skip } I \mid \square')$ that begin with an identifier step. To support use of the induction hypothesis, we do not require this execution to be complete. The execution $(P' \mid \square) \rightarrow_s^* (P'' \mid \square)$ may contain skip steps that were caused by a previous identifier step (using an identifier k such that $k < m$ provided it exists). Using Part 2 of Lemma 5, all such skip steps can be reordered to appear prior to the identifier step using m . All other skip steps within the execution $(P' \mid \square) \rightarrow_s^* (P'' \mid \square)$ are guaranteed to have been caused by the transition \xrightarrow{m} . These cannot have been caused by an identifier rule that has not yet happened, meaning their current position already respects uniformity. The final consideration is of skip steps within $(P''' \mid \square) \xrightarrow{\circ}^* (\text{skip } I \mid \square')$. It is possible for some of these steps to have

been caused by any of the preceding identifier steps and are therefore required to be reordered. As before, this can be achieved using Part 2 of Lemma 5. From here, the remaining shorter execution beginning with the identifier step using $m + 1$ can be shown to be uniform via the induction hypothesis. Therefore Lemma 4 holds. \square

Lemma 5. Given a sequence of two transitions where one is from the opposite side of a parallel to the other, these two transitions may be reordered in two specific circumstances (depending on the type of each transition). Let P be an annotated program and \square be the tuple $(\sigma, \gamma, \mu, \beta, \delta)$ of initial program state environments.

1. If $(P \mid \square) \rightarrow_{S_1} (P' \mid \square) \rightarrow_{S_2} (P''' \mid \square)$ exists, for some programs P' and P''' , such that the transition \rightarrow_{S_1} is from the opposite side of a parallel composition to the transition \rightarrow_{S_2} , then there exists the reordered execution $(P \mid \square) \rightarrow_{S_2} (P'' \mid \square) \rightarrow_{S_1} (P''' \mid \square)$ for some program P'' .
2. If $(P \mid \square) \xrightarrow{m} (P' \mid \square') \rightarrow_S (P''' \mid \square')$ exists, for some programs P' , P''' and program state \square' , such that the transition \xrightarrow{m} is from the opposite side of a parallel composition to the transition \rightarrow_S , then there exists the reordered execution $(P \mid \square) \rightarrow_S (P'' \mid \square) \xrightarrow{m} (P''' \mid \square')$ for some program P'' .

Proof. Recall Lemma 1 from Chapter 3 that states skip steps do not change the program state in any way, and the observation that any step from one side of a parallel composition cannot ‘cause’ skip steps on the other side of the same parallel. Consider Part 1 of Lemma 5. The execution of two skip steps with one on each side of a parallel means that neither skip step is the cause of the other. With one of the steps requiring [P1a] and the other needing [P2a] to occur within its inference tree, the definition of these rules (Chapter 4) shows the execution of one cannot affect the execution of the other. This means the order of these skip steps cannot change the final program state or structure, meaning the reordering of these is permitted. The intermediate program P'' is produced and is like P''' , but with the side of the parallel that contains \rightarrow_{S_1} still in its initial form.

Consider Part 2 of Lemma 5. With each step being on opposite sides of a parallel, the skip step \rightarrow_s cannot have been caused by the identifier step \xrightarrow{m} . This means that the skip step must have been caused by an identifier step \xrightarrow{k} such that $k < m$. As in Part 1, it is therefore guaranteed that both steps were available prior to the execution of either. The rules [P1a] and [P2a] show that the program structure of one side of a parallel cannot be changed via the other side. With the skip step having no effect on the program state, the identifier step \xrightarrow{m} and the skip step \rightarrow_s can therefore be reordered. The intermediate program P'' is like P''' but with the side containing the identifier step \xrightarrow{m} still in its original form. Since both parts have been shown to be valid, we can conclude that Lemma 5 holds, as required. \square

```

PP ::= ε | PS | PP; AP | PP par PP
PS ::= skip I | X = E (pa,A) | if In B B then AP AP else AP AP end (pa,A) |
      if In T B then PP AP else AP AP end (pa,A) |
      if In F B then AP AP else PP AP end (pa,A) |
      while Wn B do AP end (pa,A) | while Wn T do PP end (pa,A) |
      begin Bn ABB ABB end | begin Bn PBB ABB end |
      call Cn n (pa,A) | runc Cn AP AP end A | runc Cn PP AP end A
PBB ::= ADV; ADP; AP; ARP; ARV; | ADP; AP; ARP; ARV; | AP; ARP; ARV; | ARP; ARV; | ARV;

```

Figure 5.4: Syntax of partially executed programs

Lemma 6 corresponds to Lemma 4, showing that any uniform reverse execution can be relaxed into an equivalent standard version. Each uniform execution is standard by definition, meaning this trivial result is included here for completeness.

Lemma 6. (Equivalent standard execution) Given an arbitrary uniform forwards or reverse program execution, there exists an equivalent standard execution that modifies the program state equivalently. Let P be an arbitrary program and \square be the tuple $(\sigma, \gamma, \mu, \beta, \delta)$ of initial program state environments.

1. If $(P \mid \square) \xrightarrow{U}^* (\text{skip } I \mid \square')$ for some I and program state \square' , then there exists an equivalent standard execution $(P \mid \square) \xrightarrow{\circ}^* (\text{skip } I' \mid \square'')$ for I and program state \square'' such that $I' = I$ and $\square'' = \square'$.
2. If $(P \mid \square) \xrightarrow{U}^{\circ} (\text{skip } I \mid \square')$ for some I and program state \square' , then there exists an equivalent standard execution $(P \mid \square) \xrightarrow{\circ}^* (\text{skip } I' \mid \square'')$ for I and program state \square'' such that $I' = I$ and $\square'' = \square'$.

Proof. Correspondingly to the proof of Lemma 4, this is via induction on the length of an execution using Lemma 5. \square

5.2 Partial Executions

Later in Chapter 6, we introduce Theorem 8 that states that given an annotated execution that completes and reaches skip, the corresponding inverted program also completes and reaches skip. This is valid for all executions of complete programs.

Consider the process of proving this by induction on the length of an execution. This typically requires a small number of initial execution steps to be explicitly stated, before using the induction hypothesis on the remaining execution. After the initial steps, the remaining program may no longer be complete, and instead be a *partially executed program* (see Definition 5.2.1 and Example 21).


```

1 begin b1.0
2   var X = 20 [];
3   var Y = 40 [];
4   Z = Z + X [];
5   Z = Z + Y [];
6   remove X = 20 [];
7   remove Y = 40 [];
8 end;

```

(a) Complete program

```

1 begin b1.0
2   Z = Z + Y [];
3   remove X = 20 [];
4   remove Y = 40 [];
5 end;

```

(b) Partially executed version

Figure 5.5: An original program and a possible partially executed version

Definition 5.2.1. (Partially executed program) A partially executed program is a program produced as a result of partial execution of a complete program, respecting uniformity. Figure 5.4 gives the syntax of partially executed programs.

A partially executed program, often referred to as a partial program, can contain conditions and expressions in evaluated form and does not require a one-to-one match between declaration and removal statements. The syntax in Figure 5.4 shows conditional and **runc** statements contain a second copy of their sub-programs, used within functions defined later to determine whether the execution of a statement has already started. For example, the function inv^+ () introduced later must invert a conditional statement differently if its execution has already started, compared to when its execution has not. Partial programs respect uniformity meaning that the initial execution that produces it contains all of the available closing skip steps. Therefore executions only stop between segments, where each segment is an identifier step and all skip steps caused by it. Each partial program begins with an identifier step as a result, and is executed using the (same) operational semantics from Chapter 4.

Example 21. (Partial program) The complete program (containing only a block statement) in Figure 5.5(a) has a matching removal statement for each declaration. Now consider the partial execution of this, performing all declaration statements and the first assignment of the block body. The remaining program, shown in Figure 5.5(b), now violates the assumption of complete programs as there is not a matching declaration statement for each removal. ■

The reversal of a partial program may not reach skip. For example, the forward execution of a conditional statement beginning from the middle of the true branch is reversed by inverting only the execution that happens from this beginning position. Therefore the inverse conditional statement remains open (cannot invert the opening of the forward condition) and therefore cannot reach skip. This prevents the use of the induction hypothesis on a partial program as its reversal cannot reach skip.

$$SE(\text{skip}) = \text{skip} \quad (5.1)$$

$$SE(\text{abort}) = \text{abort} \quad (5.2)$$

$$SE(AS; AP) = se(AS); AP \quad (5.3)$$

$$SE(AP \text{ par } AP) = SE(AP) \text{ par } SE(AP) \quad (5.4)$$

$$se(X = e \text{ (pa,A)}) = \text{skip (pa,A)} \quad (5.5)$$

$$se(\text{if In b b then AP AP else AQ AQ end (pa,A)}) = \text{skip (pa,A)} \quad (5.6)$$

$$se(\text{if In T b then AP' AP else AQ AQ end (pa,A)}) \quad (5.7)$$

$$= \text{if In T b then } SE(AP') \text{ AP else AQ AQ end (pa,A)}$$

$$se(\text{if In F b then AP AP else AQ' AQ end (pa,A)}) \quad (5.8)$$

$$= \text{if In F b then AP AP else } SE(AQ') \text{ AQ end (pa,A)}$$

$$se(\text{while Wn b do AP end (pa,A)}) = \text{skip if not started} \quad (5.9)$$

$$se(\text{while Wn b do AP end (pa,A)}) = \text{while Wn b do AP end (pa,A) if started} \quad (5.10)$$

$$se(\text{while Wn T do AP end (pa,A)}) = \text{while Wn T do } SE(AP) \text{ end (pa,A) if started AP} \quad (5.11)$$

$$se(\text{begin Bn AP AP end (pa,A)}) = \text{skip (pa,A)} \quad (5.12)$$

$$se(\text{begin Bn AP AOP end (pa,A)}) = \text{begin Bn } SE(AP) \text{ AOP end (pa,A) if AP} \neq \text{AOP} \quad (5.13)$$

$$se(\text{var X = E (pa,A)}) = \text{skip (pa,A)} \quad (5.14)$$

$$se(\text{proc Pn n is AP (pa,A)}) = \text{skip (pa,A)} \quad (5.15)$$

$$se(\text{remove X = E (pa,A)}) = \text{skip (pa,A)} \quad (5.16)$$

$$se(\text{remove Pn n is AP (pa,A)}) = \text{skip (pa,A)} \quad (5.17)$$

$$se(\text{call Cn n (pa,A)}) = \text{skip (pa,A)} \quad (5.18)$$

$$se(\text{runc Cn AP AP (pa,A)}) = \text{runc Cn } SE(AP) \text{ AP (pa,A)} \quad (5.19)$$

Figure 5.6: Definition of the function $SE()$

Definition 5.2.2 introduces *skip equivalent* as the name given to the corresponding program code of the inverted execution of a partial program (note the **abort** statement that is introduced later). Example 22 shows a skip equivalent.

Definition 5.2.2. (Skip equivalent) A skip equivalent of a statement is either a single skip statement, a single abort statement or an identical version of the statement where one or more sub-programs are either skip, abort or suitable skip equivalents (abort statements are introduced in the following section). The notation $SE(T)$ represents a skip equivalent of a statement T, where $SE(T)$ is defined in Figure 5.7.

Example 22. (Skip equivalent) Consider an annotated program $AP = \text{if i1 T then AQ else AR end (pa,A)}$, where AQ is a partially executed version of the true branch. The inverse version $IP = inv(AP)$ should only invert AQ and none of the previous true branch, leaving the conditional open. Therefore the skip equivalent that corresponds to this starting position is $\text{if i1 T then skip I else } inv(AR) \text{ end (pa,A)}$ (omitting second copies). ■

5.3 Stopping an Inverted Execution

With inverted partial programs not guaranteed to reach skip and instead a skip equivalent, we must ensure a reverse execution stops at the required point within the proofs to follow in Chapter 6. For the partial forward execution to perform correctly, all previous steps of execution are assumed to have happened. With all reversal information of this previous execution available, the reverse execution may (unintentionally) continue past the desired skip equivalent, as in Example 23.

Example 23. (Inverse execution that does not stop appropriately) Recall the partial program from Figure 5.5(b) of Example 21. The inverse execution of this should invert only the execution that occurs from this point, namely the final assignment of the block body and both removal statements. The inverse version can be generated based on this partial program, where the inverse block body contains only those statements to invert. When each statement has executed, the block will be able to incorrectly close via the skip step [B2r], which leads to an incorrect program structure (the starting position had the block open). Problems also arise if the inverse version is based on the full program, since skip or identifier steps may apply (which are applicable due to the assumption that all previous execution has occurred). ■

Therefore the reverse execution of a program is stopped using an *abort statement* which is added into the syntax and written as **abort**. This has undefined behaviour and forcibly stops an execution that has only abort statement(s) as the next available step. Such **abort** statements cannot appear in original programs and are only used within the proof to follow. Any syntax now referenced contains abort statements.

We now define an extended inversion function inv^+ () in Figure 5.7, that takes an executed annotated program (either complete or partial) and returns the corresponding inverted version that contains the necessary abort statement to forcibly stop the execution of it at the desired skip equivalent. The function inv^+ () calls the original inversion function $inv()$ on all of the given program except the first statement. This statement may be partial meaning its execution may need to be stopped via an abort. Should the statement be complete (not yet started), the abort statement is sequentially composed to it. If the statement has started, the abort statement is inserted at the corresponding position of the appropriate sub-program.

Manually inserting the abort statement via the inversion function inv^+ () is sufficient in all cases except those of while loops or procedure calls. Recall the semantics of each from Chapter 4, where a copy is maintained within the while or procedure environment respectively. Adding an abort statement into the inverse version will mean that all copies of either the loop or the procedure contain it. In the case of a loop, there are potentially many iterations that should be inverted fully prior the one that only requires partial execution. Only the iteration of interest should

$$inv^+(S;P, \square) = inv(P); i^+(S, \square) \quad (5.20)$$

$$inv^+(P \text{ par } Q, \square) = inv^+(P, \square) \text{ par } inv^+(Q, \square) \quad (5.21)$$

$$i^+(\text{skip } I, \square) = \text{skip } I; \text{ abort} \quad (5.22)$$

$$i^+(X = e \text{ (pa,A)}, \square) = X = e \text{ (pa,A)}; \text{ abort} \quad (5.23)$$

$$\begin{aligned} i^+(\text{if In ob ob then AP AP else AQ AQ end (pa,A)}, \square) \\ = \text{if In ob ob then } inv(AP) \text{ } inv(AP) \text{ else } inv(AQ) \text{ } inv(AQ) \text{ end (pa,A)}; \text{ abort} \end{aligned} \quad (5.24)$$

$$\begin{aligned} i^+(\text{if In T ob then AP AP' else AQ AQ end (pa,A)}, \square) \\ = \text{if In ob ob then } inv^+(AP) \text{ } inv(AP') \text{ else } inv(AQ) \text{ } inv(AQ) \text{ end (pa,A)} \end{aligned} \quad (5.25)$$

where $AP \neq \text{skip } I$ and $AP \neq AP'$

$$\begin{aligned} i^+(\text{if In F ob then AP AP else AQ AQ' end (pa,A)}, \square) \\ = \text{if In ob ob then } inv(AP) \text{ } inv(AP) \text{ else } inv^+(AQ) \text{ } inv(AQ') \text{ end (pa,A)} \end{aligned} \quad (5.26)$$

where $AQ \neq \text{skip } I$ and $AQ \neq AQ'$

$$\begin{aligned} i^+(\text{if In T ob then skip I AP else AQ AQ end (pa,A)}, \square) \\ = \text{if In ob ob then abort } inv(AP) \text{ else } inv(AQ) \text{ } inv(AQ) \text{ end (pa,A)} \end{aligned} \quad (5.27)$$

$$\begin{aligned} i^+(\text{if In F ob then AP AP else skip I AQ end (pa,A)}, \square) \\ = \text{if In ob ob then } inv(AP) \text{ } inv(AP) \text{ else abort } inv(AQ) \text{ end (pa,A)} \end{aligned} \quad (5.28)$$

$$\begin{aligned} i^+(\text{while Wn b do AP end (pa,A)}, \square) = \text{while Wn b do } inv(AP) \text{ end (pa,A)}; \text{ abort} \\ \text{where } \beta(Wn) = \text{und} \end{aligned} \quad (5.29)$$

$$\begin{aligned} i^+(\text{while Wn b do AP' end (pa,A)}, \square) = \text{while Wn ob do } inv(AP) \text{ end (pa,A)} \\ \text{where AP is such that } \beta(Wn) = \text{while Wn ob do AP end (pa,A)} \end{aligned} \quad (5.30)$$

$$i^+(\text{begin Bn AP AP end, } \square) = \text{begin Bn } inv(AP) \text{ } inv(AP) \text{ end}; \text{ abort} \quad (5.31)$$

$$\begin{aligned} i^+(\text{begin Bn AP AP' end, } \square) = \text{begin Bn } inv^+(AP) \text{ } inv(AP) \text{ end} \\ \text{where } AP \neq AP' \end{aligned} \quad (5.32)$$

$$i^+(\text{var X = E (pa,A)}, \square) = \text{remove X = E (pa,A)}; \text{ abort} \quad (5.33)$$

$$i^+(\text{proc Pn n is AP (pa,A)}, \square) = \text{remove Pn n is } inv(AP) \text{ (pa,A)}; \text{ abort} \quad (5.34)$$

$$i^+(\text{remove X = E (pa,A)}, \square) = \text{var X = E (pa,A)}; \text{ abort} \quad (5.35)$$

$$i^+(\text{remove Pn n is AP (pa,A)}, \square) = \text{proc Pn n is } inv(AP) \text{ (pa,A)}; \text{ abort} \quad (5.36)$$

$$i^+(\text{call Cn n (pa,A)}, \square) = \text{call Cn n (pa,A)}; \text{ abort} \quad (5.37)$$

$$i^+(\text{runc Cn AP AP' (pa,A)}, \square) = \text{call Cn n (pa,A)} \quad (5.38)$$

Figure 5.7: Definition of the function $inv^+(\cdot)$

contain the abort. Similarly for call statements, only the specific copy should be forcibly stopped while all others execute completely.

In each of these cases, we require modified versions of selected transition rules from the semantics in Chapter 4. For each appropriate rule $[r]$, the modified versions are named $[rP]$ and $[rPI]$ (given below). Each modified rule is unambiguous with respect to each other and the original versions through the use of distinct premises and abort statements (that cannot appear within normal execution).

While Loop

Consider a partial execution beginning with the next evaluation of the condition (not the first). This can be the last check (rule [W2a]) or any other except the first (rule [W4a]). The modified version of each, namely [W2aP] and [W4aP], saves a true abort flag (to indicate reverse execution should stop here) within the element of the boolean sequence needed for inversion (the extended stack W that now stores triples such that $\delta[(m, T, T) \rightarrow W]$). Note all following iterations use the original rule and save a false abort flag (or nothing) (each modified rule requires the loop body to be **abort**, which is disallowed in normal execution and only within the proof).

[W2aP]

$$\frac{m = next() \quad \beta(Wn) = def \quad (b \text{ pa} \mid \beta, \square) \hookrightarrow_b^* (F \mid \beta, \square) \quad AP = \text{abort}}{(S \mid \delta, \beta, \square) \xrightarrow{m} (\text{skip } m:A \mid \delta[(m, T, T) \rightarrow W, (m, C) \rightarrow WI], \beta[Wn], \square)}$$

where $S = \text{while } Wn \text{ b do } AP \text{ end } (pa, A)$ and $C = getAI(\beta(Wn))$

[W4aP]

$$\frac{m = next() \quad \beta(Wn) = \text{while } Wn \text{ b do } AP' \text{ end } (pa, A) \quad (b \text{ pa} \mid \beta, \square) \hookrightarrow_b^* (T \mid \beta, \square) \quad AP = \text{abort}}{(S \mid \delta, \beta, \square) \xrightarrow{m} (\text{while } Wn \text{ T do } reL(AP') \text{ end } (pa, m:A) \mid \delta[(m, T, T) \rightarrow W], \beta[Wn \Rightarrow AR], \square)}$$

where $S = \text{while } Wn \text{ b do } AP \text{ end } (pa, A)$ and $AR = \text{while } Wn \text{ b do } reL(AP') \text{ end } (pa, m:A)$

The matching inverse rules, namely [W3rP] and [W4rP], can only be used in cases where a true abort flag is present (meaning one of the rules above must have been executed) and return a loop containing abort as its body. Therefore no further execution is possible and we have successfully stopped the execution.

[W3rP]

$$\frac{m = previous() \quad A = m:A' \quad \beta(Wn) = und \quad \delta(W) = (m, T, T):W' \quad \delta(WI) = (m, C):WI'}{(S \mid \delta, \beta, \square) \xrightarrow{m} (\text{while } Wn \text{ T do abort end } (pa, A') \mid \delta[W/W', WI/WI'], \beta[Wn \Rightarrow AR], \square)}$$

where $S = \text{while } Wn \text{ b do } IP \text{ end } (pa, A)$

and $AR = \text{while } Wn \text{ b do } IreL(setAI(IP, C)) \text{ end } (pa, A')$

[W4rP]

$$\frac{m = previous() \quad A = m:A' \quad \beta(Wn) = def \quad \delta(W) = (m, T, T):W'}{(S \mid \delta, \beta, \square) \xrightarrow{m} (\text{while } Wn \text{ T do abort end } (pa, A') \mid \delta[W/W'], \beta[Wn \Rightarrow AR], \square)}$$

where $S = \text{while } Wn \text{ b do } IP \text{ end } (pa, A)$ and $AR = \text{while } Wn \text{ b do } reL(IP) \text{ end } (pa, A')$

Now consider a partial program that begins part way through a loop iteration (execution of the loop body). Likewise to above, the use of an abort flag can indicate

the inverse iteration that requires partial inversion. The difference here is that the partial loop body performed forwards must also be reversed. In order to determine when to stop within the loop body, this partial version of the loop body PP is saved alongside the abort flag (note the further extension to stack W on δ such that $\delta[(m, T, T, PP) \rightarrow W]$). This must be saved during the next condition check (as we save one element behind) and therefore is required to be available after its execution. Therefore the while loop is extended to contain a copy of the partial program PP that remains unchanged during execution. Such statements do not appear during normal execution (due to abort) and only within the proofs to follow in Chapter 6.

[W2aPI]

$$\frac{m = next() \quad \beta(Wn) = def \quad (b \text{ pa} \mid \beta, \square) \hookrightarrow_b^* (F \mid \beta, \square) \quad AP = abort \quad PP \neq skip}{(S \mid \delta, \beta, \square) \xrightarrow{m} (skip \ m:A \mid \delta[(m, T, T, PP) \rightarrow W, (m, C) \rightarrow WI], \beta[Wn], \square)}$$

where $S = \text{while } Wn \ b \ \text{do } AP \ PP \ \text{end} \ (pa, A)$ and $C = getAI(\beta(Wn))$

[W4aPI]

$$\frac{m = next() \quad \beta(Wn) = \text{while } Wn \ b \ \text{do } AP' \ AP'' \ \text{end} \ (pa, A) \quad (b \text{ pa} \mid \beta, \square) \hookrightarrow_b^* (T \mid \beta, \square) \quad AP = abort \quad PP \neq skip}{(S \mid \delta, \beta, \square) \xrightarrow{m} (\text{while } Wn \ T \ \text{do } AP_1 \ skip \ \text{end} \ (pa, m:A) \mid \delta[(m, T, T, PP) \rightarrow W], \beta[Wn \Rightarrow AR], \square)}$$

where $S = \text{while } Wn \ b \ \text{do } AP \ PP \ \text{end} \ (pa, A)$ and $AP_1 = reL(AP')$

and $AR = \text{while } Wn \ b \ \text{do } AP_1 \ skip \ \text{end} \ (pa, m:A)$

The matching inverse rules, namely [W3rPI] and [W4rPI], are only available provided a true abort flag and a partial program have been saved. The loop body of the remaining execution is replaced with this saved partial program, with an abort statement added (via the function inv^+) to stop the execution as required.

[W3rPI]

$$\frac{m = previous() \quad A = m:A' \quad \beta(Wn) = und \quad \delta(W) = (m, T, T, PP):W' \quad \delta(WI) = (m, C):WI' \quad PP \neq skip}{(S \mid \delta, \beta, \square) \xrightarrow{m} (\text{while } Wn \ T \ \text{do } IreL(PP_1) \ skip \ \text{end} \ (pa, A') \mid \delta[W/W', WI/WI'], \beta[Wn \Rightarrow AR], \square)}$$

where $S = \text{while } Wn \ b \ \text{do } IP \ PP' \ \text{end} \ (pa, A)$ and $PP_1 = inv^+(PP)$

and $AR = \text{while } Wn \ b \ \text{do } IreL(setAI(IP, C)) \ skip \ \text{end} \ (pa, A')$

[W4rPI]

$$\frac{m = previous() \quad A = m:A' \quad \beta(Wn) = def \quad \delta(W) = (m, T, T, PP):W' \quad PP \neq skip}{(S \mid \delta, \beta, \square) \xrightarrow{m} (\text{while } Wn \ T \ \text{do } IreL(PP_1) \ skip \ \text{end} \ (pa, A') \mid \delta[W/W'], \beta[Wn \Rightarrow AR], \square)}$$

where $S = \text{while } Wn \ b \ \text{do } IP \ PP' \ \text{end} \ (pa, A)$ and $PP_1 = inv^+(PP)$

and $AR = \text{while } Wn \ b \ \text{do } IreL(IP) \ skip \ \text{end} \ (pa, A')$

Procedure Call

Consider a procedure call beginning at the end of the execution of the procedure body. The inverse execution should open the inverted call and then stop. Using rule [G3aP], the second program within the `runc` construct is manually set to abort (a situation that can never arise during normal execution and only within the proof). Alongside the saving of all identifiers prior to removal, a true abort flag is recorded. As for loops, consider a partial program beginning within the procedure body. The rule [G3aPI] handles this situation, where the second program of the `runc` construct is the partial program. This partial program is saved alongside the abort flag.

[G3aP]

$$\frac{\mathbf{m} = \text{next}() \quad \mu(\mathbf{Cn}) = \text{OP} \quad \text{AOP} = \text{abort}}{(\text{runc } \mathbf{Cn} \text{ skip } \mathbf{I} \text{ AOP end } \mathbf{A} \mid \delta, \mu, \square) \xrightarrow{m} (\text{skip } \mathbf{m}:\mathbf{A} \mid \delta[(\mathbf{m}, \text{getAI}(\mu(\mathbf{Cn}), \mathbf{T})) \rightarrow \mathbf{Pr}], \mu[\mathbf{Cn}], \square)}$$

[G3aPI]

$$\frac{\mathbf{m} = \text{next}() \quad \mu(\mathbf{Cn}) = \text{def} \quad \text{AOP} \neq \text{abort} \quad \text{PP} \neq \text{skip}}{(\mathbf{S} \mid \delta, \mu, \square) \xrightarrow{m} (\text{skip } \mathbf{m}:\mathbf{A} \mid \delta[(\mathbf{m}, \text{getAI}(\mu(\mathbf{Cn}), \mathbf{T}, \text{PP})) \rightarrow \mathbf{Pr}], \mu[\mathbf{Cn}], \square)}$$

where $\mathbf{S} = \text{runc } \mathbf{Cn} \text{ skip } \mathbf{I} \text{ AOP PP end } \mathbf{A}$

The inverse versions, namely [G1rP] and [G1rPI], are only available provided a true abort flag exists. The first replaces the procedure body with an abort statement stopping the execution as desired. The later replaces the body with a version of the partial program containing the necessary abort statement.

[G1rP]

$$\frac{\mathbf{m} = \text{previous}() \quad \mathbf{A} = \mathbf{m}:\mathbf{A}' \quad \mu(\text{evalP}(\mathbf{n}, \mathbf{pa})) = (\mathbf{n}, \mathbf{IP}) \quad \delta(\mathbf{Pr}) = (\mathbf{m}, \mathbf{C}, \mathbf{T}) : \mathbf{Pr}'}{(\text{call } \mathbf{Cn} \ \mathbf{n} \ (\mathbf{pa}, \mathbf{A}) \mid \delta, \mu, \square) \xrightarrow{m} (\text{runc } \mathbf{Cn} \text{ abort } \mathbf{IP}' \text{ end } \mathbf{A}' \mid \delta[\mathbf{Pr}/\mathbf{Pr}'], \mu[\mathbf{Cn} \Rightarrow (\mathbf{n}, \mathbf{IP}')], \square)}$$

where $\mathbf{IP}' = \text{IreP}(\text{setAI}(\mathbf{IP}, \mathbf{C}), \mathbf{Cn})$

[G1rPI]

$$\frac{\mathbf{m} = \text{previous}() \quad \mathbf{A} = \mathbf{m}:\mathbf{A}' \quad \mu(\text{evalP}(\mathbf{n}, \mathbf{pa})) = (\mathbf{n}, \mathbf{IP}) \quad \delta(\mathbf{Pr}) = (\mathbf{m}, \mathbf{C}, \mathbf{T}, \mathbf{PP}) : \mathbf{Pr}'}{(\text{call } \mathbf{Cn} \ \mathbf{n} \ (\mathbf{pa}, \mathbf{A}) \mid \delta, \mu, \square) \xrightarrow{m} (\text{runc } \mathbf{Cn} \ \mathbf{PP}_1 \ \mathbf{IP}' \text{ skip end } \mathbf{A}' \mid \delta[\mathbf{Pr}/\mathbf{Pr}'], \mu[\mathbf{Cn} \Rightarrow (\mathbf{n}, \mathbf{IP}')], \square)}$$

where $\mathbf{IP}' = \text{IreP}(\text{setAI}(\mathbf{IP}, \mathbf{C}), \mathbf{Cn})$ and $\mathbf{PP}_1 = \text{IreP}(\text{setAI}(\text{inv}^+(\mathbf{PP}), \mathbf{C}), \mathbf{Cn})$

5.4 Conclusion

We have identified three key challenges with proving the correctness of our method of reversibility. Each of these challenges have been overcome, with a solution to each described in detail. In Chapter 6, we state and prove each of the properties that we use to determine correctness.

Chapter 6

Correctness of Annotation and Inversion

Chapter 5 addressed several challenges related to proving our method of reversibility to be correct. Such a proof of correctness is often missing from other research in this field. Our proof concerns two properties. Firstly, the process of saving any reversal information must not alter the behaviour of the original program with respect to the program state, while also populating the auxiliary store (named the *annotation result*). Secondly, execution of the inverted version on the final program state produced via the annotated execution must restore the program state to exactly as it was initially (named the *inversion result*). We begin with some important notation.

6.1 Equivalent Program States: Forward

We define *equivalence* between the program state of an original execution and that of an annotated execution. Two such program states are equivalent provided the meaning of each is the same, regardless of equality. Specifically memory locations used for variables may be different but the values of each may still match.

We first require equivalence between variable states, shown in (Definition 6.1.1). Two variable states, each being a pair (γ, σ) , are equivalent if all variables exist and hold the same value in both, regardless of memory locations used. Example 24 shows two equivalent, but not equal, variable states.

Definition 6.1.1. (Equivalent variable states) Let σ and σ_1 be data stores, γ and γ_1 be variable environments and $\text{dom}(\gamma)$ be the domain of γ . We have (σ, γ) is *equivalent* to (σ_1, γ_1) , written as $(\sigma, \gamma) \approx_S (\sigma_1, \gamma_1)$, if and only if $\text{dom}(\gamma) = \text{dom}(\gamma_1)$ and $\sigma(\gamma(\mathbf{X}, \mathbf{Bn})) = \sigma_1(\gamma_1(\mathbf{X}, \mathbf{Bn}))$ for all $\mathbf{X} \in \text{dom}(\gamma)$ and block names \mathbf{Bn} .

Variable environment γ		Variable environment γ_1	
Var name	Mem location	Var name	Mem location
X	11	X	14
Y	12	Y	18
Z	13	Z	12

Data Store σ		Data Store σ_1	
Mem location	Value	Mem location	Value
11	5	12	4
12	2	14	5
13	4	18	2

(a) First variable state, (γ, σ)
(b) Second variable state, (γ_1, σ_1)

Figure 6.1: Two equivalent, but not equal, variable states

Example 24. (Equivalent variable states) Consider the pair of environments (σ, γ) shown in Figure 6.1(a). Variables X, Y and Z exist bound to memory locations 11, 12 and 13 respectively, such that $X = 5$, $Y = 2$ and $Z = 4$. Now consider the pair of environments (σ_1, γ_1) shown in Figure 6.1(b). As before, each variable X, Y and Z exists, this time bound to different memory locations 14, 18 and 12 respectively. Since each location holds the values 4, 5 and 2 respectively, the variable states (σ, γ) and (σ', γ') are equivalent but not equal. ■

Next we consider both the procedure and while environments. Each such environment from the annotated execution must contain the same mappings as that of the original execution, where each program (loop or procedure body) is the annotated version of the original. Therefore procedure environments (Definition 6.1.2) and while environments (Definition 6.1.3) can be equivalent but not equal. Note that \mathbf{Pn} could be \mathbf{Cn} in Definition 6.1.2.

Definition 6.1.2. (Equivalent procedure environments) Let μ be a procedure environment, and μ_1 be an annotated procedure environment. We have that μ is *equivalent* to μ_1 , written as $\mu \approx_P \mu_1$, if and only if $\text{dom}(\mu) = \text{dom}(\mu_1)$, $\mu(\mathbf{Pn}) = (\mathbf{n}, \mathbf{P})$, $\mu_1(\mathbf{Pn}) = (\mathbf{n}, \mathbf{AP})$ and $\mathbf{AP} = \text{ann}(\mathbf{P})$ for all $\mathbf{Pn} \in \text{dom}(\mu)$.

Definition 6.1.3. (Equivalent while environments) Let β be a while environment, and β_1 be an annotated while environment. We have that β is *equivalent* to β_1 , written as $\beta \approx_W \beta_1$, if and only if $\text{dom}(\beta) = \text{dom}(\beta_1)$, $\beta(\mathbf{Wn}) = \mathbf{P}$, $\beta_1(\mathbf{Wn}) = \mathbf{AP}$ and $\mathbf{AP} = \text{ann}(\mathbf{P})$ for all $\mathbf{Wn} \in \text{dom}(\beta)$.

Finally we are ready to state the definition of equivalence between program states (Definition 6.1.4). Note that the auxiliary store δ is ignored, as this may or may not be different within an original and an annotated execution.

Definition 6.1.4. (Equivalent program states) Let \square be the tuple of environments $(\sigma, \gamma, \mu, \beta, \delta)$ and \square_1 represent the tuple of annotated environments $(\sigma_1, \gamma_1, \mu_1, \beta_1, \delta_1)$. We have that \square is *equivalent* to \square_1 , written $\square \approx \square_1$, if and only if $(\sigma, \gamma) \approx_S (\sigma_1, \gamma_1)$, $\mu \approx_P \mu_1$ and $\beta \approx_W \beta_1$.

In Section 6.4, we introduce a definition of equivalence between the states of an annotated and inverted execution, where the auxiliary stores must to be equal.

6.2 Annotation Result

Our first result compares the execution of an original and the corresponding annotated program. The only difference should be that the annotated execution may also populate the auxiliary store. Theorem 7 states the annotation result.

Theorem 7. (Annotation result) Let P be an original program and AP be the corresponding annotated version $ann(P)$. Further let \square be the tuple $(\sigma, \gamma, \mu, \beta)$ of initial program state environments, and δ be the initial auxiliary store.

If an execution $(P \mid \delta, \square) \hookrightarrow^* (\text{skip} \mid \delta, \square')$ exists, for some program state \square' , then there exists an annotated execution $(AP \mid \delta, \square_1) \xrightarrow{\circ}^* (\text{skip } I \mid \delta', \square'_1)$ for some I, \square_1, \square'_1 such that $\square_1 \approx \square$ and $\square'_1 \approx \square'$, and some auxiliary store δ' .

Proof. This is via induction on the length of the execution of an original program P , namely $(P \mid \square) \hookrightarrow^* (\text{skip} \mid \square')$, and the corresponding execution of the annotated version AP , namely $(AP \mid \square) \xrightarrow{\circ}^* (\text{skip} \mid \square')$. Recall that each rule $[R]$ within the traditional semantics has a corresponding rule $[Ra]$ within the forward semantics (Chapter 4). We first consider all base cases (executions with transition length 1), namely original executions via the rules $[P3]$, $[D1]$ and $[W1]$. Each of the corresponding annotated rules, namely $[P3a]$, $[D1a]$ and $[W1a]$, is shown to behave identically with respect to the program state while potentially modifying the auxiliary store. Therefore each base case holds.

We now briefly describe the use of induction. We first assume that Theorem 7 holds for all program executions of length k such that $k \geq 1$, namely $(Q \mid \square_1) \hookrightarrow^* (\text{skip} \mid \square'_1)$. Then we assume our original program has execution length $k + 1$, namely $(T \mid \square_2) \hookrightarrow^* (\text{skip} \mid \square'_2)$. This assumed execution can then be rewritten to state the first step via transition rule $[R]$ as $(T \mid \square_2) \hookrightarrow (T' \mid \square''_2) \hookrightarrow^* (\text{skip} \mid \square'_2)$.

Considering each possible rule $[R]$ from Chapter 4, we can then see that there exists a corresponding transition rule of the annotated execution that behaves identically. In some cases, for example $[S1a]$, $[P1a]$ and $[I2a]$, there is no difference other

than the presence of the auxiliary store (which is not used). All other rules, including [D1a], [I4a] and [H1a], behave identically w.r.t the program state while differing on the use of the auxiliary store (as required). Finally the induction hypothesis can be applied to the remaining, shorter execution (namely $(T' \mid \Box_2'') \hookrightarrow^* (\text{skip} \mid \Box_2')$) to complete our proof. Therefore Theorem 7 is valid. \square

6.3 Equivalent Program States: Reversal

Prior to stating and proving our inversion result, we first require modifications to the definition of equivalence from Section 6.1 to allow comparison of program states from a forward and a reverse execution. Specifically, a pair of procedure or while environments are equivalent as before, but with all programs from the inverse environment being the inverted version of the corresponding program from the forward environment. Definition 6.3.1 and Definition 6.3.2 state this updated notion of equivalence. The equivalence of variable states is unchanged from Definition 6.1.1.

Definition 6.3.1. (Equivalent while environments) Let β be a while environment, and β_1 be an annotated while environment. We have that β is *equivalent* to β_1 , written as $\beta \approx_W \beta_1$, if and only if $\text{dom}(\beta) = \text{dom}(\beta_1)$, $\beta(\text{Wn}) = \text{AP}$, $\beta_1(\text{Wn}) = \text{IP}$ and $\text{IP} = \text{inv}(\text{AP})$ for all $\text{Wn} \in \text{dom}(\beta)$.

Definition 6.3.2. (Equivalent procedure environments) Let μ be a procedure environment, and μ_1 be an annotated procedure environment. We have that μ is *equivalent* to μ_1 , written as $\mu \approx_P \mu_1$, if and only if $\text{dom}(\mu) = \text{dom}(\mu_1)$, $\mu(\text{Pn}) = (\text{n}, \text{AP})$, $\mu_1(\text{Pn}) = (\text{n}, \text{IP})$ and $\text{IP} = \text{inv}(\text{AP})$ for all $\text{Pn} \in \text{dom}(\mu)$.

Equivalence between program states is now updated to include the auxiliary store δ , as the reversal of each statement should use all reversal information saved for it, hence garbage-free reversibility. Definition 6.1.4 states this formally.

Definition 6.3.3. (Equivalence of program states) Let \Box be the tuple of environments $(\sigma, \gamma, \mu, \beta, \delta)$ and \Box_1 represent the tuple of annotated environments $(\sigma_1, \gamma_1, \mu_1, \beta_1, \delta_1)$. We have that \Box is *equivalent* to \Box_1 , written $\Box \approx \Box_1$, if and only if $(\sigma, \gamma) \approx_S (\sigma_1, \gamma_1)$, $\mu \approx_P \mu_1$, $\beta \approx_W \beta_1$ and $\delta = \delta_1$.

By abuse of notation, any following use of \approx refers to this updated definition.

6.4 Inversion Result

We are now ready to state our inversion result. This claims that given the final program state produced by the matching annotated execution, the reverse execution

restores this to a state equivalent to that of prior to the forward execution. This includes the restoration of the auxiliary store, meaning our approach to reversibility is garbage-free. This is Theorem 8 below.

Theorem 8. Let P be an original program, AP be the corresponding annotated version $ann(P)$, and AP' be the executed version of AP . Further let \square be the tuple $(\sigma, \gamma, \mu, \beta, \delta)$ of all initial state environments.

If $(P \mid \square) \hookrightarrow^* (\mathbf{skip} \mid \square')$ for some program state \square' , and therefore by Theorem 7 the annotated execution $(AP \mid \square_1) \xrightarrow{\circ}^* (\mathbf{skip} \mid \square'_1)$ exists for some I and program state \square'_1 such that $\square'_1 \approx \square'$, then there exists a corresponding inverse execution $(inv(AP') \mid \square'_2) \xrightarrow{\circ}^* (\mathbf{skip} \mid \square_2)$, for some program states \square'_2, \square_2 , such that $\square'_2 \approx \square'_1$ and $\square_2 \approx \square_1$. Provided this holds, Definition 6.3.3 states that $\delta_2 = \delta_1$, showing the reversal to be garbage free.

With the three challenges associated with our inversion result addressed in Chapter 5, and with our notion of equivalence updated for reverse executions in Section 6.3, we are now ready to prove Theorem 8. We achieve this using the approach illustrated in Figure 6.2, where proving each of the three arrows (labelled \Rightarrow_i) valid gives us Theorem 8. We first note that \Rightarrow_1 represents the restriction of a standard execution into an equivalent uniform execution, and recall the validation of this via Lemma 4 in Chapter 5.

We focus on the arrow \Rightarrow_2 from Figure 6.2. This represents a stronger version of Theorem 8, where all executions are uniform and the annotated program may be either complete or partially executed. The inverted version is now generated using the function inv^+ and its execution may reach either skip or a skip equivalent. Proposition 9 states this stronger result.

Proposition 9. Let P be an original program, AP be the annotated program $ann(P)$, and AP' be the executed version of AP . Further let \square be the tuple $(\sigma, \gamma, \mu, \beta, \delta)$ of all initial program state environments.

If $(P \mid \square) \hookrightarrow^* (\mathbf{skip} \mid \square')$ exists for some program state \square' , and therefore by Theorem 7 the annotated execution $(AP \mid \square_1) \xrightarrow{\circ}^* (\mathbf{skip} \mid \square'_1)$ for some I and program state \square'_1 such that $\square'_1 \approx \square'$, then there exists a corresponding inverse execution $(inv^+(AP') \mid \square'_2) \xrightarrow{\circ}^* (AQ \mid \square_2)$, for the program AQ that is $\mathbf{skip} \mid I'$ if AP was complete and a skip equivalent otherwise, and program states \square'_2, \square_2 , such that $\square'_2 \approx \square'_1$ and $\square_2 \approx \square_1$.

The proof of this proposition uses two smaller results. The first, named the *Statement Property* and shown in Lemma 10, considers all statement executions that begin with an identifier step. The second, named the *Program Property* and shown in Lemma 11, is more general and considers all program executions that begin with either an identifier or a skip step.

$$\begin{array}{ccc}
(P \mid \square) \xrightarrow{\circ}^* (\text{skip } I \mid \square') & \Rightarrow 1 & (P \mid \square) \xrightarrow{U}^* (\text{skip } I \mid \square') \\
? \downarrow & & \Downarrow 2 \\
(inv^+(P) \mid \square'_1) \xrightarrow{\circ}^* (Q \mid \square_1) & \Leftarrow 3 & (inv^+(P) \mid \square'_1) \xrightarrow{U}^* (Q \mid \square_1)
\end{array}$$

Figure 6.2: Diagram representation of proof outline

$P ::= S \mid \text{skip}; \text{AP} \mid P \text{ par } P$
 $S ::= \text{skip} \mid \text{begin } B_n \text{ } P \text{ AP end}$

Figure 6.3: Restricted syntax of complete programs beginning with skip steps

Lemma 10 (Statement Property). Let AS be a complete or partially executed annotated statement and \square be the tuple $(\sigma, \gamma, \mu, \beta, \delta)$ of initial program state environments.

If a uniform execution $(AS \mid \square) \xrightarrow{U}^m (AS' \mid \square'') \xrightarrow{U}^*_s (AS'' \mid \square'') \xrightarrow{U}^{\circ}^* (\text{skip } I \mid \square')$ exists for some statements AS' , AS'' and program states \square'' , \square' , then there also exists a uniform inverse execution $(inv^+(AS) \mid \square'_1) \xrightarrow{U}^{\circ}^* (AT'' \mid \square''_1) \xrightarrow{U}^m (AT' \mid \square_1) \xrightarrow{U}^*_s (AT \mid \square_1)$ for some statements AT'' , AT' , AT such that $AT = \text{skip } I$ for some I if AS is a complete statement and AT is a skip equivalent if AS is partially executed, and program states \square'_1 , \square''_1 , \square_1 such that $\square'_1 \approx \square'$, $\square''_1 \approx \square''$ and $\square_1 \approx \square$.

Proof. The proof of Lemma 10 (Statement Property) is deferred to Section 6.5. \square

We remark that it is sufficient in Lemma 10 to consider only statement executions that begin with identifier steps. The Program Property must however consider executions that begin with skip steps (Part 1) and those that begin with an identifier step (Part 2). Respecting uniformity allows us to only consider complete programs that begin with skip steps, namely those of the restricted syntax shown in Figure 6.3. P and S are used within this syntax to represent programs and statements beginning with skip steps respectively, while by abuse of notation, AP represents an annotated program as introduced in Chapter 4.

Lemma 11 (Program Property). Let AP be a complete program (syntax from Figure 6.3) in Part 1, and either a complete or partially executed annotated program in Part 2. Further let \square be the tuple $(\sigma, \gamma, \mu, \beta, \delta)$ of initial program environments.

Part 1. If a uniform forward execution $(AP \mid \square) \xrightarrow{U}^*_s (AP' \mid \square'') \xrightarrow{U}^{\circ}^* (\text{skip } I \mid \square')$ exists for some program AP' and program states \square'' , \square' such that $\square'' = \square$ (by Lemma 1 of Chapter 3), then there exists a uniform inverse execution $(inv^+(AP) \mid \square'_1) \xrightarrow{U}^{\circ}^* (AQ' \mid \square''_1) \xrightarrow{U}^*_s (AQ \mid \square_1)$ for some programs AQ' ,

AQ such that AQ is skip if AP is a complete program and a skip equivalent otherwise, and program states $\square'_1, \square''_1, \square_1$ such that $\square'_1 \approx \square', \square''_1 \approx \square''$ and $\square_1 \approx \square$.

Part 2. If a uniform forward execution $(\text{AP} \mid \square) \xrightarrow{U}^m (\text{AP}' \mid \square'') \xrightarrow{U}^* (\text{AP}'' \mid \square'') \xrightarrow{U}^* (\text{skip } I \mid \square')$ exists for some program AP' and program states \square'' and \square' , then there exists a uniform inverse execution $(\text{inv}^+(\text{AP}) \mid \square'_1) \xrightarrow{U}^* (\text{AQ}'' \mid \square''_1) \xrightarrow{U}^m (\text{AQ}' \mid \square_1) \xrightarrow{U}^* (\text{AQ} \mid \square_1)$ for some programs AQ'', AQ', AQ is skip if AP is a complete program and a skip equivalent otherwise, and states $\square'_1, \square''_1, \square_1$ such that $\square'_1 \approx \square', \square''_1 \approx \square''$ and $\square_1 \approx \square$.

Proof. The proof of Lemma 11 (Program Property) is deferred to Section 6.6. \square

With our two smaller results introduced and proved to be valid, we are ready to state the proof of Proposition 9.

Proof. This proof is by induction on the length of a uniform execution, namely $(\text{AP} \mid \square) \xrightarrow{U}^* (\text{skip } I \mid \square')$, for some program AP and states \square and \square' . It can be split into two parts. The first part considers executions that begin with skip steps, while the second considers executions that begin with identifier steps. Each part is now considered in turn.

Executions beginning with skip steps This is done as in the proof of Part 1 of Lemma 11. We consider only skip steps that can begin a complete program execution (programs of the syntax in Figure 6.3). The base cases (already shown in the proof of Part 1 of Lemma 11) are Case 10.1 (sequential skips), Case 10.2 (empty parallel) and Case 10.3 (empty block). The inductive cases are those that correspond to Case 10.5 (sequential composition) and Case 10.6 (parallel composition) of the proof of Part 1 of Lemma 11. With all shown to be valid in Section 6.6, this part holds.

Executions beginning with identifier steps This is done as in the proof of Part 2 of Lemma 11, where only those identifier steps that can be the beginning of a complete execution are considered. The base cases (already used in the proof of Part 2 of Lemma 11 and shown in the proof of Lemma 10) are Case 9.1 (assignment) and Case 9.2 (loop with zero iterations). The inductive cases are those corresponding to Case 10.8 (sequential composition) and Case 10.9 (parallel composition) from the proof of Part 2 of Lemma 11. Since the Program Property is proved via mutual induction with the Statement Property, we also consider several cases (of first steps) from the proof of Lemma 10. This does not need any further base cases (since the two such cases are already used above), but does require the inductive cases corresponding to Case 9.3 and Case 9.4 (conditional statement), Case 9.10 (while

loop) and Case 9.13 (block) from the proof of Lemma 10. With all holding, this part is therefore valid.

With all possible executions having been considered and shown to hold, we can now conclude Proposition 9 is also valid. \square

With Proposition 9 shown above to be correct, the arrow \Rightarrow_2 from Figure 6.2 is therefore also correct. We finally consider the arrow \Rightarrow_3 from Figure 6.2. As previously noted, all uniform executions are also standard executions by definition. This means \Rightarrow_3 exists in all cases as required. For completeness, the trivial result shown in Lemma 6 of Chapter 5 proves the existence of this arrow. We have therefore completed the proof outline shown in Figure 6.2 meaning Theorem 8 is valid, as required.

6.5 Proof of Statement Property

We now give the proof of Lemma 10 (Statement Property). By abuse of notation, we use **AS** to represent either a complete or partially executed statement. In cases considering full statement execution (e.g. Case 9.1), we ignore the abort statement introduced by inv^+ as the program stops as desired.

Lemma 10 (Statement Property). Let **AS** be a complete or partially executed annotated statement and \square be the tuple $(\sigma, \gamma, \mu, \beta, \delta)$ of initial program state environments.

If a uniform execution $(\mathbf{AS} \mid \square) \xrightarrow{m}_U (\mathbf{AS}' \mid \square'') \xrightarrow{*}_U (\mathbf{AS}'' \mid \square'') \xrightarrow{*}_U (\mathbf{skip} \ I \mid \square')$ exists for some statements \mathbf{AS}' , \mathbf{AS}'' and program states \square'' , \square' , then there also exists a uniform inverse execution $(inv^+(\mathbf{AS}) \mid \square'_1) \xrightarrow{*}_U (\mathbf{AT}'' \mid \square'_1) \xrightarrow{m}_U (\mathbf{AT}' \mid \square_1) \xrightarrow{*}_U (\mathbf{AT} \mid \square_1)$ for some statements \mathbf{AT}'' , \mathbf{AT}' , \mathbf{AT} such that $\mathbf{AT} = \mathbf{skip} \ I$ for some I if \mathbf{AS} is a complete statement and \mathbf{AT} is a skip equivalent if \mathbf{AS} is partially executed, and program states \square'_1 , \square''_1 , \square_1 such that $\square'_1 \approx \square'$, $\square''_1 \approx \square''$ and $\square_1 \approx \square$.

All executions from here are uniform and so we omit the specific notation. E.g. a uniform execution $(\mathbf{P} \mid \square) \xrightarrow{m}_U (\mathbf{P}' \mid \square')$ is now written as $(\mathbf{P} \mid \square) \xrightarrow{m} (\mathbf{P}' \mid \square')$.

Proof. This proof is by mutual induction of the Statement Property (this lemma) and the Program Property (Lemma 11), on the length of the executions $(\mathbf{AS} \mid \square) \xrightarrow{*}_U (\mathbf{skip} \ I \mid \square')$ and $(\mathbf{AP} \mid \square) \xrightarrow{*}_U (\mathbf{skip} \ I \mid \square')$ respectively.

With no executions of length 0, our base cases are any executions of length 1. The two base cases are a single assignment statement (Case 9.1) and a single while loop statement that performs zero iterations (Case 9.2).

Case 9.1. (Assignment [D1a]) Consider a single assignment of the value of the expression e to the variable X . Let $\mathbf{AS} = X = e$ (pa, A) and \square be the tuple of initial

program state environments $(\sigma, \gamma, \mu, \beta, \delta)$. Assume an execution exists with the first transition via [D1a] such that

$$(AS \mid \square) = (X = v \text{ (pa,A)} \mid \square) \xrightarrow{m} (\text{skip (pa,m:A)} \mid \square')$$

for identifier m and program state \square' such that $\square' = \square[\sigma[1 \mapsto v], \delta[(m, v1) \rightarrow X]]$, where 1 is the memory location for the variable X , v is the result of evaluating the expression e (the new value) and $v1$ is the current value of X (the old value). Application of the rule [D1a] means the premises were valid at that point, namely that $m = \text{next}()$, $(e \text{ pa} \mid \square) \hookrightarrow_a^* (v \mid \square)$ (the expression evaluates to v) and $\text{evalV}(X, \text{pa}, \gamma) = 1$ (the variable is bound to memory location 1).

We need to show that there exists an execution

$$(\text{inv}^+(AS) \mid \square'_1) \overset{\circ}{\rightsquigarrow}^* (AT'' \mid \square''_1) \xrightarrow{m} (AT' \mid \square_1) \rightsquigarrow_s^* (AT \mid \square_1)$$

for some statements AT'' , AT' , AT , and program states \square'_1 , \square''_1 , \square_1 such that $\square'_1 \approx \square'$ and $\square_1 \approx \square$. Firstly, we have that $\text{inv}^+(AS) = X = e \text{ (pa,m:A)}$. Beginning in the program state \square'_1 , such that $\square'_1 \approx \square'$, we note that no further execution was performed forwards and so does not require inversion. Therefore $AT'' = X = e \text{ (pa,m:A)}$, and $\square'_1 = \square''_1$, giving the execution $(\text{inv}^+(AS) \mid \square'_1) \overset{\circ}{\rightsquigarrow}^* (AT'' \mid \square''_1)$ with length 0.

From the program state \square'_1 , we have that $m = \text{previous}()$ and that the identifier stack for this assignment is equal to $m:A$ (m as its head). Also, $\delta'_1(X) = (m, v1):X'$ (the stack X contains the pair $(m, v1)$ as its head) and $\text{evalV}(X, \text{pa}, \gamma) = 1$ (the variable is bound to the memory location 1 or equivalent). With all premises of the rule [D1r] valid, and that no other rule is applicable, we have the execution

$$(\text{inv}^+(AS) \mid \square'_1) = (X = e \text{ (pa,m:A)} \mid \square'_1) \xrightarrow{m} (\text{skip (pa,A)} \mid \square_1)$$

for state \square_1 such that $\square_1 = \square'_1[\sigma'_1[1 \mapsto v1]]$ and $\delta'_1[X/X']$, meaning $\square_1 \approx \square$ (as required). This execution has reached skip and so no further execution is available. Therefore we take $AT' = \text{skip (pa,A)}$ and $AT = AT'$. With program states matching, and AT being skip, we have shown our desired inverse execution and therefore this case to hold.

Case 9.2. (Loop [W1a]) Consider a while loop with zero iterations (loop condition immediately evaluates to false). Let $AS = \text{while Wn b do AP end (pa,A)}$ and \square be the tuple of initial program state environments $(\sigma, \gamma, \mu, \beta, \delta)$. Assume an execution exists with the first transition via [W1a]

$$(AS \mid \square) = (\text{while Wn b do AP end (pa,A)} \mid \square) \xrightarrow{m} (\text{skip (pa,m:A)} \mid \square')$$

for program state \square' such that $\square' = \square[\delta[(\mathbf{m}, \mathbf{F}, \mathbf{F}) \rightarrow \mathbf{W}]]$ (noting the false abort flag that is of no consequence here). Application of the rule [W1a] means all premises were valid at this point, namely that $\mathbf{m} = \text{next}()$, $\beta(\mathbf{Wn}) = \text{und}$ (there is no mapping for this loop within β since it has not started) and $(\mathbf{b} \text{ pa} \mid \square) \hookrightarrow_{\mathbf{b}}^* (\mathbf{F} \mid \square)$ (the condition evaluates to \mathbf{F}).

We need to show that there exists an execution

$$(\text{inv}^+(\mathbf{AS}) \mid \square'_1) \overset{\circ}{\rightsquigarrow}^* (\mathbf{AT}'' \mid \square''_1) \overset{m}{\rightsquigarrow} (\mathbf{AT}' \mid \square_1) \rightsquigarrow_s^* (\mathbf{AT} \mid \square_1)$$

for some statements \mathbf{AT}'' , \mathbf{AT}' , \mathbf{AT} , and program states \square'_1 , \square''_1 , \square_1 such that $\square'_1 \approx \square'$ and $\square_1 \approx \square$. We have that $\text{inv}^+(\mathbf{AS}) = \text{while } \mathbf{Wn} \text{ b do } \text{inv}(\mathbf{AP}) \text{ end } (\mathbf{pa}, \mathbf{m}:\mathbf{A})$ (note can ignore the abort in this case). Starting in the program state \square'_1 such that $\square'_1 \approx \square'$, the forward execution above does not contain any further execution and so no use of the induction hypothesis is required. We therefore take $\mathbf{AT}'' = \text{while } \mathbf{Wn} \text{ b do } \text{inv}(\mathbf{AP}) \text{ end } (\mathbf{pa}, \mathbf{m}:\mathbf{A})$, and $\square'_1 = \square''_1$, giving the execution $(\text{inv}^+(\mathbf{AS}) \mid \square'_1) \overset{\circ}{\rightsquigarrow}^* (\mathbf{AT}'' \mid \square''_1)$ with length 0.

From the program state \square'_1 , we have that $\mathbf{m} = \text{previous}()$ (since no further identifiers have been used) and that the identifier stack for this loop is equal to $\mathbf{m}:\mathbf{A}$ (\mathbf{m} as its head). We also have that $\beta'_1(\mathbf{Wn}) = \text{und}$ (since the while loop has not started) and $\delta'_1(\mathbf{W}) = (\mathbf{m}, \mathbf{F}, \mathbf{F}) : \mathbf{W}'$ (the stack \mathbf{W} has the triple $(\mathbf{m}, \mathbf{F}, \mathbf{F})$ as its head). This shows that all premises of [W1r] are valid, and that no other rule is applicable. This gives the execution

$$(\text{while } \mathbf{Wn} \text{ b do } \text{inv}(\mathbf{AP}) \text{ end } (\mathbf{pa}, \mathbf{m}:\mathbf{A}) \mid \square''_1) \overset{m}{\rightsquigarrow} (\text{skip } (\mathbf{pa}, \mathbf{A}) \mid \square_1)$$

for program state \square_1 such that $\square_1 = \square''_1[\delta''_1[\mathbf{W}/\mathbf{W}']]$ (with $\delta''_1(\mathbf{W}) = (\mathbf{m}, \mathbf{F}, \mathbf{F}) : \mathbf{W}'$), resulting in $\square_1 = \square$ as required. This execution has reached skip and so no further execution is available. Therefore we take $\mathbf{AT}' = \text{skip } (\mathbf{pa}, \mathbf{A})$ and $\mathbf{AT} = \mathbf{AT}'$. With \mathbf{AT} having reached skip and program states that match at each required position, we have our desired inverse execution meaning this case holds.

With Lemma 10 valid for all base cases, it is valid for all executions of length 1. We now consider inductive cases. Assume Lemma 10 holds for all statements \mathbf{AR} and program states \square^* such that the execution is of length k (where $k \geq 1$), namely $(\mathbf{AR} \mid \square) \overset{m}{\rightsquigarrow} (\mathbf{AR}' \mid \square'') \rightarrow_s^* (\mathbf{AR}'' \mid \square'') \overset{\circ}{\rightsquigarrow}^* (\text{skip } \mathbf{I} \mid \square')$ (induction hypothesis). Now assume that the execution $(\mathbf{AS} \mid \square) \overset{m}{\rightsquigarrow} (\mathbf{AS}' \mid \square'') \rightarrow_s^* (\mathbf{AS}'' \mid \square'') \overset{\circ}{\rightsquigarrow}^* (\text{skip } \mathbf{I} \mid \square')$ has length l such that $l > k$.

Inductive cases considered are for conditional statements (Case 9.3–9.8), loops (Cases 9.9–9.12), blocks (Case 9.13–9.17) and call statements (Cases 9.18–9.20).

Conditional Statements

Case 9.3. (Conditional [I1aT]) Consider the opening of a conditional statement. Let $AS = \text{if In } b \text{ then } AP \text{ else } AQ \text{ end } (pa, A)$ and the initial program state $\square = (\sigma, \gamma, \mu, \beta, \delta)$. Assume the following execution exists where the first transition is via [I1aT]

$$\begin{aligned} (AS \mid \square) &= (\text{if In } b \text{ then } AP \text{ else } AQ \text{ end } (pa, A) \mid \square) \\ &\xrightarrow{m} (\text{if In } T \text{ then } AP \text{ else } AQ \text{ end } (pa, m:A) \mid \square'') \\ &\rightarrow_s^* (\text{if In } T \text{ then } AP' \text{ else } AQ \text{ end } (pa, m:A) \mid \square'') \\ &\xrightarrow{\circ}^* (\text{skip } (pa, A') \mid \square') \end{aligned}$$

for some program AP' and program states \square'', \square' such that $\square'' = \square$ (as [I1aT] does not change the program state), and an identifier stack A' that ends with the sub-stack $(m:A)$. Application of [I1aT] means that all premises were valid at the time, namely $m = \text{next}()$ and $(b \text{ pa} \mid \square) \hookrightarrow_b^* (T \mid \square)$. Since this transition opens a conditional, AS must be a complete statement and therefore AP must also be a complete program. As such, it is possible for AP to begin with skip steps meaning the execution

$$\begin{aligned} &(\text{if In } T \text{ then } AP \text{ else } AQ \text{ end } (pa, m:A) \mid \square'') \\ &\rightarrow_s^* (\text{if In } T \text{ then } AP' \text{ else } AQ \text{ end } (pa, m:A) \mid \square'') \end{aligned}$$

exists for some AP' . This group of skip steps are said to have been caused by the identifier rule using m . The inverted version of this group will be contained within the uniform inverse execution of the remaining forward program (corresponding inverse skip steps will be caused by the previous inverse identifier rule). From here, we use AS' to represent $\text{if In } T \text{ then } AP' \text{ else } AQ \text{ end } (pa, m:A)$.

We need to show that there exists an execution

$$(inv^+(AS) \mid \square'_1) \xrightarrow{\circ}^* (AT'' \mid \square''_1) \xrightarrow{m} (AT' \mid \square_1) \rightsquigarrow_s^* (AT \mid \square_1)$$

for some statements AT'', AT', AT , and program states $\square'_1, \square''_1, \square_1$ such that $\square'_1 \approx \square'$, $\square''_1 \approx \square''$ and $\square_1 \approx \square$. From 5.24 of Figure 5.7 in Chapter 5, we have that $inv^+(AS) = \text{if In } b \text{ then } inv(AP) \text{ else } inv(AQ) \text{ end } (pa, A')$ (where the sequentially composed abort statement can be ignored here as there is no further execution and will stop at the required position).

From our assumed execution above, we note that the execution

$$\begin{aligned} (AS' \mid \square'') &= (\text{if In } T \text{ then } AP' \text{ else } AQ \text{ end } (pa, m:A) \mid \square'') \\ &\xrightarrow{\circ}^* (\text{skip } (pa, A') \mid \square') \end{aligned}$$

is a uniform execution and must begin with an identifier step. Since AS' must be a partially executed program, from 5.25 in Figure 5.7 of Chapter 5, we have that $inv^+(AS') = \text{if In } b \text{ then } inv^+(AP) \text{ } inv(AP) \text{ else } inv(AQ) \text{ } inv(AQ) \text{ end } (pa, m:A)$, where an abort is added to the end of the inverted true branch in order to stop the execution at the required position (after entire execution of the true branch but before the closure of the conditional). The induction hypothesis of the Statement Property (Lemma 10) applied to this shorter execution gives us

$$\begin{aligned} (inv^+(AS') \mid \square'_1) &= (\text{if In } b \text{ then } inv^+(AP) \text{ } inv(AP) \\ &\quad \text{else } inv(AQ) \text{ } inv(AQ) \text{ end } (pa, m:A) \mid \square'_1) \\ &\rightsquigarrow^* (AR' \mid \square''_1) \end{aligned}$$

by a sequence of rule applications SR, for a skip equivalent AR' and states \square'_1, \square''_1 such that $\square'_1 \approx \square'$ and $\square''_1 \approx \square''$. Since AP must be a full program, $inv^+(AP)$ sequentially composes an abort statement to the end. As a result, the induction hypothesis use above performs the entire inverted true branch until the abort is reached. Therefore $AR' = \text{if In } T \text{ then abort } inv(AP) \text{ else } inv(AQ) \text{ } inv(AQ) \text{ end } (pa, m:A)$.

Now compare the statements $inv^+(AS)$ and $inv^+(AS')$. They are the same with the exception that $inv^+(AS')$ contains an abort statement at the end of the true branch. Since both will begin with the same execution (inversion of the rest of the forward program after the identifier plus skip steps), then by the same sequence of rules SR we obtain

$$\begin{aligned} (inv^+(AS) \mid \square'_1) &= (\text{if In } b \text{ then } inv^+(AP) \text{ } inv(AP) \\ &\quad \text{else } inv(AQ) \text{ } inv(AQ) \text{ end } (pa, A') \mid \square'_1) \\ &\rightsquigarrow^* (AR \mid \square''_1) \end{aligned}$$

for program AR . The same sequence of rules mean that this transition sequence will perform the execution of the entire true branch, reaching the point at which the execution of $inv^+(AS')$ hit the abort statement (minus the trivial [S1a] rule that has no effect on the program state). Therefore $AR = \text{if In } T \text{ then skip } I' \text{ } inv(AP) \text{ else } inv(AQ) \text{ } inv(AQ) \text{ end } (pa, m:A)$ for some I' . With the program states matching to our desired execution, we can take $AT'' = AR$.

From program state \square''_1 (recall $\square''_1 \approx \square''$), we have that $m = \text{previous}()$ (since no further identifiers have been used) and the identifier stack for this statement is equal to $m:A$. This shows all premises of the rule [I4r] to be valid, and that no other rule is applicable. Using [I4r], we have

$$\begin{aligned} (AT'' \mid \square''_1) &= (\text{if In } T \text{ then skip } I' \text{ } inv(AP) \text{ else } inv(AQ) \text{ } inv(AQ) \\ &\quad \text{end } (pa, m:A) \mid \square''_1) \rightsquigarrow^m (\text{skip } (pa, A) \mid \square_1) \end{aligned}$$

for program state \square_1 such that $\square_1 = \square'_1$ (since this rule does not change the program state), meaning $\square_1 \approx \square$ as required. With the program states matching, we can take $AT' = \text{skip } (\text{pa}, A)$. With no skip steps to apply (already reached skip), we can take $AT = AT'$. Since AT is skip, we can conclude this case holds.

Case 9.4. (Conditional [I1aF]) Consider the opening of a conditional statement where the condition evaluates to false. This follows Case 9.3 using rules [I1aF] and [I5a] in place of [I1aT] and [I4a] respectively.

Case 9.5. (Conditional [I2a]) Consider an identifier step from within the true branch of a conditional statement. Let $AS = \text{if In T b then AP AOP else AQ AQ end } (\text{pa}, A)$ and the initial program state $\square = (\sigma, \gamma, \mu, \beta, \delta)$. Assume the following execution exists with the first transition via the rule [I2a]

$$\begin{aligned} (AS \mid \square) &= (\text{if In T b then AP AOP else AQ AQ end } (\text{pa}, A) \mid \square) \\ &\xrightarrow{m} (\text{if In T b then AP'' AOP else AQ AQ end } (\text{pa}, A) \mid \square'') \\ &\rightarrow_s^* (\text{if In T b then AP' AOP else AQ AQ end } (\text{pa}, A) \mid \square'') \\ &\xrightarrow{o}^* (\text{skip } (\text{pa}, A') \mid \square') \end{aligned}$$

for programs AP'' , AP' , and program states \square'' and \square' . We note that this assumed execution can be rewritten to highlight the point at which the true branch finishes, specifically

$$\begin{aligned} (AS \mid \square) &= (\text{if In T b then AP AOP else AQ AQ end } (\text{pa}, A) \mid \square) \\ &\xrightarrow{m} (\text{if In T b then AP'' AOP else AQ AQ end } (\text{pa}, A) \mid \square'') \\ &\rightarrow_s^* (\text{if In T b then AP' AOP else AQ AQ end } (\text{pa}, A) \mid \square'') \\ &\xrightarrow{o}^* (\text{if In T b then skip } I_1 \text{ AOP else AQ AQ end } (\text{pa}, A) \mid \square''') \\ &\xrightarrow{n} (\text{skip } (\text{pa}, n:A) \mid \square') \end{aligned}$$

for some program state \square''' . From here, let $AS' = \text{if In T b then skip } I_1 \text{ AOP else AQ AQ end } (\text{pa}, A)$.

We need to show that there exists an execution

$$(inv^+(AS) \mid \square'_1) \xrightarrow{o}^* (AT'' \mid \square''_1) \xrightarrow{m} (AT' \mid \square_1) \rightsquigarrow_s^* (AT \mid \square_1)$$

for some statements AT'' , AT' , AT , and program states \square'_1 , \square''_1 , \square_1 such that $\square'_1 \approx \square'$, $\square''_1 \approx \square''$ and $\square_1 \approx \square$. From 5.25 in Figure 5.7 of Chapter 5, we have that $inv^+(AS) = \text{if In b b then } inv^+(AP) \text{ } inv(AOP) \text{ else } inv(AQ) \text{ } inv(AQ) \text{ end } (\text{pa}, n:A)$, with abort inserted to stop the inverse execution at the end of the inverted true branch.

From our rewritten assumed execution,

$$\begin{aligned} (AS' \mid \square''') &= (\text{if In T b then skip } I_1 \text{ AOP else AQ AQ end } (pa, A) \mid \square''') \\ &\xrightarrow{n} (\text{skip } (pa, n:A) \mid \square') \end{aligned}$$

must be the closing of the conditional, via the rule [I4a]. There can only be one matching inverted execution step, namely the opening of the inverted conditional statement via the rule [I1rT] (the very first step of the inverse execution). Therefore we have the execution

$$\begin{aligned} (inv^+(AS) \mid \square'_1) &= (\text{if In b b then } inv^+(AP) \text{ } inv(AOP) \\ &\quad \text{else } inv(AQ) \text{ } inv(AQ) \text{ end } (pa, n:A) \mid \square'_1) \\ &\xrightarrow{n} (\text{if In T b then } inv^+(AP) \text{ } inv(AOP) \\ &\quad \text{else } inv(AQ) \text{ } inv(AQ) \text{ end } (pa, A) \mid \square'''_1) \\ &\xrightarrow{s^*} (\text{if In T b then } IP' \text{ } inv(AOP) \text{ else } inv(AQ) \text{ } inv(AQ) \\ &\quad \text{end } (pa, A) \mid \square'''_1) \end{aligned} \tag{6.1}$$

for some program IP' and program states \square'_1, \square'''_1 such that $\square'_1 \approx \square'$ and $\square'''_1 \approx \square'''$.

Returning to our assumed execution, repeated use of [I2a] (from conclusion to premises) allows us to obtain the shorter execution (as the conditional must close)

$$(AP \mid \square) \xrightarrow{m} (AP'' \mid \square'') \rightarrow_s^* (AP' \mid \square'') \xrightarrow{\circ}^* (\text{skip } I_1 \mid \square''')$$

With this guaranteed to be a uniform execution and to begin with an identifier step, application of the induction hypothesis of Part 2 of the Program Property (Lemma 11) gives

$$(inv^+(AP) \mid \square'''_1) \xrightarrow{\circ}^* (AR'' \mid \square''_1) \xrightarrow{m} (AR' \mid \square_1) \xrightarrow{s^*} (AR \mid \square_1)$$

for some programs AR'', AR', AR such that AR is a skip equivalent, and program states $\square'''_1, \square''_1, \square_1$ such that $\square'''_1 \approx \square'''$, $\square''_1 \approx \square''$ and $\square_1 \approx \square$. Through repeated use of the corresponding inverse rule [I2r] (premises to conclusion), we have the execution

$$\begin{aligned} &(\text{if In T b then } inv^+(AP) \text{ } inv(AOP) \text{ else } inv(AQ) \text{ } inv(AQ) \text{ end } (pa, A) \mid \square'''_1) \\ &\xrightarrow{\circ}^* (\text{if In T b then } AR'' \text{ } inv(AOP) \text{ else } inv(AQ) \text{ } inv(AQ) \text{ end } (pa, A) \mid \square''_1) \\ &\xrightarrow{m} (\text{if In T b then } AR' \text{ } inv(AOP) \text{ else } inv(AQ) \text{ } inv(AQ) \text{ end } (pa, A) \mid \square_1) \\ &\xrightarrow{s^*} (\text{if In T b then } AR \text{ } inv(AOP) \text{ else } inv(AQ) \text{ } inv(AQ) \text{ end } (pa, A) \mid \square_1) \end{aligned} \tag{6.2}$$

Since \mathbf{AR} is either skip or a skip equivalent produced via the induction hypothesis, by the definition of skip equivalents, we can conclude that $\mathbf{if\ In\ T\ b\ then\ AR\ inv(AOP)}$ $\mathbf{else\ inv(AQ)\ inv(AQ)\ end\ (pa,A)}$ is also a skip equivalent (see 5.7 in Figure 5.6) of Chapter 5. Note that if there are skip steps to apply within 6.1, then those same steps in the same order will be present within the first transition of 6.2. Through the composition of the executions 6.1 and 6.2 in that order, we get

$$\begin{aligned}
& (\mathbf{if\ In\ b\ b\ then\ inv^+(AP)\ inv(AOP)\ else\ inv(AQ)\ inv(AQ)} \\
& \quad \mathbf{end\ (pa,n:A)\ |\ \Box'_1}) \\
& \xrightarrow{n} (\mathbf{if\ In\ T\ b\ then\ inv^+(AP)\ inv(AOP)\ else\ inv(AQ)\ inv(AQ)} \\
& \quad \mathbf{end\ (pa,A)\ |\ \Box'''_1}) \\
& \xrightarrow{\circ^*} (\mathbf{if\ In\ T\ b\ then\ AR''\ inv(AOP)\ else\ inv(AQ)\ inv(AQ)\ end\ (pa,A)\ |\ \Box''_1}) \\
& \xrightarrow{m} (\mathbf{if\ In\ T\ b\ then\ AR'\ inv(AOP)\ else\ inv(AQ)\ inv(AQ)\ end\ (pa,A)\ |\ \Box_1}) \\
& \xrightarrow{s^*} (\mathbf{if\ In\ T\ b\ then\ AR\ inv(AOP)\ else\ inv(AQ)\ inv(AQ)\ end\ (pa,A)\ |\ \Box_1})
\end{aligned}$$

which matches our desired execution at each point. Therefore we take $\mathbf{AT'} = \mathbf{if\ In\ T\ b\ then\ AR'\ inv(AOP)\ else\ inv(AQ)\ inv(AQ)\ end\ (pa,A)}$ and $\mathbf{AT = if\ In\ T\ b\ then\ AR\ inv(AOP)\ else\ inv(AQ)\ inv(AQ)\ end\ (pa,A)}$, meaning this case holds.

Case 9.6. (Conditional [I3a]) Consider an identifier step that comes from the false branch of a conditional statement. This case matches Case 9.5 but uses [I3a] in place of [I2a].

Case 9.7. (Conditional [I4a]) Consider the closing of a conditional statement (the final step). Let $\mathbf{AS = if\ In\ T\ b\ then\ skip\ I\ AP\ else\ AQ\ AQ\ end\ (pa,A)}$ and the initial program state $\Box = (\sigma, \gamma, \mu, \beta, \delta)$. Assume the following execution exists with the first transition via [I4a]

$$\begin{aligned}
(\mathbf{AS\ |\ \Box}) &= (\mathbf{if\ In\ T\ b\ then\ skip\ I\ AP\ else\ AQ\ AQ\ end\ (pa,A)\ |\ \Box}) \\
&\xrightarrow{m} (\mathbf{skip\ (pa,m:A)\ |\ \Box'})
\end{aligned}$$

for program state \Box' such that $\Box' = \Box[\delta[(\mathbf{m}, \mathbf{T}) \rightarrow \mathbf{B}]]$. Using [I4a] means all premises were valid at the time, namely $\mathbf{m = next()}$. Note that no skip steps or further execution is possible.

We need to show that there exists an execution

$$(\mathbf{inv^+(AS)\ |\ \Box'_1}) \xrightarrow{\circ^*} (\mathbf{AT''\ |\ \Box''_1}) \xrightarrow{m} (\mathbf{AT'\ |\ \Box_1}) \xrightarrow{s^*} (\mathbf{AT\ |\ \Box_1})$$

for some statements $\mathbf{AT''}$, $\mathbf{AT'}$, \mathbf{AT} , and program states \Box'_1 , \Box''_1 , \Box_1 such that $\Box'_1 \approx \Box'$, $\Box''_1 \approx \Box''$ and $\Box_1 \approx \Box$. From 5.27 in Figure 5.7 of Chapter 5, we have that $\mathbf{inv^+(AS) = if\ In\ b\ b\ then\ abort\ inv(AP)\ else\ inv(AQ)\ inv(AQ)\ end\ (pa,m:A)}$, with abort

inserted to stop the inverse execution at the beginning of the execution of the inverse true branch.

From $inv^+(\mathbf{AS})$, it is clear that we are immediately in a state such that the corresponding inverse rule [I1rT] can be applied. We therefore take $\mathbf{AT}'' = inv^+(\mathbf{AS})$ and $\square_1'' = \square_1'$, giving the first part of our desired execution with length 0.

From the program state \square_1'' (recall $\square_1'' = \square_1'$), we have that $\mathbf{m} = previous()$ (since no other identifiers were used), the identifier stack for this statement is equal to $\mathbf{m}:\mathbf{A}$ and $\square_1''(\delta_1''(\mathbf{B})) = (\mathbf{m}, \mathbf{T}) : \mathbf{B}'$ where \mathbf{B}' is the remaining stack. With all premises of the rule [I1rT] shown to be valid, and that no other rule can be applied due to these premises, application of the rule [I1rT] (the corresponding inverse identifier rule as expected) gives us

$$\begin{aligned}
(\mathbf{AT}'' \mid \square_1'') &= (\text{if In } \mathbf{b} \text{ then abort } inv(\mathbf{AP}) \text{ else } inv(\mathbf{AQ}) inv(\mathbf{AQ}) \\
&\quad \text{end } (\mathbf{pa}, \mathbf{m}:\mathbf{A}) \mid \square_1'') \\
&\xrightarrow{m} (\text{if In } \mathbf{T} \text{ then abort } inv(\mathbf{AP}) \text{ else } inv(\mathbf{AQ}) inv(\mathbf{AQ}) \\
&\quad \text{end } (\mathbf{pa}, \mathbf{A}) \mid \square_1) \\
&\xrightarrow{s^*} (\text{if In } \mathbf{T} \text{ then } \mathbf{AP}' inv(\mathbf{AP}) \text{ else } inv(\mathbf{AQ}) inv(\mathbf{AQ}) \\
&\quad \text{end } (\mathbf{pa}, \mathbf{A}) \mid \square_1)
\end{aligned}$$

for some program \mathbf{AP}' and program state \square_1 such that $\square_1 = \square_1''[\delta_1''[\mathbf{B}/\mathbf{B}']]$, meaning $\square_1 \approx \square$ as required. The abort statement forcibly stops the execution at that point, meaning no skip steps are available in the final transition. Therefore $\mathbf{AP}' = \text{abort}$. We can take $\mathbf{AT}' = \text{if In } \mathbf{T} \text{ then abort } inv(\mathbf{AP}) \text{ else } inv(\mathbf{AQ}) inv(\mathbf{AQ}) \text{ end } (\mathbf{pa}, \mathbf{A})$, and $\mathbf{AT} = \mathbf{AT}'$. As such, this shows our desired execution and therefore this case to hold.

Case 9.8. (Conditional [I5a]) Consider the closing of a conditional statement that previously executed the false branch. This follows Case 9.7 and uses the rules [I5a] and [I1rF] instead of [I4a] and [I1rT] respectively.

While Loops

Case 9.9. (Loop [W2aP]) Consider the closing of a while loop. This final step of the forward execution will be the first step of the reverse execution. To stop the execution after the matching inverse step, an abort statement is manually inserted to ensure the rule [W2aP] is used. In normal execution outside of the proof, this abort will not appear and the rule [W2a] is used. Therefore let $\mathbf{AS} = \text{while Wn } \mathbf{b}$

`do abort end (pa,A)` and the initial program state $\square = (\sigma, \gamma, \mu, \beta, \delta)$. Assume the following execution exists with the first transition via [W2aP]

$$(\text{AS} \mid \square) = (\text{while Wn b do abort end (pa,A)} \mid \square) \xrightarrow{m} (\text{skip (pa,m:A)} \mid \square')$$

for program state \square' such that $\square' = \square[\delta[(\mathbf{m}, \mathbf{T}, \mathbf{T}) \rightarrow \mathbf{W}, (\mathbf{m}, \mathbf{C}) \rightarrow \mathbf{WI}], \beta[\mathbf{Wn}]]$. Note that no skip steps are available, and that a true abort flag is also pushed to stack \mathbf{W} . All premises of [W2aP] must have been valid, namely $\mathbf{m} = \text{next}()$, $\beta(\mathbf{Wn}) = \text{while Wn b do AP end (pa,A)}$ (a mapping exists as the loop must have started), $(\mathbf{b} \mid \text{pa} \mid \square) \hookrightarrow_b^* (\mathbf{F} \mid \square)$ (the condition evaluates to false) and the body of the loop is `abort`.

We need to show that there exists an execution

$$(\text{inv}^+(\text{AS}) \mid \square'_1) \xrightarrow{\circ^*} (\text{AT}'' \mid \square''_1) \xrightarrow{m} (\text{AT}' \mid \square_1) \xrightarrow{s^*} (\text{AT} \mid \square_1)$$

for some statements AT'' , AT' , AT , and program states \square'_1 , \square''_1 , \square_1 such that $\square'_1 \approx \square'$, $\square''_1 \approx \square''$ and $\square_1 \approx \square$. From 5.30 in Figure 5.7 of Chapter 5, we have that $\text{inv}^+(\text{AS}) = \text{while Wn b do inv(AP) end (pa,A)}$.

The assumed forward execution is the closing of a while loop (the final step), meaning the corresponding inverse step will be the opening of the inverse loop (the first step). Therefore we can take $\text{AT}'' = \text{inv}^+(\text{AS})$ and $\square''_1 = \square'_1$, giving the first part of our desired execution with length 0.

From program state \square''_1 , we have that $\mathbf{m} = \text{previous}()$ (since no other identifiers were saved), the identifier stack for this statement is equal to $\mathbf{m:A}$, $\square''_1(\delta''_1(\mathbf{W}) = (\mathbf{m}, \mathbf{T}, \mathbf{T}) : \mathbf{W}', \delta''_1(\mathbf{WI}) = (\mathbf{m}, \mathbf{C}) : \mathbf{WI}'))$, $\beta''_1(\mathbf{Wn}) = \text{und})$. This means that all premises of the corresponding inverse rule [W3rP] are valid, and that no other rule is applicable. Using the rule [W3rP] (the corresponding inverse identifier rule as expected) we get

$$\begin{aligned} (\text{AT}'' \mid \square''_1) &= (\text{while Wn b do inv(AP) end (pa,m:A)} \mid \square''_1) \\ &\xrightarrow{m} (\text{while Wn T do abort end (pa,A)} \mid \square_1) \end{aligned}$$

for program state \square_1 such that $\square_1 = \square''_1[\delta''_1[\mathbf{W}/\mathbf{W}', \mathbf{WI}/\mathbf{WI}'], \beta''_1[\mathbf{Wn} \mapsto \text{while Wn b do inv(AP) end (pa,A)}]]$, meaning $\square_1 \approx \square$ as required. The abort is added by this rule (and crucially not by the normal rule [W3r]) in order to stop the execution at this point. With the program state matching with our desired execution, we can take AT' to be `while Wn T do abort end (pa,A)`, which is a valid skip equivalent. Since no skip steps are available (due to abort), we can also take $\text{AT} = \text{AT}'$ and complete the desired execution showing the case to hold.

Case 9.10. (Loop [W3a]) Consider the opening of a while loop that performs at least one iteration (the condition must evaluate to true as Case 9.2 considers loops with zero iterations). Let $\text{AS} = \text{while Wn b do AP end (pa,A)}$ and the initial

program state $\square = (\sigma, \gamma, \mu, \beta, \delta)$. Assume the following execution exists with the first transition via [W3a]

$$\begin{aligned} (\text{AS} \mid \square) &= (\text{while Wn b do AP end (pa,A)} \mid \square) \\ &\xrightarrow{m} (\text{while Wn T do AP}' \text{ end (pa,m:A)} \mid \square'') \\ &\rightarrow_s^* (\text{while Wn T do AP}'' \text{ end (pa,m:A)} \mid \square'') \xrightarrow{\circ}^* (\text{skip (pa,A')} \mid \square') \end{aligned}$$

for some program AP' such that $\text{AP}' = \text{reL}(\text{AP})$, and program states \square'' , \square' such that $\square'' = \square[\delta[(\text{m}, \text{F}, \text{F}) \Rightarrow \text{W}], \beta[\text{Wn} \rightarrow \text{while Wn b do AP}' \text{ end (pa,m:A)}]]$. For [W3a] to be applied as above, all premises must have been valid prior to it. This means that $\text{m} = \text{next}()$, $\beta(\text{Wn}) = \text{und}$ (the while loop has not yet started), $(\text{b pa} \mid \square) \hookrightarrow_b^* (\text{T} \mid \square)$ (the condition evaluates to true) and the body of the loop is not equal to **abort**. Note A' is a version of the persistent identifier stack for this loop statement ending with the sub-stack (m:A) . From this point on, we use AS' to denote $\text{while Wn T do AP}'' \text{ end (pa,m:A)}$.

We need to show that there exists an execution

$$(\text{inv}^+(\text{AS}) \mid \square'_1) \xrightarrow{\circ}^* (\text{AT}'' \mid \square''_1) \xrightarrow{m} (\text{AT}' \mid \square_1) \rightsquigarrow_s^* (\text{AT} \mid \square_1)$$

for some statements AT'' , AT' , AT , and program states \square'_1 , \square''_1 , \square_1 such that $\square'_1 \approx \square'$, $\square''_1 \approx \square''$ and $\square_1 \approx \square$. From 5.29 in Figure 5.7 of Chapter 5, we have that $\text{inv}^+(\text{AS}) = \text{while Wn b do inv(AP) end (pa,A)}$ (note the use of $\text{inv}()$ as AP is guaranteed to be a full program).

Firstly, the part of the assumed execution

$$\begin{aligned} &(\text{while Wn T do AP}' \text{ end (pa,m:A)} \mid \square'') \\ &\rightarrow_s^* (\text{while Wn T do AP}'' \text{ end (pa,m:A)} \mid \square'') \end{aligned}$$

only exists if the loop body AP' begins with skip steps. If this is the case, the inverted version of this group of skip steps will be performed in the inverse execution after the previous inverse identifier rule. Therefore this will be provided via the induction hypothesis use shown later.

Next, the execution

$$(\text{AS}' \mid \square'') = (\text{while Wn T do AP}'' \text{ end (pa,m:A)} \mid \square'') \xrightarrow{\circ}^* (\text{skip (pa,A')} \mid \square')$$

must be shorter than our original (since the while loop is first started). From 5.30 in Figure 5.7 of Chapter 5, we have that $\text{inv}^+(\text{AS}') = \text{while Wn b do inv(AP) end (pa,A)}$. Then application of the induction hypothesis of the Statement Property

(Lemma 10) on this shorter execution (which is guaranteed to be uniform and to begin with an identifier step) gives

$$(inv^+(AS') \mid \square'_1) = (\text{while } Wn \ b \ \text{do } inv(AP) \ \text{end } (pa, m:A) \mid \square'_1) \overset{\circ}{\rightsquigarrow}^* (AR \mid \square''_1)$$

by a sequence of rule applications SR, for program AR such that AR is a skip equivalent of some program. As this shorter execution begins with an identifier step from within the while loop body, this shorter execution is guaranteed to stop via the modified rules [W2aPI] or [W4aPI] as shown in Case 9.12. This means that $AR = \text{while } Wn \ T \ \text{do } \text{abort} \ \text{end } (pa, m:A)$ (with abort coming from the use of the modified rules).

Now compare the programs $inv^+(AS)$ and $inv^+(AS')$. Since both begin with the same prefix of execution, then by the same sequence of rules SR, we obtain

$$(inv^+(AS) \mid \square'_1) = (\text{while } Wn \ b \ \text{do } inv(AP) \ \text{end } (pa, A) \mid \square'_1) \overset{\circ}{\rightsquigarrow}^* (AT'' \mid \square''_1)$$

for some programs AT'' . With the program AR containing an abort statement to stop the execution at the corresponding point, AT'' will be identical but without this abort. However, since this will be replaced with **skip**, a single application of the skip rule [W6a] occurs to reset the while loop. Therefore $AT'' = \text{while } Wn \ b \ \text{do } inv(AP) \ \text{end } (pa, m:A)$.

Consider the current state \square''_1 . We have $m = previous()$ (with all subsequently used identifiers reversed via the induction hypothesis), the identifier stack for this loop is equal to $m:A$ (m as its head), $\beta''_1(Wn) = def$ (there is a mapping for this loop since it will have already started) and that $\delta''_1(W) = (m, F, F)$ (the stack W has the triple (m, F, F) as its head). All premises of the rule [W2r] are valid (and as such mean no other rule is applicable), giving us

$$(\text{while } Wn \ b \ \text{do } inv^+(AP) \ \text{end } (pa, m:A) \mid \square''_1) \overset{m}{\rightsquigarrow} (\text{skip } (pa, A) \mid \square_1)$$

for some program state $\square_1 = \square''_1[\delta''_1[W/W], \beta''_1[Wn]]$, such that $\square_1 \approx \square$ as required. The program state matches at each required point with our desired execution, meaning we can take AT' to be **skip** (pa, A) and AT to be AT' . Since AT has reached skip, this case is shown to be valid.

Case 9.11. (Loop [W4aP]) Consider the loop condition evaluation for any iteration that is not the first or the last (each have separate rules [W1a]/[W3a]/[W2a]. To trigger the modified rule [W4aP], an abort is inserted manually into the initial program. This cannot occur during normal execution and the rule [W4a] will be applied instead. Let $AS = \text{while } Wn \ b \ \text{do } \text{abort} \ \text{end } (pa, A)$ and the initial pro-

gram state $\square = (\sigma, \gamma, \mu, \beta, \delta)$. Assume the following execution exists with the first transition via [W4aP]

$$\begin{aligned} (\text{AS} \mid \square) &= (\text{while Wn b do abort end (pa,A)} \mid \square) \\ &\xrightarrow{m} (\text{while Wn T do AP}' \text{ end (pa,m:A)} \mid \square'') \\ &\rightarrow_s^* (\text{while Wn T do AP}'' \text{ end (pa,m:A)} \mid \square'') \xrightarrow{\circ}^* (\text{skip (pa,A')} \mid \square') \end{aligned}$$

for some programs AP'' , AP' such that AP' is a renamed version of AP and states $\square'' = \square[\delta[(\text{m}, \text{T}, \text{T}) \rightarrow \text{W}], \beta[\text{Wn} \rightarrow \text{while Wn b do AP}' \text{ end (pa,m:A)}]]$, and \square' . Application of the rule [W2aP] means the premises were valid at that point, namely that $\text{m} = \text{next}()$, $\beta(\text{Wn}) = \text{while Wn b do AP end (pa,m:A)}$ (there is no mapping for this while loop as it has not yet started), $(\text{b pa} \mid \square) \hookrightarrow_b^* (\text{T} \mid \square)$ (the condition evaluates to true) and the body of the loop is equal to **abort**. Note A' is a version of the persistent identifier stack for this loop statement that ends with the sub-stack (m:A) . From this point on, we use AS' to denote $\text{while Wn T do AP}'' \text{ end (pa,m:A)}$.

We need to show that there exists an execution

$$(\text{inv}^+(\text{AS}) \mid \square'_1) \xrightarrow{\circ}^* (\text{AT}'' \mid \square''_1) \xrightarrow{m} (\text{AT}' \mid \square_1) \rightsquigarrow_s^* (\text{AT} \mid \square_1)$$

for some statements AT'' , AT' , AT , and program states \square'_1 , \square''_1 , \square_1 such that $\square'_1 \approx \square'$, $\square''_1 \approx \square''$ and $\square_1 \approx \square$. From 5.30 in Figure 5.7 of Chapter 5, we have that $\text{inv}^+(\text{AS}) = \text{while Wn b do inv(AP) end (pa,A')}$.

We first note that the execution

$$\begin{aligned} &(\text{while Wn T do AP}' \text{ end (pa,m:A)} \mid \square'') \\ &\rightarrow_s^* (\text{while Wn T do AP}'' \text{ end (pa,m:A)} \mid \square'') \end{aligned}$$

will have length in all cases except where the loop body AP' begins with at least one skip step. In this situation, the inverted version of this group of skip steps will be applied immediately following the previous inverse identifier step. This means due to uniformity that these will be given within the induction hypothesis use on the rest of the forward execution.

Returning to our assumed transition sequence above, the execution

$$(\text{AS}' \mid \square'') = (\text{while Wn T do AP}'' \text{ end (pa,m:A)} \mid \square'') \xrightarrow{\circ}^* (\text{skip (pa,A')} \mid \square')$$

must be shorter than our original (since the iteration of the loop must have first been started). From 5.30 in Figure 5.7 of Chapter 5, we have that $\text{inv}^+(\text{AS}') = \text{while Wn b do inv(AP) end (pa,A')}$. Then application of the induction hypothesis of the

Statement Property (Lemma 10) on this shorter execution (which is guaranteed to be uniform and to begin with an identifier step) gives

$$(inv^+(AS') \mid \square'_1) = (\text{while } Wn \ b \ \text{do } inv(AP) \ \text{end } (pa, A') \mid \square'_1) \overset{\circ}{\rightsquigarrow}^* (AR \mid \square''_1)$$

by a sequence of rule applications SR, for program AR such that AR is a skip equivalent of some program. As in Case 9.10, this shorter execution (that must start with an identifier rule) is guaranteed to stop via the modified rules [W2aPI] or [W4aPI] as shown in Case 9.12. This means that $AR = \text{while } Wn \ T \ \text{do } \text{abort} \ \text{end } (pa, m:A)$ (with abort coming from the use of the modified rules).

Now compare the programs $inv^+(AS)$ and $inv^+(AS')$. As in Case 9.10, both start with identical transition sequences meaning the sequence of rules SR gives

$$(inv^+(AS) \mid \square'_1) = (\text{while } Wn \ b \ \text{do } inv(AP) \ \text{end } (pa, A') \mid \square'_1) \overset{\circ}{\rightsquigarrow}^* (AT'' \mid \square''_1)$$

for some programs AT'' . With the program AR containing an abort statement to stop the execution at the corresponding point, AT'' will be identical but without this abort. However, since this will be replaced with **skip**, a single application of the skip rule [W6a] occurs to reset the while loop. Therefore $AT'' = \text{while } Wn \ b \ \text{do } inv^+(AP) \ \text{end } (pa, m:A)$.

We are now in the program state \square''_1 . We have that $m = previous()$ (with all subsequently used identifiers reversed via the induction hypothesis), the identifier stack for this loop will have m as its head, $\beta''_1(Wn) = def$ (there is a mapping for this loop since it will have already started), that $\delta''_1(W) = (m, T, T)$ (the stack W has the triple (m, T, T) as its head) and $\delta''_1(WI) = m, C$ (the stack WI has the pair (m, C) as its head). This means all premises of the rule [W4rP] to be valid (and that no other rule is applicable). Therefore we have

$$\begin{aligned} & (\text{while } Wn \ b \ \text{then } inv(AP) \ \text{end } (pa, m:A) \mid \square''_1) \\ & \overset{m}{\rightsquigarrow} (\text{while } Wn \ b \ \text{then } \text{abort} \ \text{end } (pa, A) \mid \square_1) \end{aligned}$$

for some program state $\square_1 = \square''_1[\delta''_1[W/W'], \beta''_1[Wn]]$, such that $\square_1 \approx \square$ as required. With no skip steps available and all stores matching at the required positions, we can take AT' to be $\text{while } Wn \ b \ \text{then } \text{abort} \ \text{end } (pa, A)$ (a valid skip equivalent) and AT to be AT' , showing our desired execution to be valid. Therefore this holds.

Case 9.12. (Loop [W5a]) Consider an identifier rule from within a loop body. Let $AS = \text{while } Wn \ T \ \text{do } AP \ \text{end } (pa, A)$ and the initial state $\square = (\sigma, \gamma, \mu, \beta, \delta)$. Assume the following execution exists with the first step via [W5a]

$$\begin{aligned} (AS \mid \square) &= (\text{while } Wn \ b \ \text{do } AP \ \text{end } (pa, A) \mid \square) \\ &\xrightarrow{m} (\text{while } Wn \ T \ \text{do } AP' \ \text{end } (pa, m:A) \mid \square'') \\ &\rightarrow_s^* (\text{while } Wn \ T \ \text{do } AP'' \ \text{end } (pa, m:A) \mid \square'') \\ &\xrightarrow{\circ}^* (\text{skip } (pa, A') \mid \square') \end{aligned}$$

for some programs AP' and AP'' , and program states \square'' and \square' . Note A' is a modified version of the identifier stack that ends with the sub-stack $m:A$. From here, let $AS' = \text{while } Wn \ T \ \text{do } AP'' \ \text{end } (pa, m:A)$. The while loop must have started, meaning $\beta(Wn) = \text{while } Wn \ b \ \text{do } AQ \ \text{end } (pa, A)$.

We need to show that there exists an execution

$$(inv^+(AS) \mid \square'_1) \xrightarrow{\circ}^* (AT'' \mid \square''_1) \xrightarrow{m} (AT' \mid \square_1) \rightsquigarrow_s^* (AT \mid \square_1)$$

for statements AT'' , AT' , AT , and states \square'_1 , \square''_1 , \square_1 such that $\square'_1 \approx \square'$, $\square''_1 \approx \square''$ and $\square_1 \approx \square$. From 5.30 in Figure 5.7 of Chapter 5, we have that $inv^+(AS) = \text{while } Wn \ b \ \text{do } inv(AP) \ \text{end } (pa, A)$.

In order to correctly stop the inverse execution, we must consider two possible cases. The first case is where the first transition from our assumed execution is from within the final iteration of the loop, and the second case is where it is not. Since the function $inv^+(\cdot)$ cannot insert the abort as further iterations may need to be inverted first, each of our cases uses one of our modified semantic rules, namely [W2aPI] or [W4aPI] respectively. We rewrite our execution to show the next identifier step made by the loop itself (to next evaluate the condition), where the partial program will be saved. This will then ensure that the inverse partial rules are used, namely [W3rPI] and [W4rPI] to insert the abort at the correct position. Recall the syntax of while loops for these rules to apply require a copy of the partial body AP to be added.

Final iteration Assuming the transition is from within the final iteration, our assumed execution can be rewritten as

$$\begin{aligned} (AS \mid \square) &= (\text{while } Wn \ b \ \text{do } AP \ AP \ \text{end } (pa, A) \mid \square) \\ &\xrightarrow{m} (\text{while } Wn \ T \ \text{do } AP' \ AP \ \text{end } (pa, m:A) \mid \square'') \\ &\rightarrow_s^* (\text{while } Wn \ T \ \text{do } AP'' \ AP \ \text{end } (pa, m:A) \mid \square'') \\ &\xrightarrow{\circ}^* (\text{while } Wn \ T \ \text{do } \text{skip } I \ AP \ \text{end } (pa, m:A) \mid \square''') \\ &\rightarrow_s (\text{while } Wn \ T \ \text{do } AQ \ AP \ \text{end } (pa, m:A) \mid \square''') \xrightarrow{n} (\text{skip } (pa, A') \mid \square') \end{aligned}$$

for some program state \square''' and program AQ such that AQ is the full loop body (retrieved from the while environment), where the penultimate transition is via [W6a] and the final step (using n) is via [W2aPI].

Repeated use of [W5a] (from conclusion to premises) gives us

$$(AP \mid \square) \xrightarrow{m} (AP' \mid \square'') \rightarrow_s^* (AP'' \mid \square'') \xrightarrow{\circ}^* (\text{skip } I \mid \square''')$$

which must be shorter than our original execution (as the while loop itself has to finish). Applying the induction hypothesis of Part 2 of Lemma 11 means we obtain

$$(inv^+(AP) \mid \square_1''') \xrightarrow{\circ}^* (AR'' \mid \square_1'') \xrightarrow{m} (AR' \mid \square_1) \rightsquigarrow_s^* (AR \mid \square_1)$$

for some programs AR'' , AR' , AR such that AR is a valid skip equivalent of some program. Repeated use of the rule [W5r] gives

$$\begin{aligned} & (\text{while } Wn \ T \ \text{do } inv^+(AP) \ \text{end } (pa, m:A) \mid \square_1''') \\ & \xrightarrow{\circ}^* (\text{while } Wn \ T \ \text{do } AR'' \ \text{end } (pa, m:A) \mid \square_1'') \\ & \xrightarrow{m} (\text{while } Wn \ T \ \text{do } AR' \ \text{end } (pa, A) \mid \square_1'') \\ & \rightsquigarrow_s^* (\text{while } Wn \ T \ \text{do } AR \ \text{end } (pa, A) \mid \square_1) \end{aligned} \tag{6.3}$$

from which a single application of [W6r] will then reset the while loop (recall a while loop is a special case of skip equivalent). This gives us the majority of our desired execution, with only the last step of the inverse execution left to consider. From our assumed execution (with the copy of the loop body omitted),

$$(\text{while } Wn \ T \ \text{do } AQ \ \text{end } (pa, m:A) \mid \square''') \xrightarrow{n} (\text{skip } (pa, A') \mid \square')$$

takes a single step, with no skip steps available after. The corresponding inverse rule [W3aPI] gives the execution (starting from the inverted complete program as this must be very first step)

$$(inv^+(AS) \mid \square_1') \xrightarrow{n} (\text{while } Wn \ b \ \text{do } inv^+(AP) \ \text{end } (pa, m:A) \mid \square_1''') \tag{6.4}$$

as expected. From here, composition of the executions 6.4 and 6.3 (in that order) gives our desired execution and therefore shows that this part of the case holds.

nth iteration We now consider executions where the first step is from the loop body of any iteration of a loop other than the last. Such executions can be rewritten as

$$\begin{aligned}
(AS \mid \square) &= (\text{while } Wn \ b \ \text{do } AP \ AP \ \text{end } (pa, A) \mid \square) \\
&\xrightarrow{m} (\text{while } Wn \ T \ \text{do } AP' \ AP \ \text{end } (pa, m:A) \mid \square'') \\
&\rightarrow_s^* (\text{while } Wn \ T \ \text{do } AP'' \ AP \ \text{end } (pa, m:A) \mid \square'') \\
&\xrightarrow{\circ}^* (\text{while } Wn \ T \ \text{do } \text{skip } I \ AP \ \text{end } (pa, m:A) \mid \square''') \\
&\rightarrow_s (\text{while } Wn \ T \ \text{do } AQ \ AP \ \text{end } (pa, m:A) \mid \square''') \\
&\xrightarrow{n} (\text{while } Wn \ T \ \text{do } AQ' \ \text{skip} \ \text{end } (pa, n:m:A) \mid \square''') \\
&\rightarrow_s^* (\text{while } Wn \ T \ \text{do } AQ'' \ \text{skip} \ \text{end } (pa, n:m:A) \mid \square''') \\
&\xrightarrow{\circ}^* (\text{skip } (pa, A') \mid \square')
\end{aligned}$$

where the identifier step using n is via [W4aPI], for some program state \square''' and programs AQ, AQ', AQ'' such that AQ is the full loop body (retrieved from the while environment).

The proof of such an execution follows closely to that for the first iteration above. The major difference is there is an extra use of the induction hypothesis, specifically on the execution

$$(\text{while } Wn \ T \ \text{do } AQ'' \ \text{end } (pa, n:m:A) \mid \square''') \xrightarrow{\circ}^* (\text{skip } (pa, A') \mid \square')$$

that does not appear in the first part of this case. All other parts of the proof follow as above. We therefore conclude that this part of the case holds, meaning we have shown this case to hold.

Block statements

Case 9.13. Block [B1a] This concerns identifier step from within a block. This follows Case 9.5, but uses [B2a] in place of [I2a], and [B2a] in place of [I4a].

Since we assume that local variables and procedures can only be declared within block statements, we consider each case of declaration and removal statements here.

Case 9.14. Variable declaration [L1a] Declaring a local variable completes in a single step. This follows Case 9.1 and so it omitted.

Case 9.15. Procedure declaration [L2a] A single application of [L2a] declares a procedure. The proof of this case is omitted as it follows Case 9.1.

Case 9.16. Variable removal [H1a] As with the declaration of a local variable, removal takes a single step. This proof is not shown as it follows closely to Case 9.1.

Case 9.17. Procedure removal [H2a] The rule [H2a] takes a single step to remove a procedure. The proof is not listed as it follows Case 9.1.

Procedure call statements

Case 9.18. Call [G1a] Consider the opening of a procedure call, which declares a local copy of the procedure body and produces the corresponding `runc` construct. This follows with the opening of a conditional statement in Case 9.3 (though with changes made to the procedure environment μ) using rules [G1a] and [G3r] in place of [I1aT] and [I4r], and so is omitted.

Case 9.19. Call [G2a] Consider an execution beginning with an identifier step from within a procedure body. This follows the case of such steps from within a conditional statement, namely Case 9.5. A partial execution beginning within a call statement uses the modified rule [G3aPI], and follows as in Case 9.12.

Case 9.20. Call [G3a] The closing of a procedure call will remove the local copy of the procedure body for this specific call statement, after saving any identifiers assigned to it. This case follows with the closing of a conditional statement, namely Case 9.7 with rules [G3a] and [G1r] in place of [I4a] and [I1rT]. A partial execution beginning at the closure of a block uses the modified rule [G3aP] and follows Case 9.9.

All inductive cases are therefore valid. Since all base cases also hold, we can conclude that Lemma 10 (Statement Property) is valid. \square

6.6 Proof of Program Property

In this section we give the proof of Lemma 11 (Program Property). By abuse of notation, we use **AP** to represent either a complete or partially executed program. In cases considering full program execution (e.g. Case 10.1), we ignore the abort statement introduced by inv^+ as the program stops as desired.

Lemma 11 (Program Property). Let **AP** be a complete program (syntax from Figure 6.3) in Part 1, and either a complete or partially executed annotated program in Part 2. Further let \square be the tuple $(\sigma, \gamma, \mu, \beta, \delta)$ of initial program environments.

Part 1. If a uniform forward execution $(\mathbf{AP} \mid \square) \xrightarrow{U}^* (\mathbf{AP}' \mid \square'') \xrightarrow{U}^* (\mathbf{skip} \mid \square')$ exists for some program \mathbf{AP}' and program states \square'', \square' such that $\square'' = \square$ (by Lemma 1 of Chapter 3), then there exists a uniform inverse execution $(inv^+(\mathbf{AP}) \mid \square'_1) \xrightarrow{U}^* (\mathbf{AQ}' \mid \square''_1) \xrightarrow{U}^* (\mathbf{AQ} \mid \square_1)$ for some programs $\mathbf{AQ}', \mathbf{AQ}$ such that \mathbf{AQ} is skip if \mathbf{AP} is a complete program and a skip equivalent otherwise, and program states $\square'_1, \square''_1, \square_1$ such that $\square'_1 \approx \square', \square''_1 \approx \square''$ and $\square_1 \approx \square$.

Part 2. If a uniform forward execution $(AP \mid \square) \xrightarrow{m}_U (AP' \mid \square'') \xrightarrow{*}_s (AP'' \mid \square'') \xrightarrow{*}_s (\text{skip } I \mid \square')$ exists for some program AP' and program states \square'' and \square' , then there exists a uniform inverse execution $(inv^+(AP) \mid \square'_1) \xrightarrow{*}_s (AQ'' \mid \square''_1) \xrightarrow{m}_U (AQ' \mid \square_1) \xrightarrow{*}_s (AQ \mid \square_1)$ for some programs AQ'' , AQ' , AQ is skip if AP is a complete program and a skip equivalent otherwise, and states \square'_1 , \square''_1 , \square_1 such that $\square'_1 \approx \square'$, $\square''_1 \approx \square''$ and $\square_1 \approx \square$.

All executions from here are uniform and so we omit the specific notation. For example, a uniform execution $(P \mid \square) \xrightarrow{m}_U (P' \mid \square')$ is written as $(P \mid \square) \xrightarrow{m} (P' \mid \square')$.

Proof. This proof is by mutual induction of the Program Property (this lemma) and the Statement Property (Lemma 10), on the length of the executions $(AP \mid \square) \xrightarrow{*}_s (\text{skip } I \mid \square')$ and $(AS \mid \square) \xrightarrow{*}_s (\text{skip } I \mid \square')$ respectively. We now consider the two types of possible executions, namely those that start with at least one skip step (**Part 1**) and those that start with an identifier step (**Part 2**).

6.6.1 Proof of Part 1

Consider all executions that begin with at least one skip step, of the form $(AP \mid \square) \rightarrow_s^* (AP' \mid \square) \xrightarrow{*}_s (\text{skip } I \mid \square')$, where AP is a complete program of the following syntax (recalled from Figure 6.3) as no partial programs can begin with a skip step.

$$\begin{aligned} P &::= S \mid \text{skip}; AP \mid P \text{ par } P \\ S &::= \text{skip} \mid \text{begin } Bn \ P \ AP \ \text{end} \end{aligned}$$

Since no executions of length 0 exist, we consider our base cases to be any execution of length 1. There are three examples of such cases: two sequentially composed skip statements (Case 10.1), a parallel statement where both component programs are skip (Case 10.2) and a block statement whose body is skip (Case 10.3).

Case 10.1. Skips [S2a] Consider two sequentially composed skip statements. Let AP be the program $\text{skip } I; \text{skip } I'$, and \square be the tuple of initial program state environments $(\sigma, \gamma, \mu, \beta, \delta)$. Assume the following uniform execution via the rule [S2a]

$$(\text{skip } I; \text{skip } I' \mid \square) \rightarrow_s (\text{skip } I'; \mid \square)$$

that completes in a single step and does not alter the program state.

We need to show that there exists an execution

$$(inv^+(AP) \mid \square'_1) \xrightarrow{*}_s (AT' \mid \square_1) \xrightarrow{*}_s (AT \mid \square_1)$$

for some statements AT' , AT , and program states \square'_1 , \square_1 such that $\square'_1 \approx \square'$ and $\square_1 \approx \square$. By 5.20 and 5.22 in Figure 5.7 of Chapter 5, we have that $inv^+(AP) = \text{skip } I'; \text{ skip } I$ (ignoring the abort).

With no further execution, we can take $AT' = inv^+(AP)$ and $\square_1 = \square'_1$. A single use of [S2r] can be applied to remove the first skip statement while not changing the program state, giving the execution

$$(AT' \mid \square_1) \rightsquigarrow_s (\text{skip } I \mid \square_1)$$

Taking $AT = \text{skip } I$, we have shown our desired execution and this case to hold.

Case 10.2. Empty Parallel [P3a] Consider a program containing only a parallel statement where both component programs are single skip statements. Let AP be the program $\text{skip } I \text{ par skip } I'$, and let \square be the tuple of initial program state environments $(\sigma, \gamma, \mu, \beta, \delta)$. Assume the uniform execution via the rule [P3a]

$$(\text{skip } I \text{ par skip } I' \mid \square) \rightarrow_s (\text{skip} \mid \square)$$

with length 0 and no effect on the program state.

We need to show that there exists an execution

$$(inv^+(AP) \mid \square'_1) \rightsquigarrow^{\circ*} (AT' \mid \square_1) \rightsquigarrow_s^* (AT \mid \square_1)$$

for some statements AT' , AT , and program states \square'_1 , \square_1 such that $\square'_1 \approx \square'$ and $\square_1 \approx \square$. By 5.21 and 5.22 in Figure 5.7 of Chapter 5, we have $inv^+(AP) = \text{skip } I \text{ par skip } I'$ (ignoring the abort).

With no further execution, we can take $AT' = inv^+(AP)$ and $\square_1 = \square'_1$. A single application of [P3r] can close the inverted parallel statement while not changing the program state, giving the execution

$$(AT' \mid \square_1) \rightsquigarrow_s (\text{skip} \mid \square_1)$$

Taking $AT = \text{skip}$, this case has been shown to hold.

Case 10.3. Empty Block [B2a] Consider a program containing a single block statement, whose body is a single skip statement. Let AP be the program $\text{begin Bn skip } I \text{ skip } I \text{ end (pa,A)}$ (with the original block body also being skip as expected), and \square be the tuple of initial program state environments $(\sigma, \gamma, \mu, \beta, \delta)$. Assume the uniform execution via [B2a]

$$(\text{begin Bn skip } I \text{ skip } I \text{ end (pa,A)} \mid \square) \rightarrow_s (\text{skip } I' \mid \square)$$

with length 1 and no effect on the program state.

We need to show that there exists an execution

$$(inv^+(\mathbf{AP}) \mid \square'_1) \overset{\circ}{\rightsquigarrow}^* (\mathbf{AT}' \mid \square_1) \rightsquigarrow_s^* (\mathbf{AT} \mid \square_1)$$

for some statements \mathbf{AT}' , \mathbf{AT} , and states \square'_1 , \square_1 such that $\square'_1 \approx \square'$ and $\square_1 \approx \square$. By 5.31 in Figure 5.7 of Chapter 5, we have $inv^+(\mathbf{AP}) = \mathbf{begin} \ \mathbf{Bn} \ \mathbf{skip} \ \mathbf{I} \ \mathbf{skip} \ \mathbf{I} \ \mathbf{end} \ (\mathbf{pa}, \mathbf{A})$ (ignoring the abort).

With no further execution, we can take $\mathbf{AT}' = inv^+(\mathbf{AP})$ and $\square_1 = \square'_1$. Application of [B2r] closes the inverted block statement via

$$(\mathbf{AT}' \mid \square_1) \rightsquigarrow_s (\mathbf{skip} \mid \square_1)$$

Since $\mathbf{AT} = \mathbf{skip}$, this case holds.

Each base case has been shown to valid, meaning Part 1 of Lemma 11 (Program Property) holds for all executions of length 1. We now consider all inductive cases. Assume the Program Property holds for all programs \mathbf{AR} and program states \square^* such that the execution $(\mathbf{AR} \mid \square^*) \rightarrow_s (\mathbf{AR}' \mid \square^*) \overset{\circ}{\rightarrow}^* (\mathbf{skip} \ \mathbf{I} \mid \square'^*)$ has length k (where $k \geq 1$). Now assume that the execution $(\mathbf{AP} \mid \square) \rightarrow_s (\mathbf{AP}' \mid \square) \overset{\circ}{\rightarrow}^* (\mathbf{skip} \ \mathbf{I} \mid \square')$ has length l such that $l > k$.

Each inductive case is now shown. This consists of executions of complete programs beginning with skip steps. These are skip steps from within a block statement (Case 10.4), and as a result of sequential composition (Case 10.5) and parallel composition (Cases 10.6–10.7). We note that there are no cases for steps via a conditional branch or loop body as these cannot be the first step of a complete program. Each require at least one previous step to open the conditional or the loop. There is also no case for the skip step [W6a] with the same reasoning.

Case 10.4. (Block [B1a]) Consider a block statement containing a body that begins with at least one skip step. Let \mathbf{AP} be $\mathbf{begin} \ \mathbf{Bn} \ \mathbf{AQ} \ \mathbf{AQ} \ \mathbf{end}$ and \square be the initial program state. If \mathbf{AQ} contains only skip statements and execution of those mean the block itself will close, then this is a version of the (base) Case 10.3. Therefore assume \mathbf{AQ} contains more than just skip steps. Assume the following uniform execution

$$(\mathbf{begin} \ \mathbf{Bn} \ \mathbf{AQ} \ \mathbf{AQ} \ \mathbf{end} \mid \square) \rightarrow_s^* (\mathbf{begin} \ \mathbf{Bn} \ \mathbf{AQ}' \ \mathbf{AQ} \ \mathbf{end} \mid \square) \overset{\circ}{\rightarrow}^* (\mathbf{skip} \mid \square')$$

exists for some program \mathbf{AQ}' and state \square' . Crucially all available skip steps are performed, meaning the first step of the remaining execution must be via an identifier

rule. We note that this execution can be rewritten to highlight the point at which the body finishes, namely as

$$\begin{aligned} & (\text{begin Bn AQ AQ end} \mid \square) \rightarrow_s^* (\text{begin Bn AQ' AQ end} \mid \square) \\ & \xrightarrow{\circ}^* (\text{begin Bn skip I AQ end} \mid \square') \rightarrow_s (\text{skip} \mid \square') \end{aligned}$$

where the final transition is via [B2a].

We need to show that there exists an execution

$$(\text{inv}^+(\text{AP}) \mid \square'_1) \xrightarrow{\circ}^* (\text{AT}' \mid \square_1) \rightsquigarrow_s^* (\text{AT} \mid \square_1)$$

for some statements AT' , AT , and program states \square'_1 , \square_1 such that $\square'_1 \approx \square'$ and $\square_1 \approx \square$. By 5.31 in Figure 5.7 of Chapter 5, we have $\text{inv}^+(\text{AP}) = \text{begin Bn inv(AQ) inv(AQ) end}$.

From our rewritten assumed execution, repeated use of the rule [B1a] (from conclusion to premises), we have the uniform execution

$$(\text{AQ} \mid \square) \rightarrow_s^* (\text{AQ}' \mid \square) \xrightarrow{\circ}^* (\text{skip I} \mid \square')$$

which is guaranteed to be shorter (as the block must close). By the induction hypothesis of Part 1 of Lemma 11 (Program Property), we get

$$(\text{inv}^+(\text{AQ}) \mid \square''_1) \xrightarrow{\circ}^* (\text{AR}' \mid \square_1) \rightsquigarrow_s^* (\text{skip I} \mid \square_1)$$

for some program AR' and states \square''_1 and \square_1 , which will reach skip as it must be a full program. We note that since AQ must be a complete program, $\text{inv}^+(\text{AQ}) = \text{inv(AQ)}$. Then using this substitution and repeated use of the corresponding inverse rule [B1r], we get

$$\begin{aligned} & (\text{begin Bn inv(AQ) inv(AQ) end} \mid \square'_1) \xrightarrow{\circ}^* (\text{begin Bn AR' inv(AQ) end} \mid \square_1) \\ & \rightsquigarrow_s^* (\text{begin Bn skip I inv(AQ) end} \mid \square_1) \end{aligned}$$

Finally, since AP must be a complete program, the inverse execution is required to reach skip. A single application of the skip step [B2a] will close this block and allow the whole execution to reach skip. Therefore we have the execution

$$\begin{aligned} & (\text{begin Bn inv(AQ) inv(AQ) end} \mid \square'_1) \xrightarrow{\circ}^* (\text{begin Bn AR' inv(AQ) end} \mid \square_1) \\ & \rightsquigarrow_s^* (\text{begin Bn skip I inv(AQ) end} \mid \square_1) \rightsquigarrow_s (\text{skip} \mid \square_1) \end{aligned}$$

that matches our desired execution and so shows this case to be valid.

Case 10.5. (Sequential Composition [S1a]) Consider sequential composition. Let \mathbf{AS} be a statement, \mathbf{AP} be a program, and \square be the initial program state environments $(\sigma, \gamma, \mu, \beta, \delta)$. Assume a program of the form $\mathbf{AS}; \mathbf{AP}$, and the execution via the initial skip rule [S1a]

$$(\mathbf{AS}; \mathbf{AP} \mid \square) \rightarrow_s^* (\mathbf{AS}'; \mathbf{AP} \mid \square) \xrightarrow{\circ}^* (\mathbf{skip} \ \mathbf{I} \mid \square')$$

for some statement \mathbf{AS}' , a pair \mathbf{I} and program state \square' . As in previous cases, note that all available skip steps are performed and the first step of the remaining execution must be via a identifier rule.

We need to show that there exists an execution

$$(inv^+(\mathbf{AS}; \mathbf{AP}) \mid \square'_1) \rightsquigarrow^* (\mathbf{AT}' \mid \square_1) \rightsquigarrow_s^* (\mathbf{AT} \mid \square_1)$$

for some programs \mathbf{AT}' , \mathbf{AT} such that \mathbf{AT} is skip or a skip equivalent, and program states \square'_1 , \square_1 such that $\square'_1 \approx \square'$ and $\square_1 \approx \square$. By 5.20 in Figure 5.7 of Chapter 5, we have that $inv^+(\mathbf{AS}; \mathbf{AP}) = inv(\mathbf{AP}); inv^+(\mathbf{AS})$.

Our assumed execution can be rewritten to highlight the point at which \mathbf{AS} concludes, namely as

$$\begin{aligned} (\mathbf{AS}; \mathbf{AP} \mid \square) &\rightarrow_s^* (\mathbf{AS}'; \mathbf{AP} \mid \square) \xrightarrow{\circ}^* (\mathbf{skip} \ \mathbf{I}'; \mathbf{AP} \mid \square'') \\ &\rightarrow_s (\mathbf{AP} \mid \square'') \xrightarrow{\circ}^* (\mathbf{skip} \ \mathbf{I} \mid \square') \end{aligned}$$

for program state \square'' . From this rewritten version of the execution, the application of the rule [S1a] repeatedly (from conclusion to premises), we can get the execution

$$(\mathbf{AS} \mid \square) \rightarrow_s^* (\mathbf{AS}' \mid \square) \xrightarrow{\circ}^* (\mathbf{skip} \ \mathbf{I}' \mid \square'')$$

which concerns only the statement \mathbf{AS} . Therefore this is guaranteed to be shorter than our original (as \mathbf{AP} must require at least one step). With a single statement equivalent to a program containing only that statement, the induction hypothesis of Part 1 of the Program Property (Lemma 11) on this shorter execution gives

$$(inv^+(\mathbf{AS}) \mid \square''_1) \rightsquigarrow^* (\mathbf{AR}' \mid \square_1) \rightsquigarrow_s^* (\mathbf{AR} \mid \square_1) \tag{6.5}$$

for programs \mathbf{AR}' , \mathbf{AR} such that \mathbf{AR} is a valid skip equivalent of some program, and program states \square''_1 , \square_1 such that $\square''_1 \approx \square''$ and $\square_1 \approx \square$.

Returning to our rewritten assumed execution above, the uniform execution $(\mathbf{AP} \mid \square'') \xrightarrow{\circ}^* (\mathbf{skip} \ \mathbf{I} \mid \square')$ is guaranteed to be both shorter than our original and to be a complete program (since it was sequentially composed with \mathbf{AS}). In most cases this will begin with an identifier step, however it could also begin with at least one skip

step. Application of the induction hypothesis of Lemma 11 (Part 1 if the execution begins with a skip step, or Part 2 if the execution begins with an identifier step) on this gives

$$(inv^+(\mathbf{AP}) \mid \square'_1) \xrightarrow{\circ}^* (\mathbf{skip} \ I_1 \mid \square''_1)$$

for program state \square''_1 such that $\square'_1 \approx \square''_1$. Since \mathbf{AP} must be a complete statement, we note that $inv^+(\mathbf{AP}) = inv(\mathbf{AP})$. With this substitution, use of the rule [S1r] repeatedly (from premises to conclusion) gives us

$$(inv(\mathbf{AP}); inv^+(\mathbf{AS}) \mid \square'_1) \xrightarrow{\circ}^* (\mathbf{skip} \ I_1; inv^+(\mathbf{AS}) \mid \square''_1) \quad (6.6)$$

A single application of the rule [S2r] will remove the skip, giving the execution

$$(\mathbf{skip} \ I_1; inv^+(\mathbf{AS}) \mid \square''_1) \rightsquigarrow_s (inv^+(\mathbf{AS}) \mid \square''_1) \quad (6.7)$$

At this point, we have three equations that can easily be composed. We therefore combine 6.6, 6.7 and 6.5 in that order, giving the execution

$$\begin{aligned} & (inv(\mathbf{AP}); inv^+(\mathbf{AS}) \mid \square'_1) \xrightarrow{\circ}^* (\mathbf{skip} \ I_1; inv^+(\mathbf{AS}) \mid \square''_1) \rightsquigarrow_s (inv^+(\mathbf{AS}) \mid \square''_1) \\ & \xrightarrow{\circ}^* (\mathbf{AR}' \mid \square_1) \rightsquigarrow_s^* (\mathbf{AR} \mid \square_1) \end{aligned}$$

Taking $\mathbf{AT}'' = \mathbf{AR}''$, $\mathbf{AT}' = \mathbf{AR}'$ and $\mathbf{AT} = \mathbf{AR}$ (\mathbf{AR} is either skip or a skip equivalent by the induction hypothesis), we have our desired execution, meaning the case holds.

Case 10.6. (Parallel Composition [P1a]) Consider parallel composition. Let \mathbf{AS} be a statement (of the syntax shown in Figure 6.3), \mathbf{AP} and \mathbf{AQ} be programs, and \square be the initial program state. Assume a program of the form $(\mathbf{AS}; \mathbf{AP}) \ \mathbf{par} \ \mathbf{AQ}$, where \mathbf{AQ} begins with an identifier step (this could be extended to allow skip steps from \mathbf{AQ} which would have to be ordered in the fixed manner of after all skip steps from \mathbf{AS}). The matching case of a program of the form $\mathbf{AQ} \ \mathbf{par} \ (\mathbf{AS}; \mathbf{AP})$ is considered in Case 10.7. Further, any number of nested parallel statements could have been used, all of which can be reduced to a program of the form above via the use of [P1a]/[P2a]. Now assume an execution with all of the initial skip steps via [P1a]

$$(((\mathbf{AS}; \mathbf{AP}) \ \mathbf{par} \ \mathbf{AQ}) \mid \square) \rightarrow_s^* (((\mathbf{AS}'; \mathbf{AP}) \ \mathbf{par} \ \mathbf{AQ}) \mid \square) \xrightarrow{\circ}^* (\mathbf{skip} \ I \mid \square')$$

for some statement \mathbf{AS}' and program state \square' . Consider the initial sequence of skip steps, where all available skip steps are performed. This means that the first step of the renaming execution must be via an identifier step. Each skip step transition has an inference tree associated with it that proves the transition to be valid. Each of these inference trees has a leaf, which in each case is an instance of one of the following three rules.

1. **Rule [B2a]**: the closing of a block meaning the block body is empty. **AS** contains **begin Bn skip I₁ end** (potentially nested within constructs), and **AS'** contains **skip** (with same nesting).
2. **Rule [S2a]**: the removal of a skip statement. **AS** contains **skip I₁; AR** (with nesting), and **AS'** contains **AR** (with same nesting).
3. **Rule [P3a]**: an empty parallel statement. **AS** contains **skip I₁ par skip I₂; AR** (potentially nested), and **AS' = skip; AR** (with the same nesting).

We need to show that there exists an execution

$$(inv^+((AS;AP) \text{ par } AQ) \mid \square'_1) \overset{\circ}{\rightsquigarrow}^* (AT' \mid \square_1) \rightsquigarrow_s^* (AT \mid \square_1)$$

for some programs **AT'**, **AT** such that **AT** is skip or a skip equivalent, and program states \square'_1, \square_1 such that $\square'_1 \approx \square'$ and $\square_1 \approx \square$. By 5.21 and 5.20 in Figure 5.7 of Chapter 5, we have that $inv^+((AS;AP) \text{ par } AQ) = (inv(AP); inv^+(AS) \text{ par } inv(AQ))$.

From our assumed transition sequence, the execution $((AS';AP) \text{ par } AQ \mid \square) \overset{\circ}{\rightarrow}^* (\text{skip I} \mid \square')$ is guaranteed to both be shorter than our original and to begin with an identifier step (since the overall execution is uniform meaning all skip steps are performed in the previous transition). Application of the induction hypothesis of Part 2 of Lemma 11 on this shorter uniform execution gives us

$$(inv(AP); inv^+(AS') \text{ par } inv(AQ) \mid \square'_1) \overset{\circ}{\rightsquigarrow}^* (AT'' \mid \square_1)$$

by a sequence of rule applications SR, for some program **AT''** and program states \square'_1, \square_1 such that $\square'_1 \approx \square'$ and $\square_1 \approx \square$.

With **AP** guaranteed to be a complete program, inversion will reach **skip I₄** for some **I₄**. **AQ** also must be a complete program meaning inversion will reach **skip I₅**. **AS'** will be a partial program (since the beginning skip steps have been performed), meaning the inversion of this will reach **AS₁** (either skip or a skip equivalent). Therefore **AT'' = AS₁ par skip I₅**.

Now compare the programs $inv(AP); inv^+(AS) \text{ par } inv(AQ)$ and $inv(AP); inv^+(AS') \text{ par } inv(AQ)$. Since both executions begin with the same prefix, by the same sequence of rule applications SR we obtain

$$(inv(AP); inv^+(AS) \text{ par } inv(AQ) \mid \square'_1) \overset{\circ}{\rightsquigarrow}^* (AT' \mid \square_1)$$

for a program **AT'**. With the same reasoning as above, we have that **AT' = AS₂ par skip I₅**. The format of **AS₂** is identical to **AS₁** but with further execution of skip steps only available. All of the constructs that were closed (or removed) during the

initial skip steps of the forward execution will appear in inverted form within the inverse program. In order to complete the inverse execution, all of these constructs must also be closed (or removed) here. The skip steps that will be required will depend on the types of construct that began the forward execution. We therefore return to our three cases of skip step and consider each in turn.

1. **Rule [B2r]**: The initially empty block will be here in inverted form. Therefore the skip rule [B2r] will close this block to skip.
2. **Rule [S2r]**: A hard-coded skip statement will appear in the inverted program, with the rule [S2r] used to remove it.
3. **Rule [P3r]**: A hard-coded empty parallel statement appears identically in the inverted program. The inverse skip step [P3r] is used to close it.

In all cases, we have that there exists the execution

$$(\mathbf{AT}' \mid \square_1) \rightsquigarrow_s^* (\mathbf{AT} \mid \square_1)$$

such that $\mathbf{AT} = \text{skip}$.

At this point, recall Example 18 from Chapter 5 showing non-matching skip steps from a forward and reverse execution. Since all skip steps cannot alter the program state, any mismatch here is not a problem. Therefore we have shown the desired execution and this case to hold.

Case 10.7. (Parallel Composition [P2a]) This case follows closely to Case 10.6, using the rule [P2a] in place of [P1a] to represent the situation where the execution begins with skip rules from the right hand side.

With all inductive cases shown to be valid, we have therefore completed the proof of Part 1 of Lemma 11.

6.6.2 Proof of Part 2

We now consider all executions beginning with an identifier step, of the form $(\mathbf{AP} \mid \square) \xrightarrow{m} (\mathbf{AP}' \mid \square) \rightarrow_s^* (\mathbf{AP}'' \mid \square) \xrightarrow{\circ}^* (\text{skip } \mathbf{I} \mid \square')$. With no executions of length 0, we consider our base cases to be any execution of length 1. The two base cases include a program containing a single assignment and a program containing a single loop that performs zero iterations. The proof of each is omitted as they follow correspondingly to Case 9.1 and Case 9.2 of Lemma 10 from page 111 (with **AS** replaced by **AP**).

All base cases are therefore valid, meaning we now consider all inductive cases. Assume the Program Property holds for all programs **AR** and program states \square^* such that the execution $(\mathbf{AR} \mid \square^*) \xrightarrow{m} (\mathbf{AR}' \mid \square^*) \rightarrow_s^* (\mathbf{AR}'' \mid \square^*) \xrightarrow{\circ}^* (\text{skip } \mathbf{I} \mid \square'^*)$ has

length k (where $k \geq 1$). Now assume that the execution $(\mathbf{AR} \mid \square) \xrightarrow{m} (\mathbf{AR}' \mid \square) \rightarrow_s^* (\mathbf{AR}'' \mid \square) \xrightarrow{\circ}^* (\text{skip } \mathbf{I} \mid \square')$ has length l such that $l > k$.

Each inductive case is now considered. We note that the execution of a program containing a single statement is equivalent to an execution of that single statement. This means all such executions have been considered within the proof of Lemma 10 and so are omitted here. We therefore consider the remaining cases for sequential composition (Case 10.8) and for parallel composition (Case 10.9–10.10).

Case 10.8. (Sequential Composition [S1a]) Consider sequential composition. Let \mathbf{AS} be a statement, \mathbf{AP} be a program, and \square be the tuple of initial program state environments $(\sigma, \gamma, \mu, \beta, \delta)$. Assume a program of the form $\mathbf{AS}; \mathbf{AP}$ (sequential composition), and the uniform execution with the first transition via the rule [S1a]

$$(\mathbf{AS}; \mathbf{AP} \mid \square) \xrightarrow{m} (\mathbf{AS}'; \mathbf{AP} \mid \square'') \rightarrow_s^* (\mathbf{AS}''; \mathbf{AP} \mid \square'') \xrightarrow{\circ}^* (\text{skip } \mathbf{I} \mid \square')$$

for some statements \mathbf{AS}' and \mathbf{AS}'' , and program states \square'' and \square' .

We need to show that there exists an execution

$$(\text{inv}^+(\mathbf{AS}; \mathbf{AP}) \mid \square'_1) \rightsquigarrow^* (\mathbf{AT}'' \mid \square''_1) \xrightarrow{m} (\mathbf{AT}' \mid \square_1) \rightsquigarrow_s^* (\mathbf{AT} \mid \square_1)$$

for some statements \mathbf{AT}'' , \mathbf{AT}' , \mathbf{AT} , and program states \square'_1 , \square''_1 , \square_1 such that $\square'_1 \approx \square'$, $\square''_1 \approx \square''$ and $\square_1 \approx \square$. By 5.20 in Figure 5.7 of Chapter 5, we have that $\text{inv}^+(\mathbf{AS}; \mathbf{AP}) = \text{inv}(\mathbf{AP}); \text{inv}^+(\mathbf{AS})$.

Our assumed execution can be rewritten to show the point at which \mathbf{AS} has executed completely, namely as

$$\begin{aligned} (\mathbf{AS}; \mathbf{AP} \mid \square) &\xrightarrow{m} (\mathbf{AS}'; \mathbf{AP} \mid \square'') \rightarrow_s^* (\mathbf{AS}''; \mathbf{AP} \mid \square'') \xrightarrow{\circ}^* (\text{skip } \mathbf{I}'; \mathbf{AP} \mid \square''') \\ &\rightarrow_s (\mathbf{AP} \mid \square''') \xrightarrow{\circ}^* (\text{skip } \mathbf{I} \mid \square') \end{aligned}$$

for some program state \square''' . From this rewritten version of the execution, the application of the rule [S1a] repeatedly (from conclusion to premises), we can get the execution

$$(\mathbf{AS} \mid \square) \xrightarrow{m} (\mathbf{AS}' \mid \square'') \rightarrow_s^* (\mathbf{AS}'' \mid \square'') \xrightarrow{\circ}^* (\text{skip } \mathbf{I}' \mid \square''')$$

Since this concerns only \mathbf{AS} , it is guaranteed to be shorter than the original execution as \mathbf{AP} takes at least one step of execution. This allows application of the induction hypothesis of the Statement Property (Lemma 10) on this shorter execution, giving us

$$(\text{inv}^+(\mathbf{AS}) \mid \square'''_1) \rightsquigarrow^* (\mathbf{AR}'' \mid \square''_1) \xrightarrow{m} (\mathbf{AR}' \mid \square_1) \rightsquigarrow_s^* (\mathbf{AR} \mid \square_1) \quad (6.8)$$

for some programs \mathbf{AR}'' , \mathbf{AR}' and \mathbf{AR} such that \mathbf{AR} is either **skip** or a skip equivalent of some program, and some program states \square_1'' such that $\square_1'' \approx \square''$, and \square_1 such that $\square_1 \approx \square$.

Returning to our rewritten assumed execution above, the uniform execution $(\mathbf{AP} \mid \square''') \xrightarrow{\circ^*} (\mathbf{skip} \ I \mid \square')$ is shorter than our original. This program will begin with an identifier step in the majority of cases, with a chance for it to begin with skip steps (hard coded skips, empty blocks etc.). This means that application of the induction hypothesis of Lemma 11 (Part 1 if the execution begins with a skip step, or Part 2 if the execution begins with an identifier step) on this shorter execution gives us

$$(inv^+(\mathbf{AP}) \mid \square_1') \xrightarrow{\circ^*} (\mathbf{skip} \ I_1 \mid \square_1''')$$

for some program state \square_1''' such that $\square_1''' \approx \square'''$. Since \mathbf{AP} must be a complete program (as it is sequentially composed with \mathbf{AS}), we note that $inv^+(\mathbf{AP}) = inv(\mathbf{AP})$. Using this substitution, application of the rule [S1r] repeatedly (from premises to conclusion) gives

$$(inv(\mathbf{AP}); inv^+(\mathbf{AS}) \mid \square_1') \xrightarrow{\circ^*} (\mathbf{skip} \ I_1; inv^+(\mathbf{AS}) \mid \square_1''') \quad (6.9)$$

A single applicable rule is available at this point, namely [S2r] to remove the skip. Since this is a skip rule and so does not affect the program state in any way (see Lemma 1 of Chapter 3), we have the execution

$$(\mathbf{skip} \ I_1; inv^+(\mathbf{AS}) \mid \square_1''') \rightsquigarrow_s (inv^+(\mathbf{AS}) \mid \square_1''') \quad (6.10)$$

At this point, the three equations 6.9, 6.10 and 6.8 can be composed in that order. This gives

$$\begin{aligned} (inv(\mathbf{AP}); inv^+(\mathbf{AS}) \mid \square_1') &\xrightarrow{\circ^*} (\mathbf{skip} \ I_1; inv^+(\mathbf{AS}) \mid \square_1''') \rightsquigarrow_s (inv^+(\mathbf{AS}) \mid \square_1''') \\ &\xrightarrow{\circ^*} (\mathbf{AR}'' \mid \square_1'') \xrightarrow{m} (\mathbf{AR}' \mid \square_1) \rightsquigarrow_s^* (\mathbf{AR} \mid \square_1) \end{aligned}$$

Taking $\mathbf{AT}'' = \mathbf{AR}''$, $\mathbf{AT}' = \mathbf{AR}'$ and $\mathbf{AT} = \mathbf{AR}$ (such that \mathbf{AR} is either skip or a skip equivalent by induction hypothesis), we have shown this case to hold.

Case 10.9. (Parallel Composition [P1a]) Consider parallel composition. Let \mathbf{AS} be a statement, \mathbf{AP} and \mathbf{AQ} be programs, and \square be the tuple of initial program state environments $(\sigma, \gamma, \mu, \beta, \delta)$. Assume a program of the form $(\mathbf{AS}; \mathbf{AP}) \mathbf{par} \ \mathbf{AQ}$. We note that the corresponding form $\mathbf{AQ} \mathbf{par} \ (\mathbf{AS}; \mathbf{AP})$ follows accordingly (see Case 10.10) and that any number of nested parallel statements could be used,

with matching uses of [P1a]/[P2a] reducing this into a program of the above form. Assume the following uniform execution with the first transition via [P1a]

$$\begin{aligned} &(((AS;AP) \text{ par } AQ) \mid \square) \xrightarrow{m} (((AS'';AP) \text{ par } AQ) \mid \square'') \\ &\rightarrow_s^* (((AS';AP) \text{ par } AQ) \mid \square'') \xrightarrow{\circ}^* (\text{skip } I \mid \square') \end{aligned}$$

for statements AS'' and AS' , and program states \square'' and \square' .

We need to show that there exists an execution

$$(inv^+(AS;AP \text{ par } AQ) \mid \square'_1) \xrightarrow{\circ}^* (AT'' \mid \square''_1) \xrightarrow{m} (AT' \mid \square_1) \xrightarrow{\sim}_s^* (AT \mid \square_1)$$

for some statements AT'' , AT' , AT , and program states \square'_1 , \square''_1 , \square_1 such that $\square'_1 \approx \square'$, $\square''_1 \approx \square''$ and $\square_1 \approx \square$. From 5.20 and 5.21 in Figure 5.7 of Chapter 5, we have that $inv^+(AS;AP \text{ par } AQ) = inv(AP);inv^+(AS) \text{ par } inv^+(AQ)$.

From our assumed transition sequence, the execution $((AS';AP) \text{ par } AQ) \mid \square'' \xrightarrow{\circ}^* (\text{skip } I \mid \square')$ must be shorter than our original and must begin with an identifier step (since the overall execution is uniform meaning all available skip steps must be performed in the previous transition). This means application of the induction hypothesis of Part 2 of Lemma 11 on this shorter execution gives us

$$(inv(AP);inv^+(AS') \text{ par } inv^+(AQ) \mid \square'_1) \xrightarrow{\circ}^* (AT''' \mid \square''_1)$$

by a sequence of rules SR, for some program AT''' and program states \square'_1 , \square_1 such that $\square'_1 \approx \square'$ and $\square_1 \approx \square$. From the definition of [P1a], the first transition is via an underlying identifier rule [R]. Each case of [R] is considered with the proof of the Statement Property (Lemma 10), with the format of AS , AS' and AS'' (which we must consider to be within the necessary context) shown along with \square'' .

Since AT''' is returned via the induction hypothesis, we know its format. With AP guaranteed to be a complete program, inversion will reach $\text{skip } I_4$ for some I_4 . AQ is either a complete or a partial program meaning inversion will either reach $\text{skip } I_5$ or a skip equivalent program AQ_1 . Therefore $AT''' = inv^+(AS') \text{ par } \text{skip } I_5$ or $AT''' = inv^+(AS') \text{ par } AQ_1$ where AQ_1 is a skip equivalent.

Now compare the programs $inv(AP);inv^+(AS) \text{ par } inv(AQ)$ and $inv(AP);inv^+(AS') \text{ par } inv(AQ)$. Since both executions begin with the same prefix, by the same sequence of rule applications SR we obtain

$$(inv(AP);inv^+(AS') \text{ par } inv(AQ) \mid \square'_1) \xrightarrow{\circ}^* (AT'' \mid \square_1)$$

for a program AT'' . Drawing conclusions as above, we have that $AT'' = AS_2 \text{ par } \text{skip } I_5$ or $AT'' = AS_2 \text{ par } AQ_1$ for a statement AS_2 , such that AS_2 is equal to AS_1 but

with further execution allowed. The format of \mathbf{AS}_2 depends on the underlying rule $[R]$, with each case having been considered in the proof of the Statement Property (Lemma 10). From this, each case shows the format of \mathbf{AS}_2 (from which we know the execution must continue), and details that the only available next step of execution will be via the corresponding inverse identifier rule. From each case, we can see that the execution

$$(\mathbf{AT}'' \mid \square_1'') \xrightarrow{m} (\mathbf{AT}' \mid \square_1) \rightsquigarrow_s^* (\mathbf{AT} \mid \square_1)$$

exists for some programs \mathbf{AT}' and \mathbf{AT} (without the context of the parallel statement), and program state \square_1 such that $\square_1 \approx \square$, as required. Therefore this case holds.

Case 10.10. (Parallel Composition [P2a]) Consider an identifier step from the right hand side of a parallel statement. This follows correspondingly to Case 10.9 and so is omitted.

We have now shown all inductive cases to be valid for Part 2 of Lemma 11. Together with the proof of Part 1, we have therefore proved Lemma 11 to be valid, as required. \square

6.7 Conclusion

In this section, we have proved two key properties relating to our approach. The first demonstrates that annotation does not change the behaviour of the original program with respect to the program state, and that it can populate the auxiliary store. The second shows that given the correct final program state and final auxiliary store, reverse execution correctly restores the state to as it prior to the forward execution, and that all reversal information is used hence our approach is garbage free.

Chapter 7

Ripple: Simulation Tool and Performance Evaluation

In this chapter we introduce our simulation tool named Ripple (standing for Reversing an Imperative Parallel Programming Language). Developed to aid testing and validating of our approach, Ripple takes a program written in our language and automatically produces the annotated and inverted versions. Execution in each direction (including traditional forward with no saving) can be simulated, with the program state and reversal information viewable throughout. Ripple is available on request to the author at <https://github.com/jameshoey/ripple-sim>.

7.1 Software Architecture of Ripple

Ripple is written in the commonly used and widely supported C++, using some of its many libraries. The object oriented nature means the architecture of Ripple consists of six key types of object, namely *FileIO*, *LinkedList*, *Statement*, *ExpTree*, *Parser* and *Simulator*. Each type of object is described in detail below, and contained within the simplified class diagram in Figure 7.1 (where some fields, functions and sub-classes are omitted). The *LinkedList*, *Statement* and *ExpTree* structures are then outlined, followed by a discussion of how key design choices presented in Chapter 3 are reflected into the implementation of Ripple. Finally we mention each of the main directories used by Ripple.

Objects

Ripple consists of instances of six key types of object. Each object, shown in the simplified class diagram in Figure 7.1, are outlined in turn.

- *FileIO* - used to access the file system, allowing a user of the simulator to load a pre-existing text file that contains a program into the parser object.

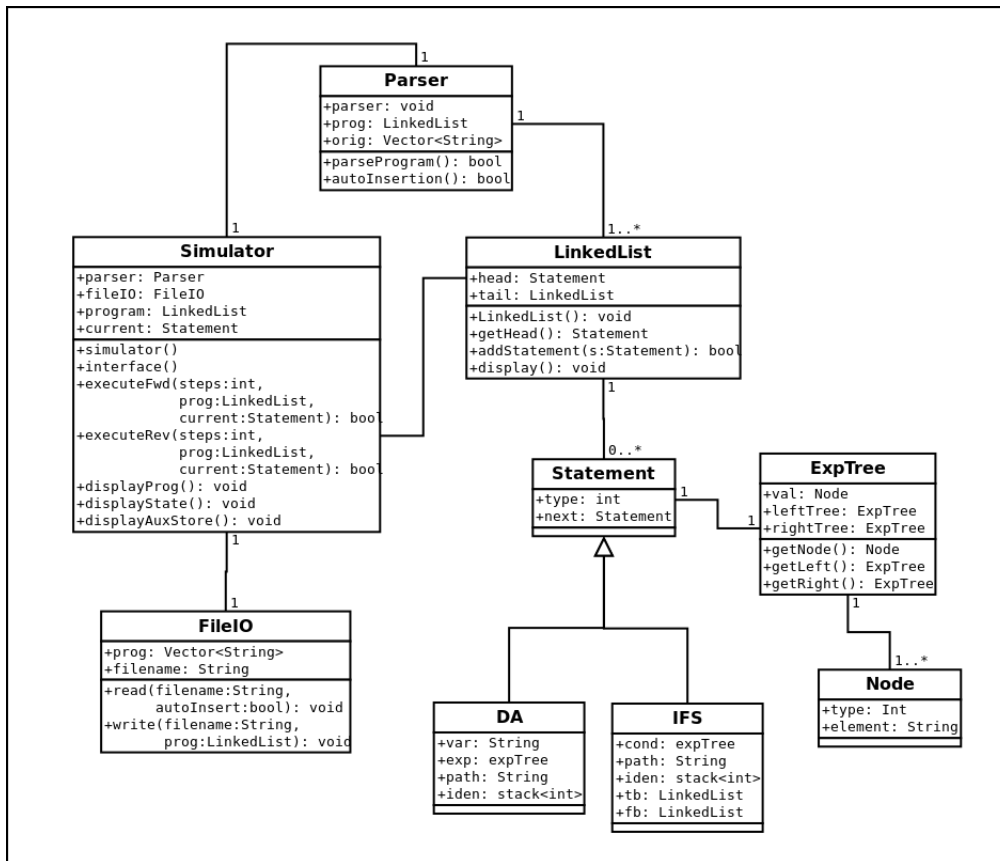


Figure 7.1: The class diagram representing each type of object and how it is linked to others, with only two sub-statement classes shown (namely DA and IFS).

All information written to files during the simulation is processed via a fileIO object, including the execution traces and histories.

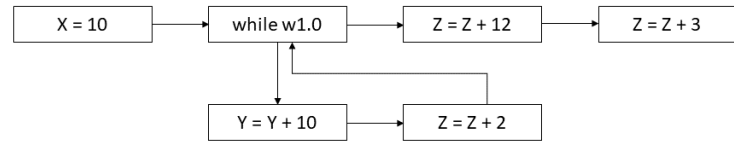
- *ExpTree* - a tree structure used to represent an arithmetic or boolean expressions. Each node (another type of object) is either a leaf (integer value or variable name) or a binary (arithmetic or boolean) operator with two children.
- *Statement* - used to represent a single statement. This is a super-class, with eleven sub-classes that each inherit from it. Each sub-class represents a specific statement, e.g. destructive assignment (DA) and conditional statement (IFS). The super-class allows generic use of a statement without specifying the type.
- *LinkedList* - an instance of the linked list object represents a program in the form expected by the simulator object. This is a redefined version (not the usual C++ linked list), in that each element is a Statement object. Each Statement object contains a pointer to the next (sequential composition), with the Statement sub-class named PAR containing a pointer to both the left and right side of a parallel (parallel composition).

```

1 X = 10;
2 while w1 (Y < 20) do
3   Y = Y + 10;
4   Z = Z + 2;
5 end;
6 Z = Z + 12;
7 X = Z + 3;

```

(a) Original source code



(b) Linked list representation

Figure 7.2: A program and its linked list representation

- *Parser* - an instance of this parses an original program into a linked list representation. This parser is implemented by ourselves, and may result in inefficiencies. In the future we plan to use a pre-defined parser.
- *Simulator* - a simulator object represents an instance of the tool. This instance is created by the user and automatically completes the setup. This includes creating instances of each of the previous objects, and providing the user interface allowing direct interaction (see below). This object controls execution in both directions, and is the only one instantiated by the user.

ExpTree, Statement and Linked List Structures

ExpTree structures represent a condition or expression. A node is considered a leaf if it contains only an integer value or variable name, and has no children. Any other node contains an operator (either arithmetic or logical) and two children. ExpTrees are evaluated in a depth first manner, and can be performed either atomically or non-atomically in Ripple. To aid testing, Ripple is also capable of evaluating an expression atomically, but independently to the rest of the statement. For example, an assignment would complete in two steps, the first atomically evaluating the expression and the second performing the assignment.

Each statement is represented as an instance of Statement. This generic superclass (parent) encapsulates all possible types, containing a variable **type** indicating the underlying type of statement and a pointer to the next statement (sequential composition). The value of this allows casting of a statement into the appropriate sub-class. Each sub-class represents a specific type of statement and contains the required variables. For example a conditional statement contains an ExpTree representation of its condition, and a pointer to each branch.

Each program is parsed into a LinkedList structure. Each element (an instance of Statement) captures the high level order of statements, as each contains a pointer to the next. Each complex statement that contains sub-programs also contains a

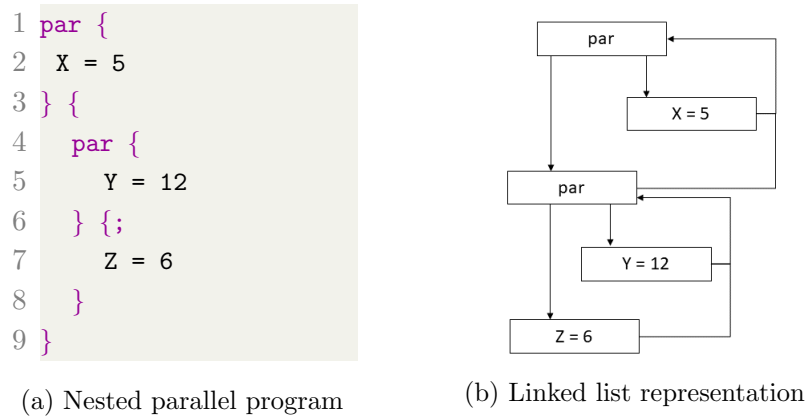


Figure 7.3: A parallel program and its linked list representation

pointer to the linked list representation of those (lower-level order of statements). An important part of parsing is named *linking*, where the end of nested LinkedList structures are linked back to the parent. The four cases in which this happens are:

- The end of a while loop body is linked back to the while loop statement (allowing for iteration). The example program in Figure 7.2(a) is parsed into the LinkedList structure in Figure 7.2(b), where the end of the loop body is linked back to the loop.
- The end of both the true and the false branch are linked back to the conditional statement (allowing the saving of any reversal information for the conditional to happen only after the execution of the branch).
- The end of a block body is linked back to the block statement (to allow a block to close explicitly).
- The end of each side of a parallel statement is linked back to the parallel statement (ensuring both sides have to be finished before further progress can be made as in the semantics). Figure 7.3(a) shows an example of nested parallelism, with Figure 7.3(b) showing the corresponding LinkedList structure.

Design Choices

Chapter 3 highlighted two key design choices of our approach. Firstly, all reversal information is saved into stacks. Ripple implements this, using a collection of stacks as the auxiliary store. All reversal information is pushed onto the top of the stack during the annotated execution, and removed from the top during inverse execution.

The second design choice is to keep the auxiliary store separate to the program state. This aids our proof of correctness (see Chapter 6) as an original and its annotated execution produce equivalent final program states. Therefore Ripple keeps


```

P ::= ε | S | P; P | P par P
S ::= skip | X = E | if B then P else Q end |
      while B do P end | begin BB end | call n
BB ::= DV; DP; P
DV ::= ε | var X = v pa; DV
DP ::= ε | proc Pn n is P end pa; DP
E ::= X | n | (E) | E Op E
B ::= T | F | ¬B | (B) | E == E | E > E | B ∧ B

```

Figure 7.4: Simplified syntax of programs supplied to the simulator

the auxiliary store separate, providing different commands to display separate environments.

Directories

The simulation tool requires a number of different directories that are used by an instance of the FileIO object. These include

- */examples* - the default directory where all programs will be read from (provided a valid filename is provided and that the corresponding file exists). This can be changed by the user at runtime (see below).
- */execution-traces* - the default directory where all execution traces will be saved. The forward and inverse trace file will be overwritten for each execution
- */execution-traces/perm* - the default directory where any execution trace file can be permanently saved to. This allows a specific trace file to be saved prior to it being overwritten via the simulator.

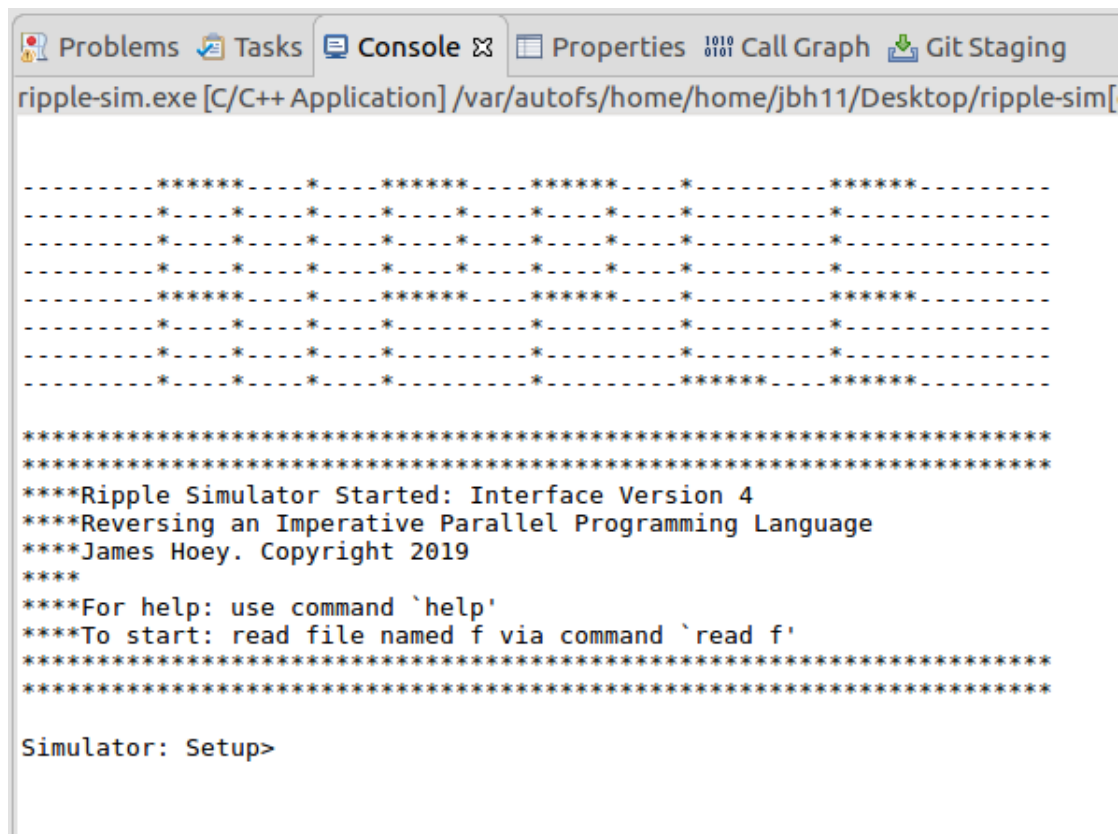
7.2 Key Functionality of Ripple

The following is a list of the major features of our simulator.

- *Automatic generation of precise syntax* - the burden on the user of the simulator is reduced as much as possible as the simulator can automatically add parts of the syntax. Paths are easily produced based on the source code and so can be automatically inserted. Additionally, all removal statements can be added automatically and so are not required in the code provided by the user (avoiding the risk of human error). Finally all construct identifiers can be inserted automatically. We therefore show a simplified version of the program

syntax that the simulator is capable of parsing correctly, shown in Figure 7.4 (with `runc` omitted as this only appears as a result of execution).

- *Automatic program state initialisation* - the initial program state is generated automatically. Since all global variables are assumed to exist prior to the execution, the source code is analysed to find all global variables used, each of which are then initialised to a fresh memory location and the value 0. This reduces the amount of memory needed to begin an execution as only the variables that are used exist. All other environments are instantiated, including the initially empty auxiliary store.
- *Random or user-defined interleaving* - the simulator is capable of two forms of parallel interleaving. Firstly, all decisions of which branch of a parallel to take can be made randomly at runtime via the simulator. This is achieved using random number generation and is used to simulate an arbitrary execution. Secondly, any interleaving decision can be taken by the user at runtime. In this mode each time an interleaving decision must be made, the execution is paused while the simulator waits for the user to decide from the available options. This feature is ideal for testing interleavings that occur rarely under random executions, or for recreating a specific interleaving that contained a software bug. User defined interleaving can also be switched on and off at runtime, allowing the user to only determine a part of an execution.
- *Full/Partial execution* - execution in both directions can either be performed completely or in a step-by-step manner. Full execution can be used to simulate an arbitrary execution, while step-by-step allows a more in depth execution. Crucially, the execution type is not fixed, allowing step-by-step execution of part of the program, before completing fully. Each execution is defined in terms of the possible types of statements. The behaviour of each statement is exactly as described in the operational semantics of Chapter 4.
- *Viewable program state* - the program state, including all intermediate program states during a step-by-step execution, can be viewed at runtime via the interface. The current program can also be displayed, where an arrow indicates the current position (multiple arrows will be present if currently within parallel composition). The value of all variables can be displayed with a reference to the corresponding memory location. The entire while and procedure environments can also be displayed, where each mapping currently stored is shown along with the construct identifier to which it applies
- *Viewable reversal information* - any information saved for the purposes of inversion can be viewed at runtime during executions in both directions. This



```

Problems Tasks Console Properties Call Graph Git Staging
ripple-sim.exe [C/C++ Application] /var/autofs/home/home/jbh11/Desktop/ripple-sim[

-----*****-----*-----*****-----*****-----*-----*****-----
-----*-----*-----*-----*-----*-----*-----*-----*-----
-----*-----*-----*-----*-----*-----*-----*-----*-----
-----*-----*-----*-----*-----*-----*-----*-----*-----
-----*****-----*-----*****-----*****-----*-----*****-----
-----*-----*-----*-----*-----*-----*-----*-----*-----
-----*-----*-----*-----*-----*-----*-----*-----*-----
-----*-----*-----*-----*-----*-----*-----*****-----*****-----

*****
*****
****Ripple Simulator Started: Interface Version 4
****Reversing an Imperative Parallel Programming Language
****James Hoey. Copyright 2019
****
****For help: use command `help'
****To start: read file named f via command `read f'
*****
*****

Simulator: Setup>

```

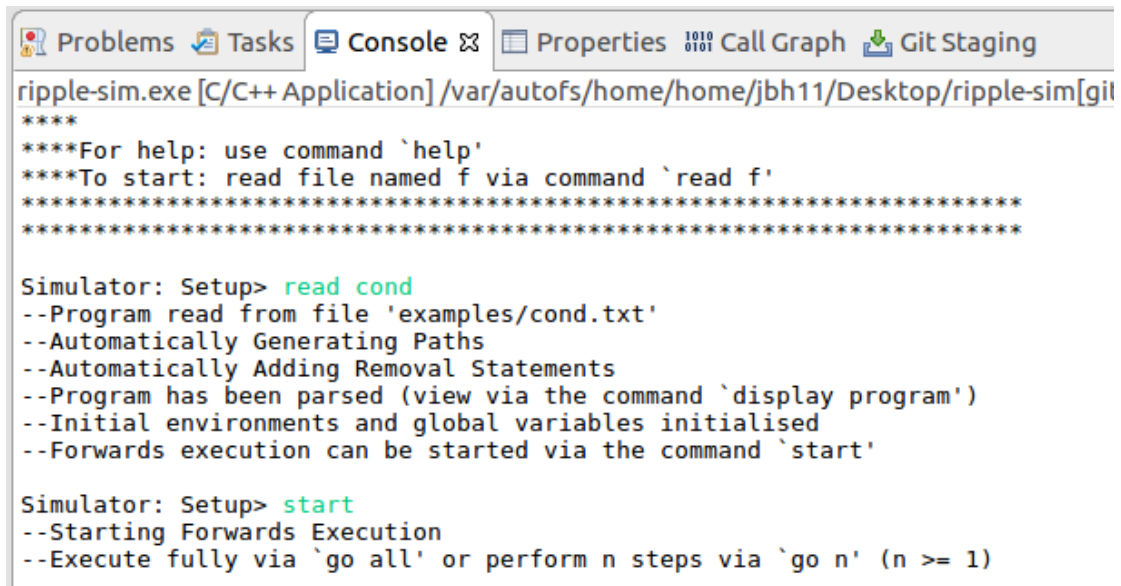
Figure 7.5: The welcome screen of the command line interface

information can also be used to see all values a variable has held during the execution, the number of iterations performed by a loop or the result of evaluating any conditional statement.

- *Execution tracing* - the simulator can maintain a trace throughout an execution in either direction. This is not used for reversal, but instead designed for testing and debugging purposes. A trace file captures an execution, including all interleaving decisions, allowing an execution that contains a bug to be repeated. The contents of example history files are shown in Chapter 8.

7.3 User Interface of Ripple

Our simulation tool comes with a command line interface, allowing the user to easily interact with the simulator. This command line interface opens immediately with the beginning of the simulator, with the welcome screen shown in Figure 7.5. For a full list of supported commands, open the simulator and open the help section via the command ‘help’. The interface begins in setup mode, where the tool is waiting for a file to read. We note here that there is a focus on user-friendliness, with a help function accessed via the command ‘help’ and the simulator hinting the typically



```

Problems Tasks Console Properties Call Graph Git Staging
ripple-sim.exe [C/C++ Application] /var/autofs/home/home/jbh11/Desktop/ripple-sim[gil
****
****For help: use command `help'
****To start: read file named f via command `read f'
*****
*****

Simulator: Setup> read cond
--Program read from file 'examples/cond.txt'
--Automatically Generating Paths
--Automatically Adding Removal Statements
--Program has been parsed (view via the command `display program')
--Initial environments and global variables initialised
--Forwards execution can be started via the command `start'

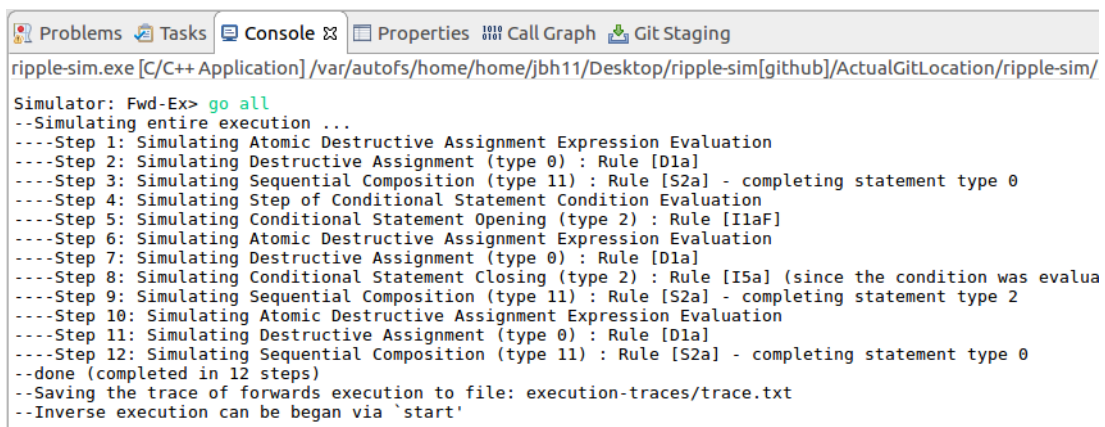
Simulator: Setup> start
--Starting Forwards Execution
--Execute fully via `go all' or perform n steps via `go n' (n >= 1)

```

Figure 7.6: Successful reading of a file

used next statement at the end of the previous. In Figure 7.5, the simulator states that the next command should be to read a file.

After reading a file (shown in Figure 7.6 using a file named `cond`), the simulator confirms the file has been read, all missing parts of the syntax have been automatically inserted and that parsing was successful. The start command begins the forward execution, while the command ‘statesaving off’ can be used to switch to an original (irreversible) execution, where no reversal information is saved and therefore the inverse execution will not be valid). The forward execution of a program (in this case the entire program using the command ‘go all’) is shown in Figure 7.7, while the process of inversion and subsequent execution of it is shown in Figure 7.8. The related command ‘go n ’ performs the next n steps of the current execution provided n steps are available, or as many of the n steps until the execution completes. There are a series of display commands used to view different parts of the program, the



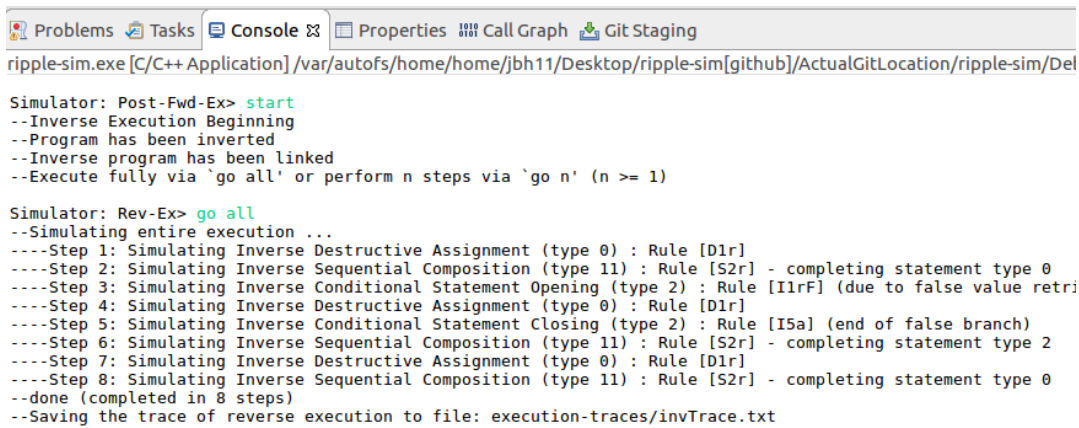
```

Problems Tasks Console Properties Call Graph Git Staging
ripple-sim.exe [C/C++ Application] /var/autofs/home/home/jbh11/Desktop/ripple-sim[github]/ActualGitLocation/ripple-sim/

Simulator: Fwd-Ex> go all
--Simulating entire execution ...
----Step 1: Simulating Atomic Destructive Assignment Expression Evaluation
----Step 2: Simulating Destructive Assignment (type 0) : Rule [D1a]
----Step 3: Simulating Sequential Composition (type 11) : Rule [S2a] - completing statement type 0
----Step 4: Simulating Step of Conditional Statement Condition Evaluation
----Step 5: Simulating Conditional Statement Opening (type 2) : Rule [I1aF]
----Step 6: Simulating Atomic Destructive Assignment Expression Evaluation
----Step 7: Simulating Destructive Assignment (type 0) : Rule [D1a]
----Step 8: Simulating Conditional Statement Closing (type 2) : Rule [I5a] (since the condition was evalua
----Step 9: Simulating Sequential Composition (type 11) : Rule [S2a] - completing statement type 2
----Step 10: Simulating Atomic Destructive Assignment Expression Evaluation
----Step 11: Simulating Destructive Assignment (type 0) : Rule [D1a]
----Step 12: Simulating Sequential Composition (type 11) : Rule [S2a] - completing statement type 0
--done (completed in 12 steps)
--Saving the trace of forwards execution to file: execution-traces/trace.txt
--Inverse execution can be began via `start'

```

Figure 7.7: Forward execution of a program



```

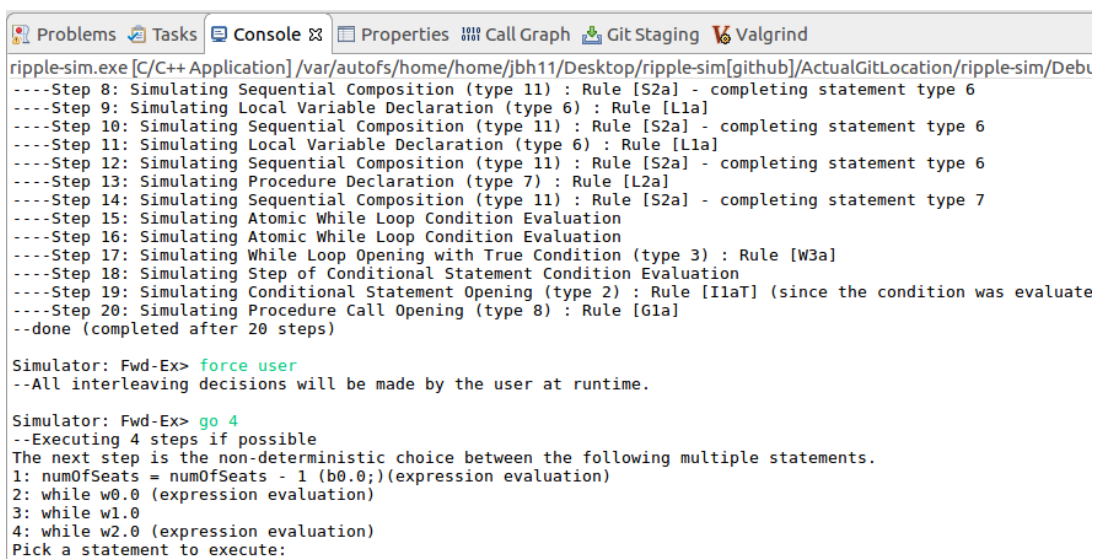
Problems Tasks Console Properties Call Graph Git Staging
ripple-sim.exe [C/C++ Application] /var/autofs/home/home/jbh11/Desktop/ripple-sim[github]/ActualGitLocation/ripple-sim/Del

Simulator: Post-Fwd-Ex> start
--Inverse Execution Beginning
--Program has been inverted
--Inverse program has been linked
--Execute fully via `go all` or perform n steps via `go n` (n >= 1)

Simulator: Rev-Ex> go all
--Simulating entire execution ...
----Step 1: Simulating Inverse Destructive Assignment (type 0) : Rule [D1r]
----Step 2: Simulating Inverse Sequential Composition (type 11) : Rule [S2r] - completing statement type 0
----Step 3: Simulating Inverse Conditional Statement Opening (type 2) : Rule [I1rF] (due to false value retri
----Step 4: Simulating Inverse Destructive Assignment (type 0) : Rule [D1r]
----Step 5: Simulating Inverse Conditional Statement Closing (type 2) : Rule [I5a] (end of false branch)
----Step 6: Simulating Inverse Sequential Composition (type 11) : Rule [S2r] - completing statement type 2
----Step 7: Simulating Inverse Destructive Assignment (type 0) : Rule [D1r]
----Step 8: Simulating Inverse Sequential Composition (type 11) : Rule [S2r] - completing statement type 0
--done (completed in 8 steps)
--Saving the trace of reverse execution to file: execution-traces/invTrace.txt

```

Figure 7.8: Reverse execution of a program



```

Problems Tasks Console Properties Call Graph Git Staging Valgrind
ripple-sim.exe [C/C++ Application] /var/autofs/home/home/jbh11/Desktop/ripple-sim[github]/ActualGitLocation/ripple-sim/Debu

----Step 8: Simulating Sequential Composition (type 11) : Rule [S2a] - completing statement type 6
----Step 9: Simulating Local Variable Declaration (type 6) : Rule [L1a]
----Step 10: Simulating Sequential Composition (type 11) : Rule [S2a] - completing statement type 6
----Step 11: Simulating Local Variable Declaration (type 6) : Rule [L1a]
----Step 12: Simulating Sequential Composition (type 11) : Rule [S2a] - completing statement type 6
----Step 13: Simulating Procedure Declaration (type 7) : Rule [L2a]
----Step 14: Simulating Sequential Composition (type 11) : Rule [S2a] - completing statement type 7
----Step 15: Simulating Atomic While Loop Condition Evaluation
----Step 16: Simulating Atomic While Loop Condition Evaluation
----Step 17: Simulating While Loop Opening with True Condition (type 3) : Rule [W3a]
----Step 18: Simulating Step of Conditional Statement Condition Evaluation
----Step 19: Simulating Conditional Statement Opening (type 2) : Rule [I1aT] (since the condition was evaluate
----Step 20: Simulating Procedure Call Opening (type 8) : Rule [G1a]
--done (completed after 20 steps)

Simulator: Fwd-Ex> force user
--All interleaving decisions will be made by the user at runtime.

Simulator: Fwd-Ex> go 4
--Executing 4 steps if possible
The next step is the non-deterministic choice between the following multiple statements.
1: numOfSeats = numOfSeats - 1 (b0.0;)(expression evaluation)
2: while w0.0 (expression evaluation)
3: while w1.0
4: while w2.0 (expression evaluation)
Pick a statement to execute:

```

Figure 7.9: Manual interleaving decision presented to the user

program state and the reversal information, including ‘display program’, ‘display vars’ and ‘display aux’.

Many other commands allow the user to interact with the simulator. The program state and any reversal information currently saved can be viewed, with all environments displayed within the command line. Execution traces and history files can be viewed within the simulator, and can be permanently saved to avoid them being overwritten. The manual or random interleaving mode can be switched on and off at runtime, where any interleaving decision occurring under manual interleaving pauses the current execution and waits for input from the user. Each option is numbered, allowing the user to enter the number corresponding to the statement that should be executed next. Figure 7.9 contains an example of Ripple being put into manual interleaving (via the command ‘force user’), where two nested parallels mean there are four possible next steps. By default, the simulator randomly interleaves threads using random number generation. Other commands allow the default

directories to be changed, saving of reversal information to be switched on and off (to simulate an original execution) and record mode to be turned on or off.

7.4 Performance Evaluation

Implementing our approach allows the evaluation of its performance. This focuses on the *execution time* and *memory usage* of both annotated and inverted execution via Ripple. The usability of our approach depends on both of these factors being reasonable, as too large an execution time or memory usage would restrict the use of Ripple to smaller programs. For example an original program that executes over a long period of time using a large amount of memory would not be able to be simulated if the increase of either time or space is too large. We have not yet compared the original execution of programs compiled and ran in the normal way (C++ compiler) with those in Ripple. This is due to difficulties in performing this comparison, and therefore this is a target of future work.

Table 7.1: Performance evaluation of our approach

Program	Original	Annotated	Ann Overhead (ann/orig)	Inverse	Inv Overhead to Ann (inv/ann)	Inv Overhead to Orig (inv/orig)
Loop 1	0.37014943	0.3977392	1.074536843	0.24140381	0.6069399496	0.6521793374
Loop 2	1.8362783	1.9805249	1.078553779	1.1934618	0.6025987353	0.6499351433
Loop 3	3.657365	3.9569446	1.081911321	2.390812	0.6042065891	0.6536979492
Airline 1	0.01374155	0.01417653	1.031654362	0.01359069	0.9586753599	0.9890216169
Airline 2	0.02621123	0.02744856	1.047206102	0.0263826	0.9611651759	1.006538037
Airline 3	0.05101487	0.05421426	1.062714852	0.05228272	0.9643721043	1.024852558
General 1	0.11839651	0.13549897	1.144450711	0.15150618	1.118135289	1.279650726
General 2	0.20901442	0.24076207	1.151892152	0.26914268	1.117878244	1.287675176
General 3	0.30034935	0.34899057	1.161948811	0.39145901	1.121689363	1.303345621
Bsort 1	0.00102031	0.00110894	1.086865756	0.00097308	0.8774866088	0.9537101469
Bsort 2	0.00465217	0.00519253	1.116152247	0.00503436	0.9695389338	1.08215306
Bsort 3	0.02696733	0.03092997	1.146942245	0.02922749	0.9449569463	1.083811041

7.4.1 Forward Execution Time

We first compare the execution time of an original execution with that of the matching annotated execution, with all results shown in Table 7.1 (in seconds) and represented as a graph in Figure 7.10. This determines the overhead as a result of saving information, and uses the following four programs, each executed 100 times (to calculate an average) on an Intel Core i5 quad core 3.2GHz computer with 7.7Gb memory, running Linux Ubuntu 16.04.

1. **Loop** an outer loop performs 100 iterations, with a nested loop performing 1000 (Loop 1 - see Figure 7.11), 5000 (Loop 2) and 10,000 (Loop 3) iterations, each with an assignment. The average overheads are 7.45%, 7.86% and 8.19% respectively.

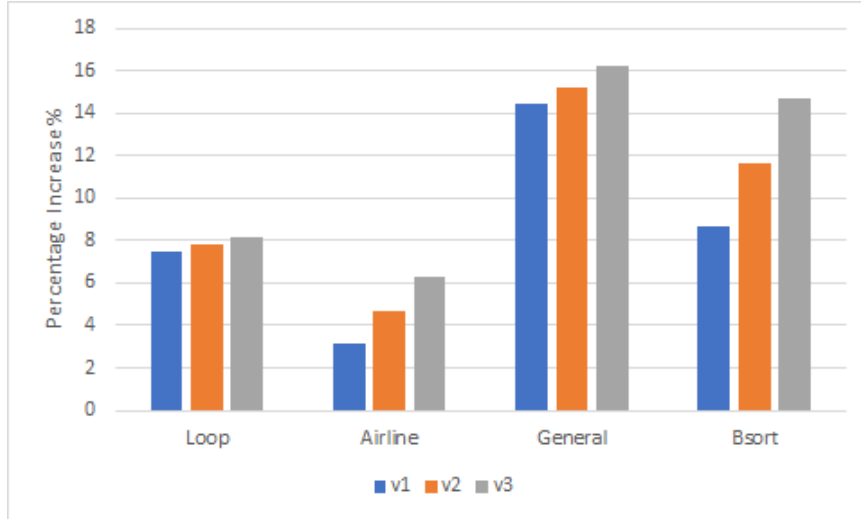


Figure 7.10: Graph representation of the overhead of annotated execution for each of our programs, showing the percentage increase of execution time of each of the three versions.

2. **Airline** an airline model where tickets are sold by four parallel agents. There are initially 100 (Airline 1 - see Figure 7.12), 200 (Airline 2) and 400 (Airline 3) free seats. The average overheads are 3.17%, 4.72% and 6.27% respectively.
3. **General** a program using a mixture of all constructs of our language. Two loops in parallel each (with 100 iterations) using a procedure call with 5 (General 1 - see Figure 7.13), 10 (General 2) and 15 (General 3) recursive calls. The average overheads are 14.45%, 15.19% and 16.19% respectively.
4. **Bubble sort** a bubble sort of 5 (Bsort 1 - see Figure 7.14), 10 (Bsort 2) and 20 (Bsort) elements. The average overheads are 8.69%, 11.62% and 14.69% respectively.

As shown by the data in Table 7.1 and the graph in Figure 7.10, the overhead associated with the four programs above ranges from 3.17% (Airline 2) up to 16.19% (General 3), a range that we deem to be reasonable. The results show a trend of *linearly* increased overhead as the length of the program execution increases (as expected since the amount of reversal information increases).

We note that the three versions of the Bsort are essentially different programs. Due to not supporting arrays, as the number of elements increases, the length of the program does also. In this case, this does not adversely affect our results.

We briefly compare the results shown here to those using the Backstroke framework in [75]. Though not perfect for comparison as Backstroke focuses on applying reverse computation to Parallel Discrete Event Simulation (PDES), evaluation in [75] shows an increase factor of between 2 and 3 for an execution of 100,000 events (in a mode that allows memory reuse and reduces re-initialisation of structures),

```

1 X = 100;
2 while w1.0 (X > 0) do
3   Y = 1000;
4   while w2.0 (Y > 0) do
5     Y = Y - 1;
6   end;
7   X = X - 1;
8 end;

```

Figure 7.11: The program **Loop 1**

```

1 numOfSeats = 100;
2 begin b1.0
3   var agent1open = 1;
4   var agent2open = 1;
5   var agent3open = 1;
6   var agent4open = 1;
7   proc p1.0 sellTicket is
8     numOfSeats = numOfSeats - 1;
9   end
10
11   par {
12     par {
13       while w1.0 (agent1open == 1) do
14         if i1.0 (numOfSeats > 0) then
15           call c1.0 sellTicket;
16         else
17           agent1open = 0;
18         end;
19       end;
20     } {
21       while w3.0 (agent3open == 1) do
22         if i3.0 (numOfSeats > 0) then
23           call c3.0 sellTicket;
24         else
25           agent3open = 0;
26         end;
27       end;
28     }
29   } {
30     par {
31       while w2.0 (agent2open == 1) do
32         if i2.0 (numOfSeats > 0) then
33           call c2.0 sellTicket;
34         else
35           agent2open = 0;
36         end;
37       end;
38     } {
39       while w4.0 (agent4open == 1) do
40         if i4.0 (numOfSeats > 0) then
41           call c4.0 sellTicket;
42         else
43           agent4open = 0;
44         end;
45       end;
46     }
47   }
48 end

```

Figure 7.12: The program **Airline 1**

which can rise for executions with a small number of operations. As the number of operations increases, the factor increase becomes less and less significant.

7.4.2 Reverse Execution Time

We now compare the time of an inverted execution with that of both the corresponding original and annotated executions. Table 7.1 contains the results, detailing the overhead incurred as a result of reversal. Figure 7.15 compares the inverse execution times with the matching annotated execution times, while Figure 7.16 compares the inverse and original execution times. We return to the four programs used above and consider each in turn.


```

1 begin b1.0
2   var left = 100;
3   var right = 100;
4   var loop1Count = 5;
5   var loop2Count = 5;
6   proc p1.0 fun1 is
7     begin b2.0
8       var other = 0;
9       if i3.0 (loop1Count > 0) then
10        loop1Count = loop1Count - 1;
11        call c1.0 fun1;
12      else
13        loop1Count = loop1Count - 1;
14        other = other + 1;
15      end;
16    end;
17  end
18  proc p2.0 fun2 is
19    begin b3.0
20      var other = 0;
21      if i4.0 (loop3Count > 0) then
22        loop2Count = loop2Count - 1;
23      call c2.0 fun2;
24    else
25      loop2Count = loop2Count - 1;
26      other = other + 1;
27    end;
28  end;
29 end
30 par {
31   while w1.0 (left > 0) do
32     left = left - 1;
33     call c3.0 fun1;
34     loop1Count = 5;
35   end;
36 } {
37   while w2.0 (right > 0) do
38     right = right - 1;
39     call c4.0 fun2;
40     loop2Count = 5;
41   end;
42 }
43 end

```

Figure 7.13: The program **General 1**

```

1  l1 = 19;
2  l2 = 14;
3  l3 = 5;
4  l4 = 4;
5  l5 = 1;
6  countu1 = 5;
7  countu2 = 5;
8  tempu = 0;
9
10 while w1.0 (countu1 > 1) do
11   while w2.0 (countu2 > 1) do
12     if i1.0 (countu2 == 5) then
13       if i2.0 (l1 > l2) then
14         tempu = l2;
15         l2 = l1;
16         l1 = tempu;
17       else
18         skip;
19       end;
20     else
21       if i2.0 (countu2 == 4) then
22         if i2.0 (l2 > l3) then
23           tempu = l3;
24           l3 = l2;
25           l2 = tempu;
26         else
27           skip;
28         end;
29       else
30         if i3.0 (countu2 == 3) then
31           if i4.0 (l3 > l4) then
32             tempu = l4;
33             l4 = l3;
34             l3 = tempu;
35           else
36             skip;
37           end;
38         else
39           if i5.0 (countu2 == 2) then
40             if i6.0 (l4 > l5) then
41               tempu = l5;
42               l5 = l4;
43               l4 = tempu;
44             else
45               skip;
46             end;
47           else
48             skip;
49           end;
50         end;
51       end;
52     end;
53     countu2 = countu2 - 1;
54   end;
55   countu2 = 5;
56   countu1 = countu1 - 1;
57 end;

```

Figure 7.14: The program **Bsort 1**

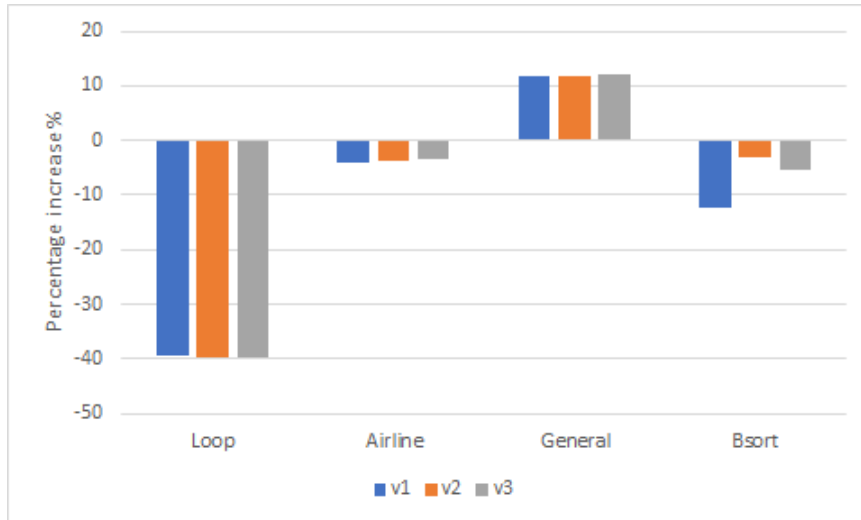


Figure 7.15: The overhead of inverted executions compared to the corresponding annotated execution. The percentage change in execution time of the three versions of each program.

1. **Loop** each version shows an inverse execution time reduction of 39.31%, 39.74% and 39.58% respectively when compared to the annotated execution, and 34.78%, 35.01% and 34.63% respectively compared to the original.
2. **Airline** each version shows an inverse execution time reduction of 4.13%, 3.88% and 3.56% compared to the annotated execution, and a reduction of 1.10%, and increase of 0.65% and 2.49% when compared to the original execution.
3. **General** each version shows an inverse execution time overhead of 11.81%, 11.79% and 12.17% compared to the annotated execution, and 27.97%, 28.77% and 30.33% when compared to the original execution.
4. **Bubble sort** each version has an inverse execution time reduction of 12.25%, 3.05% and 5.50% compared to the original execution, and a reduction of 4.63% and overhead of 8.22% and 8.38% when compared to the original.

We see that the inverse execution time ranges from 39.74% (Loop 2) faster to 12.17% (General 3) slower than the respective annotated execution, and from 35.01% (Loop 2) faster to 30.33% (General 3) slower than the original execution. Programs high in evaluation such as the loop programs are shown to have a much faster inverse execution time as none of this evaluation is performed during reversal, with all values retrieved from a stack. We also see that the inverse execution is typically faster than the annotated execution of such programs, as operations to write reversal information (forward) tend to be slower than those to read it (reverse).

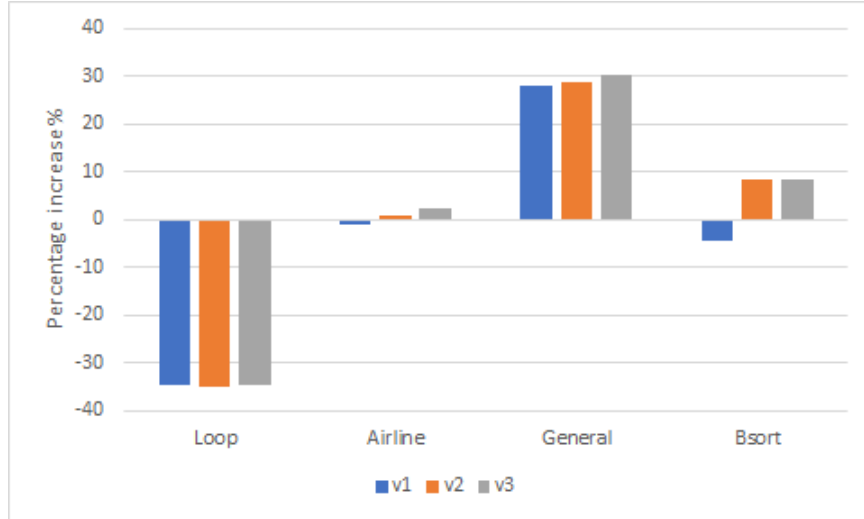


Figure 7.16: The overhead of inverted executions compared to the corresponding original execution. The percentage change in execution time of the three versions of each program.

We also highlight the results of the program **General**, where the inverse execution is slower than both the original and annotated executions. This execution contains large sections of parallel composition, where a forward execution can arbitrarily determine the interleaving order. The reverse execution of a parallel program cannot interleave statements arbitrarily and must use identifiers to determine the next step. In the worst case scenario, each interleaving decision within a reverse execution would require all possible statements to be checked (retrieving the most recent identifier it used) before finding the one with the correct identifier (namely the current value of *previous()*). This process may lead to an increased execution time, and is one example of an inefficiency within the current implementation of Ripple. In general no optimisations have been applied to Ripple, though several features could be optimised in the future, as discussed in Chapter 9.

7.4.3 Forward Memory Usage

We move now to evaluate the memory usage (heap), beginning with the difference between an original program and the matching annotated version, with our results shown in Table 7.2 (where all memory values are in Kibibytes). All results were obtained using the memory profiler Valgrind [88]. We begin with a discussion of the trade-off between execution time and memory usage in reversible computation [92], relating to the current implementation of saving all identifiers from a loop or procedure copy prior to its removal from the appropriate environment.

The majority of statements within such a copy contain identifier stacks, populated throughout its execution. These are lost as the copy is deleted. The imple-

Table 7.2: Forward memory use evaluation of our approach

Program	Original	Annotated	Ann Overhead (ann/orig)	2nd Implem.	Ann Overhead (2nd Implem./orig)
Airline 1	286,512	488,792	1.7060	295,967	1.0330
Airline 2	422,576	806,176	1.9078	439,272	1.0395
Airline 3	691,072	1,441,112	2.0853	759,552	1.0991
Bsort 1	297,488	452,968	1.5226	306,132	1.0290
Bsort 2	869,088	1,581,312	1.8195	893,032	1.0276
Bsort 3	3,225,080	6,270,056	1.9442	3,353,992	1.0340

mentation of saving these will typically favour either execution time, or memory usage. All results shown in this chapter use an implementation that produces the best execution times, namely where a copy of each stack is recorded. In the previous subsection we concluded that the execution time of an annotated execution incurs a reasonable *linear* overhead. We now consider the memory usage of the bubble sort and airline programs.

1. **Airline** - the annotated execution of each version of the airline incurs a memory usage overhead of 70.60%, 90.78% and 108.53% respectively.
2. **Bubble sort** - the annotated execution of each version of the bubble sort incurs a memory usage overhead of 52.26%, 81.95% and 94.42% respectively.

This shows both programs incur a manageable increase in memory usage. In light of these results, we have performed other experiments using a different implementation of saving any identifiers lost when a copy is removed from either the while or procedure environment. This version involved traversing all of the stacks that need to be saved, and producing a string representation of all identifiers. This is time consuming, leading to an increased execution time, however due to the trade-off between time and memory, this increase in execution time allows this more memory efficient approach to be applied. As can be seen from Table 7.2, this second implementation incurs a much lower memory overhead, namely 3.30% to 9.91% for the airline programs, and 2.76% to 3.40% for the bubble sort programs.

The results shown here indicate the current implementation of this process, where copies of multiple identifier stacks are stored, can be improved in the future. We return to this in Chapter 9 where we detail our future work of finding an implementation that balances the increase of execution time and memory usage.

7.4.4 Reverse Memory Usage

Our final consideration is the memory usage of a reverse execution, compared only to that of the annotated execution (with the implementation favouring execution time as described in the previous subsection). We again evaluate the airline and

bubble sort programs used above, with the results detailed in Table 7.3 where all values are in Kibibytes.

Table 7.3: Reverse memory use evaluation of our approach

Program	Annotated	Inverted	Inv Overhead (Inv/ann)
Airline 1	355,528	114,421	0.3218340046
Airline 2	665,968	167,336	0.2512673282
Airline 3	1,304,584	192,992	0.1479337475
Bsort 1	248,168	246,592	0.9936494633
Bsort 2	1,013,688	656,464	0.6475996559
Bsort 3	4,276,224	2,541,776	0.5943973

1. **Airline** each version shows an inverted memory use reduction of 67.82%, 74.87% and 85.21% respectively, compared to the annotated execution.
2. **Bubble sort** each version shows an inverted memory use reduction of 0.64%, 35.24% and 40.56% respectively, compared to the annotated forward execution.

All of the programs tested above show that the inverse execution uses less memory than the corresponding annotated execution, ranging from 0.64% to 85.21%. This is as expected since the annotated execution saves reversal information, while the inverted version uses this information (and does not save anything else). The result for Bsort 1 is different to the rest, showing only a very small reduction. Though further analysis is required, we believe this to be a result of the inefficient bubble sort program (as we do not support arrays) and potential inefficiencies in the implementation of Ripple. Obviously there remains the use of memory during the inverted execution, including to maintain the program state.

7.5 Conclusion

We have introduced the simulation tool Ripple, and outlined both the software architecture and the user interface. This chapter details the performance evaluation produced via Ripple, showing that the execution time overhead associated with annotation (between 4.72% and 16.19% slower for the programs tested) is linear. Inverted executions are shown to typically be faster than the forward execution, with the notable exception to this likely produced as a result of inefficient optimisation. The evaluation of memory usage shows that the current implementation can result in a large increase (between 52.26% and 108.53%). We have highlighted the process within Ripple that is responsible for this increase, and performed experiments using a modified implementation which performed better (between 2.90% and 9.91%).

We have discussed that the trade-off for decreasing the memory usage increases the execution time, and how in future work we would like to find another implementation that balances these two factors.

Chapter 8

Application to Debugging

A correct approach to reversibility of a concurrent programming language has several potential applications, including to Parallel Discrete Event Simulation (PDES) [21], as shown by the Backstroke framework [93] and by the work of Schordan et al [75, 76, 77]. Another possibility is to the field of reversible debugging, which is explored further in this chapter. Recall the background of debugging and the discussion regarding the link between reverse execution and debugging from Chapter 1. Previous examples of reverse debuggers include [12, 91], and causal-consistent reverse debuggers include [46, 24].

8.1 Suitability of the Approach

The simulation tool Ripple, introduced in Chapter 7, was initially developed to implement the operational semantics of our programming language as defined in Chapter 4. This tool can be used to simulate the execution of an original, annotated and inverted program. As Ripple was developed, we began to add features that could be useful for debugging. In addition, other features required for execution simulation can also help with finding software errors. Such features included:

1. Small-step semantics allow the execution to be paused at any point. Intermediate program states can be viewed, and compared with the expected state. This includes current position and current values of variables.
2. All reversal information saved up to a specific point can also be viewed. This can display the current number of loop iterations, all previous values of a variable and all results of evaluating conditional statements.
3. Program state is accessible in intermediate states, and can be changed to test, for example, bug fixes.

```

1  if i1.0 (X < 5) then          X = 20 [0];
2    X = X + 10 [];             par    Y = 4 [3];
3  else
4    X = X - 7 [2];
5    Y = 30 [4];
6  end; [1,5];

```

Figure 8.1: Example concurrent program under a specific interleaving

4. Ripple contains a *record mode* which produces two files containing sufficient information to reproduce a specific execution. These are the *interleaving history* and the *semantics history*, both described in Section 8.2 below.

8.2 Record Mode and Execution Histories

The record mode of our simulation tool is not related to reversal. No information saved by this mode is used to implement reverse execution and is instead used for the purposes of debugging. Aside from reversible execution, it is useful to be able to reproduce a specific execution. Consider an arbitrary forward execution that unexpectedly experiences a bug. Reverse execution can be used to find the underlying cause (as shown in Section 8.3 and Section 8.4 below) while any fix that is then implemented can be tested via executing the program forwards again.

In order to correctly test a fix, the specific interleaving order of the original incorrect program must be repeated. Due to interleaving parallel composition, there is no guarantee that two consecutive executions will have the same behaviour. Using the inverted version of the program, the populated identifier stacks contain sufficient information to reproduce an execution. To aid this process, Ripple produces an interleaving and semantics history. Since the process is identical regardless of whether a bug occurs, the description and example of each uses the small concurrent program (that does not contain a bug) shown in Figure 8.1.

Interleaving history

The difficulty of reproducing a bug within a concurrent program is the potential for different interleaving orders. Any bug can successfully be reproduced in our approach provided the result of each interleaving decision is known and can be used during the following forward execution. Therefore Ripple can record the outcome of each interleaving decision, indicating all of the possible statements and showing which was executed. Recalling the program in Figure 8.1, the corresponding interleaving history file is shown in Figure 8.2(a). Each line corresponds to an interleaving decision, with all possible next steps shown in the first set of brackets

1 [if i1.0 (open), X = 20] ----> [X = 20]	1 [D1a]
2 [if i1.0 (open), skip (X = 20)] ----> [if i1.0 (open)]	2 [I1aT]
3 [X = X - 7, skip (X = 20)] ----> [X = X - 7]	3 [D1a]
4 [skip (X = X - 7), skip (X = 20)] ----> [skip (X = 20)]	4 [S2a]
5 [skip (X = X - 7), Y = 4] ----> [skip (X = X - 7)]	5 [S2a]
6 [Y = 30, Y = 4] ----> [Y = 4]	6 [D1a]
7 [Y = 30] ----> [Y = 30]	7 [D1a]
8 [if i1.0 (close)] ----> [if i1.0 (close)]	8 [I5a]
9 [par (close)] ----> [par (close)]	9 [P3a]
10 END OF FILE	10 END OF FILE

(a) Interleaving history file

(b) Semantics history file

Figure 8.2: Example history files of an execution via Ripple

and the decision shown in the second. Consider line 1 of Figure 8.2(a), which states the first interleaving decision made was between the opening of the conditional statement `if i1.0 (open)` (where `open` simply indicates the conditional is about to open and is not part of the language syntax) and the assignment `X = 20`. The arrow `---->` indicates the choice at this point was the assignment `X = 20`. Line 4 shows the decision at this point was between two applications of [S2a] that remove a skip statement. Each entry contains the statement from which this skip originated appended in brackets (avoiding ambiguity of which was executed). This file can then be used (reading from top to bottom) during a following forward, manually interleaved execution of the same program, to determine the exact interleaving order. Any ambiguity between identical statements can be broken by looking further into the history file (to determine which statement it makes available) or by using the identifiers from the inverted version.

Semantics history

Recall the semantics from Chapter 4 and the discussion of inference trees. In addition to all interleaving decisions, Ripple also records a sequence of transition rule names. Specifically this contains only the leaf rules (those that appear at the top of the corresponding inference tree). Returning to the program in Figure 8.1, the semantics history file of this execution is shown in Figure 8.2(b). Each line contains a single rule name (the leaf rule from the top of the inference tree) and corresponds to the matching line in Figure 8.2(a). Consider again line 1 of both history files in Figure 8.2. The decision to perform the assignment is reflected by inserting [D1a] into the semantics history. Line 4 of both files again reflect the decision to perform the skip step by inserting [S2a]. The two history files shown here can be used together to reproduce any given execution. They can also be used to test equality

<pre> 1 seats = 3 [0]; 2 begin b1.0 3 var agent1 = 1 [1]; 4 var agent2 = 1 [2]; 5 proc p1.0 sell is 6 seats = seats - 1 [7,13,22,23]; 7 end [3]; 8 par { 9 while w1.0 (agent1 == 1) do 10 if i1.0 (seats > 0) then 11 call c1.0 sell [6,8,20,24]; 12 else 13 agent1 = 0 [31]; 14 end [5,9,18,26,30,32]; 15 end [4,17,29,33]; 16 } { 17 while w2.0 (agent2 == 1) do 18 if i2.0 (seats > 0) then 19 call c2.0 sell [12,14,21,25]; 20 else 21 agent2 = 0 [35]; 22 end [11,15,19,27,34,36]; 23 end [10,16,28,37]; 24 } 25 remove proc p1.0 sell end [38]; 26 remove var agent2 = 1 [39]; 27 remove var agent1 = 1 [40]; 28 end </pre>	<pre> 1 begin b1.0 2 var agent1 = 1 [40]; 3 var agent2 = 1 [39]; 4 proc p1.0 sell is 5 seats = seats - 1 [7,13,22,23]; 6 end [38]; 7 par { 8 while w1.0 (agent1 == 1) do 9 if i1.0 (seats > 0) then 10 call c1.0 sell [6,8,20,24]; 11 else 12 agent1 = 0 [31]; 13 end [5,9,18,26,30,32]; 14 end [4,17,29,33]; 15 } { 16 while w2.0 (agent2 == 1) do 17 if i2.0 (seats > 0) then 18 call c2.0 sell [12,14,21,25]; 19 else 20 agent2 = 0 [35]; 21 end [11,15,19,27,34,36]; 22 end [10,16,28,37]; 23 } 24 remove p1.0 sell end [3]; 25 remove agent2 = 1 [2]; 26 remove agent1 = 1 [1]; 27 end 28 seats = 3 [0]; </pre>
---	---

(a) Executed Annotated Program

(b) Inverted Program

Figure 8.3: Airline Example

between two different executions. If both files of two distinct executions match, we can conclude that both executions are identical since both are guaranteed to have the same effect on the program state.

8.3 Airline Example

Consider a program that models an airline selling tickets via two agents. Each agent is represented using a while loop that iterates until the agent is closed. Each iteration uses a conditional statement to determine whether at least one free seat remains. If it does, the agent calls the procedure to sell a ticket, while if it does not, the agent must close. Each of the two agents operates in parallel, using a shared procedure

`sell` and current number of seats. This program is shown in Figure 8.3(a), where the number of initially free seats is 3, and the number of agents is 2, in order to keep the execution and accompanying environments concise enough for discussion here. Note that this example can easily be extended to have multiple agents, as is done in Chapter 7 when evaluating performance. The specific execution captured in Figure 8.3(a) incorrectly results in 4 tickets being sold when only 3 are initially available. This is observable from the final program state, where `seats = -1`.

We now investigate this bug using reverse execution. The inverted version of this program is shown in Figure 8.3(b). The inverse execution begins in the incorrect final state and opens the block and re-declares the local variables and the procedure (using identifiers 40–38). Next, the parallel statement is started, with each while loop executing an entire iteration (to simulate the inversion of the final iteration of each that closed each agent) using identifiers 37–30. From here, we now begin the inversion of the penultimate iterations of each while loop (the last time each allocated a seat). The identifiers 29–20 are used to govern the interleaving across the two threads. At this point, the choice of next step is between the closing of the two inverse conditional statements. Recall from the semantics of Chapter 4 that the closing of an inverse conditional statement reverses the opening of the forward version of it. The inverse execution can now perform the next two steps using identifiers 19 and 18, where each conditional is closed. This implies that both conditional statements were opened consecutively during the forward execution, with the condition of each evaluated before execution of the branches. Viewing the program state via Ripple shows that the current number of seats is 1, meaning both of the conditional statements evaluate to true.

The simulator has shown that both agents commit to selling another ticket in the situation where only one remains. Therefore the cause of the bug is a race between the reading of and updating of the variable `seats`. Specifically there is an *atomicity violation* between checking for a free seat and the allocation of it.

This bug can be fixed by ensuring the critical sections of each agent (namely the conditional statement of each) are mutually exclusive. The interleaving order followed in the example above can be avoided by restricting the interleaving once an agent enters the critical section. This would mean that once an agent checks for at least one free seat, no interleaving is allowed until the agent has sold one ticket. This can be implemented using *atomic sections* to encapsulate a critical section, where the operational semantics of parallel composition are updated to restrict interleaving if one potential step is within an atomic section.

```

1 begin b1.0
2   var X1 = -1 [0];
3   var X2 = -1 [1];
4   var X3 = -1 [2];
5   var toPush = 0 [3];
6   var current = 1 [4];
7   proc p1.0 push is
8     if i1.0 (current == 1) then
9       X1 = toPush [10,61];
10    else
11      if i2.0 (current == 2) then
12        X2 = toPush [18];
13      else
14        if i3.0 (current == 3) then
15          X3 = toPush [];
16        else
17          Y = 100 [52];
18        end [51,53];
19      end [17,19,50,54];
20    end [9,11,16,20,49,55,60,62];
21    current = current + 1 [12,21,56,63];
22  end [5];
23  proc p2.0 pop is
24    if i4.0 (current == 1) then
25      Y = 200 [44];
26    else
27      if i5.0 (current == 2) then
28        X1 = -1 [36];
29      else
30        if i6.0 (current == 3) then
31          X2 = -1 [27];
32        else
33          if i7.0 (current == 4) then
34            X3 = -1 [];
35          else
36            Y = 400 [];
37          end [];
38        end [26,28];
39      end [25,29,35,37];
40    end [24,30,34,38,43,45];
41    current = current - 1 [31,39,46];
42  end [6];
43
44  toPush = 4 [7];
45  call c1.0 push [8,13];
46  toPush = 7 [14];
47  call c2.0 push [15,22];
48  call c3.0 pop [23,32];
49  call c4.0 pop [33,40];
50  toPush = 100 [41];
51  call c5.0 pop [42,47];
52  call c6.0 push [48,57];
53  toPush = 55 [58];
54  call c7.0 push [59,64];
55
56  remove p2.0 pop end [65];
57  remove p1.0 push end [66];
58  remove current = 1 [67];
59  remove toPush = 0 [68];
60  remove X3 = -1 [69];
61  remove X2 = -1 [70];
62  remove X1 = -1 [71];
63 end

```

Figure 8.4: Executed annotated stack program

8.4 Stack Example

Figure 8.4 contains a program written using our programming language that implements a stack. Since stacks and arrays are not currently supported, the implementation relies on the use of variables and conditional statements. This program implements a stack of three elements, each represented using local variables `X1`, `X2` and `X3` (used in ascending order). As procedures do not support arguments, the value pushed next to the stack is always that of `toPush`, with `current` indicating the head of the stack.

The block statement contains two procedures, namely `push` and `pop`. Through the use of nested conditional statements, each of the functions modifies the top of the stack. Consider the execution of the program in Figure 8.4. Lines 44–54 describe the interaction with the stack, beginning with the pushing of both 4 and 7 to the stack. Lines 48, 49 and 51 show three calls of the `pop` procedure, removing both elements of the stack. Though this is deemed incorrect, this should not lead to an

```

1 begin b1.0
2   remove X1 = -1 [71];
3   remove X2 = -1 [70];
4   remove X3 = -1 [69];
5   remove toPush = 0 [68];
6   remove current = 1 [67];
7   proc p1.0 push is
8     current = current + 1 [12,21,56,63];
9     if i1.0 (current == 1) then
10      X1 = toPush [10,61];
11    else
12      if i2.0 (current == 2) then
13        X2 = toPush [18];
14      else
15        if i3.0 (current == 3) then
16          X3 = toPush [];
17        else
18          Y = 100 [52];
19        end [51,53];
20      end [17,19,50,54];
21    end [9,11,16,20,49,55,60,62];
22  end [66];
23  proc p2.0 pop is
24    current = current - 1 [31,39,46];
25    if i4.0 (current == 1) then
26      Y = 200 [44];
27    else
28      if i5.0 (current == 2) then
29        X1 = -1 [36];
30      else
31        if i6.0 (current == 3) then
32          X2 = -1 [27];
33        else
34          if i7.0 (current == 4) then
35            X3 = -1 [];
36          else
37            Y = 400 [];
38          end [];
39        end [26,28];
40      end [25,29,35,37];
41    end [24,30,34,38,43,45];
42  end [65];
43
44  call c7.0 push [59,64];
45  toPush = 55 [58];
46  call c6.0 push [48,57];
47  call c5.0 pop [42,47];
48  toPush = 100 [41];
49  call c4.0 pop [33,40];
50  call c3.0 pop [23,32];
51  call c2.0 push [15,22];
52  toPush = 7 [14];
53  call c1.0 push [8,13];
54  toPush = 4 [7];
55
56  remove p2.0 pop end [6];
57  remove p1.0 push end [5];
58  remove current = 1 [4];
59  remove toPush = 0 [3];
60  remove X3 = -1 [2];
61  remove X2 = -1 [1];
62  remove X1 = -1 [0];
63 end

```

Figure 8.5: Inverted version of the stack program

error as popping an empty stack should simply do nothing. Finally, we push the values 100 and 55 to the stack. The expected final stack after this execution (and prior to the removal statements) is $X1 = 100$, $X2 = 55$ and $X3 = -1$ (where the top of the stack is currently $X2$ as $X3$ is not used). Note we assume that the value -1 is restricted and cannot be pushed to our stack.

Now consider the execution of the program from Figure 8.4, where the final version of the stack (prior to the removal statements) incorrectly contains the values $X1=55$, $X2=-1$ and $X3=-1$ (where the top of the stack is $X1$). The observable bug here is that the value 100 pushed via lines 50–51 is not within the stack.

We now investigate this bug using reverse execution. The inverted version is generated and shown in Figure 8.5. With this program being sequential, the identifiers are not used to determine the inverse interleaving order (as this is fixed in a sequential program) and instead only used to link reversal information to the statement that requires it. The inverse execution begins by opening the block and re-declaring all local variables and the two procedures. The call `c7.0` is then reversed, removing

the value 55 from the stack. The final step of this inverted call restores the value of `current` to 1. This raises concerns as it indicates an empty stack (when we know that it should contain 100). The first element `X1` has also been restored to -1, meaning no value was incorrectly overwritten here (namely the missing 100).

We therefore continue further back, executing line 45 which restores `toPush` to the value 100. Next the call `c6.0` is now inverted, with the only change being to the variable `current` which is restored to 0. This is very alarming for the same reasons as above, where an empty stack (as here) should have `current` = 1. It is clear that this value is incorrectly changed at some point in the remaining execution.

The next step is to invert the call `c5.0` to the `pop` procedure. We see that the assignment from line 24 is used to restore `current` to the value 1. No other change is made here as each of the conditionals retrieve an `F` from the stack (auxiliary store). This call to `pop` corresponds to the previously mentioned third instance, called during the forward execution on an empty stack. Such a call should not lead to a fatal error, and instead should do nothing. Examining the code of this procedure (lines 23–42 of Figure 8.4), we note that `current` is decremented regardless of whether an element is popped. This is the cause of the error, as this `pop` action results in the following push (of 100) not correctly inserting the value.

This bug can be fixed by re-working the procedure `pop`, ensuring the value of `current` is not changed unless an element is removed. This involves removing line 41 of Figure 8.4 and instead performing this decrement in each of the true branches of the conditionals with construct identifiers `i4.0`, `i5.0`, `i6.0` and `i7.0`. The updated version of the procedure `pop` is shown below.

```

1  proc p2.0 pop is
2    if i4.0 (current == 1) then
3      Y = 200 [];
4    else
5      if i5.0 (current == 2) then
6        X1 = -1 [];
7        current = current - 1 [];
8      else
9        if i6.0 (current == 3) then
10         X2 = -1 [];
11         current = current - 1 [];
12       else
13         if i7.0 (current == 4) then
14           X3 = -1 [];
15           current = current - 1 [];
16         else
17           Y = 400 [];
18         end [];
19       end [];
20     end [];
21   end [];
22 end [6];

```

8.5 Conclusion

We have demonstrated that our simulation tool Ripple is suitable for use within debugging. We have used examples of common types of software bugs and described the process of using Ripple to help find the cause. The ability to step-by-step reverse execute a program with the program state viewable at each point is useful, as is the debugging-specific feature of record mode. The history files produced by record mode allow a user to perform traditional, cyclic debugging as any arbitrary execution can be (manually) reproduced. Ripple is not a complete debugger, and cannot be used to fix software errors at runtime. It is however a helpful aid in determining the underlying cause of misbehaviour.

Chapter 9

Conclusion

We now summarise and evaluate the research presented here. We also outline topics of potential future work.

9.1 Summary

We have proposed an approach to reversibly executing programs written in an imperative, concurrent programming language. Reversal information is recorded during the forward execution, via a modified operational semantics defining the annotated execution. We have presented a method of recording the number of iterations of each loop without modifying the behaviour of the program, which means we do not require a fresh variable for each loop. We have introduced the use of identifiers as a method of capturing an arbitrary interleaving order resulting from interleaving parallel composition. Each time a statement is executed, the next available identifier (used in ascending order) is assigned to that statement. This records the result of each interleaving decision, ensuring the ability to reproduce any execution.

We have generated an inverted version of a given program which has the inverted statement order, and executes via a third modified operational semantics that reverse the effects of the respective forward step. All interleaving decisions made within a reverse execution are determined via the identifiers, with only the statement containing the current highest identifier allowed to execute (with the exception of skip steps that can freely interleave).

The reversibility implemented here is shown to be correct. We have shown that saving all reversal information, including the identifiers, does not interfere with the underlying program behaviour. Two executions of the same program, one with saving enabled and one without, will produce equivalent final program states (one records reversal information). Secondly, execution of an inverted program beginning in the appropriate final program state restores this state to as it was prior to the

corresponding forward execution, including using all reversal information such that it is garbage free.

We have implemented the three operational semantics in our simulator Ripple. It allowed us to demonstrate our proposed method of reversibility did indeed work, and to test the reversibility of larger programs that would be difficult to perform by hand. This experimentation with larger programs aided the development, highlighting issues and the necessary adjustments to overcome these. Ripple has also been used to produce performance evaluation. Our results show that our current implementation incurs linearly increasing overheads with respect to both execution time and memory usage. This has also indicated that Ripple can be optimised with respect to the space-time trade-off [92]. We have demonstrated the use of Ripple as an aid to debugging with two examples of commonly occurring bugs. The ability to execute step-by-step in both directions, to view the program state at any intermediate position and to determine interleaving manually means Ripple can be useful within reverse debugging.

9.2 Evaluation

The following is a list of objectives we set out to achieve in Chapter 1. Now we discuss whether or not each of them was met.

- To propose a method of reversing executions of a concurrent imperative programming language.

We have achieved this, considering first a concurrent while language (Chapter 3), before extending this with additional constructs (Chapter 4). This is achieved by saving information during a forward execution that is used during an (step-by-step) inverted execution to revert all effects on the program state.

- To support common programming constructs from typical, widely used programming languages.

Our initial while language was then extended with block statements, local variable declarations and potentially recursive procedures (Chapter 4). This also includes removal statements that clean any locally declared variables or procedures at the end of the corresponding block statement. Some other programming constructs frequently used in traditional programming languages are not currently supported, including procedure arguments and return statements within procedure calls (functions), pointers and data structures such as arrays or lists. This addition is an example of possible future work (see Section 9.3).

- To allow concurrent composition of programs, and to reverse in backtracking order (reversing statements in exactly the inverted order in which they were performed forwards).

Our programming language supports an interleaving form of parallel composition. Executions containing interleaving have the specific execution order captured by annotating a series of identifiers to each statement as they execute. Backtracking reversibility is performed in the majority of cases, using the identifiers in descending order to determine which is due to be reversed next. We show our approach performs one possible relaxation to this, namely skip steps during an inverse execution can happen at any point after they become available (respecting the program order of statements). Another possible relaxation (that has not been implemented here) is to support causal-consistent reversibility [15, 64, 67], which we believe can be achieved using the approach outlined in Section 9.3.

- To ensure that our proposed method performs correct reversibility and does not produce garbage.

We have proved the correctness of our method in Chapter 6. This shows that the process of saving information during a forward execution and annotating a program with identifiers does not change the behaviour of the underlying program. Secondly, the inverse execution correctly restores the initial program state, as well as using all reversal information saved and therefore being garbage free.

- To implement our method in the form of a simulator, that can be used to show that our approach works, to perform tests that would be difficult to do so by hand, and to evaluate the performance in terms of execution time and memory usage overheads.

We have introduced Ripple, a simulator implementing the operational semantics defined in this work. This has been used to evaluate the performance. We have focused on the overhead on a forward execution time, and the increased memory usage. The approach to reversibility proposed here results in a linear increase in both execution time and memory usage (see Chapter 7). Our experiments highlighted the space-time trade-off, and that possible future work could be to optimise the implementation of how information is saved, and therefore balance the execution time increase with that of the memory usage. Other optimisations could reduce the amount of information saved, as described in Section 9.3.

- To demonstrate the use of our method to the field of reverse debugging.

Chapter 8 explores the link between Ripple and reverse debugging. Extra features have been implemented within Ripple that do not directly relate to reversibility,

but that aid the use of Ripple as a debugger. This includes a ‘record mode’ that maintains two history files throughout a execution, namely an interleaving decision history (all possible statements and the one that was actually executed at each point) and a semantics history (an ordered list of leaf transition rules performed). Both of these histories are not required for reverse execution, but to allow cyclic debugging when used in conjunction with ‘manual interleaving mode’ (a user can reproduce a specific execution using these histories). Ripple is an aid to debugging, and could have key debugging functionality added, as described in Section 9.3.

9.3 Future Work

We now describe a number of topics of potential future work.

1. Extension of our programming language and Ripple

The obvious next step is to maintain the current approach and extend our programming language. This extension could be towards frequently used languages such as C++, with constructs including procedure arguments and return statements within procedures, data structures such as arrays, heaps, lists or stacks, pointers and different types of variables such as boolean.

2. Relaxation of backtracking reversibility to causal-consistent reversibility.

We could relax the order in which statements are allowed to invert, which currently follows backtracking reversibility. A typical execution of a concurrent program may involve performing a number of independent steps at any time. It is clear that such statements can be performed in any order as well as undone in any order (as they are independent). As in [15, 64, 67], our reversibility could be relaxed to allow causal-consistent reversibility, where the reverse execution of parallel composition can follow any order that respects causal dependencies. This would require all dependencies between statements to be known, and to be able to determine whether two statements (or steps of their execution) are independent.

This would also require modification to the use of stacks within the auxiliary store and identifiers. The LIFO nature of stacks make them ideal for backtracking reversibility as the final information saved is used first. This would no longer be the case as all variables, conditionals and loops share their respective stacks. A solution is to use a data structure that still has order but allows random access, e.g. an array or heap, or to use separate stacks for each variable, conditional and loop respectively.

Identifiers are used in descending order during an reverse execution, meaning statements can be reversed provided it holds the appropriate identifier. Causal-consistent reversibility breaks this, potentially allowing statements with lower identifiers to execute first. Our method could be updated to allow any statement with

an identifier less than or equal to the current value of *previous()* execute, provided it is independent of any other possible identifier steps with a higher identifier.

3. Optimisation of space-time trade-off.

In Chapter 7, we gave the performance evaluation of our simulation tool. We noted one particular process, namely the saving of all identifiers that are assigned to the copy of a loop or procedure body, is implemented in a such a way that favours execution time, but that requires larger memory usage. Given more time, we could optimise the implementation of this process. This could lead to an entirely different approach to recording all of the identifiers that are lost when a program copy is removed, for example assigning the appropriate identifiers directly into the auxiliary store during forward execution, which would remove the need to extract and insert (during inversion) each identifier. An alternative is to develop an implementation that extracts and inserts identifiers in such a way that balances the space-time trade-off. A possibility is to choose between our two current implementations (see Chapter 7) at runtime, choosing either the time-efficient or memory-efficient approach depending on the current program and state. We could potentially reduce the amount of identifiers that need to be stored, by spotting patterns within an execution such as sequential use of identifiers. Any sequential identifiers can be recorded simply by saving the beginning and end values. Further work is required to compare the overhead of analysing the identifiers with that of saving them all.

4. General optimisation of Ripple.

Our results in Chapter 7 show Ripple can be optimised. Guided by a memory profiler, we could determine any slow or inefficient segments of the simulator and improve the implementation. Inspired by the reversible language Janus [55, 97], we could pre-process an original program and convert all assignments (destructive) into equivalent versions (constructive increments or decrements), removing the need to save old values of variables. A technique suggested by Perumalla [63] is to spot invariant conditional statements and no longer save reversal information for it (as the condition can be re-evaluated). A challenge here is how to determine during an inverse execution whether to evaluate the condition (in the case of an invariant expression - which would increase the execution time of the inverted version) or to retrieve the value from the auxiliary store.

Our programming language could be extended with constructs commonly supported in traditional programming languages. This includes arguments and return statements within procedures, data structures such as arrays, lists or stacks, pointers and different types of variables such as boolean.

5. Develop Ripple towards a debugger.

Ripple is currently an aid to debugging, performing reverse execution of programs. We could move Ripple towards being a debugger, adding functionality including modifying the program state at runtime (to allow testing) and loading an execution (performed previously) into the simulator without having to execute it first. Another desired capability would be to perform *bi-directional executions*, where reversal can begin at any point within a forward execution (without requiring it to complete first) and programs can be executed forward again after any number of reverse steps. Other extensions could help with the ability of Ripple to perform cyclic debugging, for example to automate the process of using history files and manual interleaving to reproduce a forward execution. Finally, debugging features such as break points could also be added.

Bibliography

- [1] S. M. Abramov and R. Glück. Principles of inverse computation and the universal resolving algorithm. In *The Essence of Computation*, pages 269–295, 2002.
- [2] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.
- [3] S. V. Adve and M. D. Hill. Weak ordering - A new definition. *International Symposium on Computer Architecture*, pages 2–14, 1990.
- [4] H. Agrawal, R. A. DeMillo, and E. H. Spafford. An execution backtracking approach to debugging. *IEEE Software*, 8(3):21–26, 1991.
- [5] T. Akgul and V. J. Mooney III. Assembly instruction level reverse execution for debugging. *ACM Transactions on Software Engineering and Methodology*, 13(2):149–198, 2004.
- [6] J. Alglave and L. Maranget. diy, Release Seven. <http://diy.inria.fr/>. Accessed: 01-09-2019.
- [7] K. Barylska, A. Gogolinska, L. Mikulski, A. Philippou, M. Piatkowski, and K. Psara. Reversing computations modelled by coloured Petri nets. In *Algorithms and Theories for the Analysis of Event Data*, volume 2115 of *CEUR Workshop Proceedings*, pages 91–111, 2018.
- [8] C. H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17(6):525–532, 1973.
- [9] J. A. Bergstra, A. Ponse, and J. Van Wamel. Process algebra with backtracking. In *REX School Symposium*, pages 46–91, 1993.
- [10] B. Biswas and R. Mall. Reverse execution of programs. *SIGPLAN Notices*, 34(4):61–69, 1999.

- [11] C. D. Carothers, K. S. Perumalla, and R. Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM Transactions on Modeling and Computer Simulation*, 9(3):224–253, 1999.
- [12] S.-K. Chen, W. K. Fuchs, and J.-Y. Chung. Reversible debugging using program instrumentation. *IEEE Transactions on Software Engineering.*, 27, 2001.
- [13] J. Conrod. Tutorial: Reverse debugging with gdb 7. <http://jayconrod.com/posts/28/tutorial-reverse-debugging-with-gdb-7>, 2009.
- [14] M. H. Cservenká, R. Glück, T. Haulund, and T. Æ. Mogensen. Data structures and dynamic memory management in reversible languages. In *Reversible Computation*, volume 11106 of *Lecture Notes in Computer Science*, pages 269–285. Springer, 2018.
- [15] V. Danos and J. Krivine. Reversible communicating systems. In *CONCUR Concurrency Theory*, pages 292–307. Springer, 2004.
- [16] J. Engblom. A review of reverse debugging. In *System, Software, SoC and Silicon Debug*, pages 1–6. IEEE, 2012.
- [17] S. I. Feldman and C. B. Brown. IGOR: A system for program debugging via reversible execution. *Workshop on Parallel and Distributed Debugging*, pages 112–123, 1988.
- [18] M. P. Frank. Physical foundations of Landauer’s principle. In *Reversible Computation*, volume 11106 of *Lecture Notes in Computer Science*, pages 3–33. Springer, 2018.
- [19] M. P. Frank and T. F. Knight Jr. *Reversibility for efficient computing*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1999.
- [20] E. Fredkin and T. Toffoli. Conservative logic. *International Journal of Theoretical Physics*, 21(3):219–253, 1982.
- [21] R. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, 1990.
- [22] D. Geels, G. Altekár, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In *USENIX Annual Technical Conference*, pages 289–300, 2006.
- [23] E. Giachino, I. Lanese, and C. A. Mezzina. CaReDeb website. URL: <http://www.cs.unibo.it/caredeb>.

- [24] E. Giachino, I. Lanese, and C. A. Mezzina. Causal-consistent reversible debugging. In *FASE*, volume 8411 of *Lecture Notes in Computer Science*, pages 370–384. Springer, 2014.
- [25] E. Giachino, I. Lanese, C. A. Mezzina, and F. Tiezzi. Causal-consistent rollback in a tuple-based language. *Journal of Logical and Algebraic Methods in Programming*, 88:99–120, 2017.
- [26] R. Glück and T. Yokoyama. A linear-time self-interpreter of a reversible imperative language. *Computer Software*, 33(3), 2016.
- [27] R. Glück and T. Yokoyama. A minimalist’s reversible while language. *IEICE Transactions on Information and Systems*, E100.D:1026–1034, 2017.
- [28] E. Graversen, I. Phillips, and N. Yoshida. Event structure semantics of (controlled) reversible CCS. In *Reversible Computation*, Lecture Notes in Computer Science, pages 102–122. Springer, 2018.
- [29] T. Haulund. Design and implementation of a reversible object-oriented programming language. *CoRR*, abs/1707.07845, 2017.
- [30] T. Haulund, T. Æ. Mogensen, and R. Glück. Implementing reversible object-oriented language features on reversible machines. In *Reversible Computation*, volume 10301 of *Lecture Notes in Computer Science*, pages 66–73. Springer, 2017.
- [31] J. Hoey, I. Lanese, N. Nishida, I. Ulidowski, and G. Vidal. A case study for reversible computing: Reversible debugging of concurrent programs. In *Reversible Computation: Theory and Applications*, volume 12070 of *Lecture Notes in Computer Science*. Springer, 2020. To appear.
- [32] J. Hoey and I. Ulidowski. Reversible imperative parallel programs and debugging. In *Reversible Computation*, volume 11497 of *Lecture Notes in Computer Science*, pages 108–127. Springer, 2019.
- [33] J. Hoey, I. Ulidowski, and S. Yuen. Reversing imperative parallel programs. In *EXPRESS/SOS*, volume 255 of *Electronic Proceedings in Theoretical Computer Science*, pages 51–66, 2017.
- [34] J. Hoey, I. Ulidowski, and S. Yuen. Reversing parallel programs with blocks and procedures. In *EXPRESS/SOS*, volume 276 of *Electronic Proceedings in Theoretical Computer Science*, pages 69–86, 2018.
- [35] H. Hüttel. *Transitions and Trees - An Introduction to Structural Operational Semantics*. Cambridge University Press, 2010.

- [36] European COST Actions IC1405 on “Reversible Computation - Extending Horizons of Computing” website. <http://www.revcomp.eu/>.
- [37] P. A. H. Jacobsen, R. Kaarsgaard, and M. K. Thomsen. Corefun: A typed functional reversible core language. In *Reversible Computation*, volume 11106 of *Lecture Notes in Computer Science*, pages 304–321. Springer, 2018.
- [38] A. Jaffe, T. Moscibroda, L. Effinger-Dean, L. Ceze, and K. Strauss. The impact of memory models on software reliability in multiprocessors. In *Principles of Distributed Computing*, pages 89–98. ACM, 2011.
- [39] J. Kari. Reversible cellular automata: From fundamental classical results to recent developments. *New Generation Computing*, 36(3):145–172, 2018.
- [40] S. Kuhn and I. Ulidowski. A calculus for local reversibility. In *Reversible Computation*, volume 9720 of *Lecture Notes in Computer Science*, pages 20–35. Springer, 2016.
- [41] S. Kuhn and I. Ulidowski. Local reversibility in a Calculus of Covalent Bonding. *Science of Computer Programming*, 151:18–47, 2018.
- [42] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.
- [43] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.
- [44] R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5(3):183–191, 1961.
- [45] I. Lanese, C. A. Mezzina, and F. Tiezzi. Causal-consistent reversibility. *Bulletin of the EATCS*, 114:121–139, 2014.
- [46] I. Lanese, N. Nishida, A. Palacios, and G. Vidal. CauDEr website. URL: <https://github.com/mistupv/cauder>.
- [47] I. Lanese, N. Nishida, A. Palacios, and G. Vidal. CauDEr: A causal-consistent reversible debugger for Erlang. In *FLOPS*, volume 10818 of *Lecture Notes in Computer Science*, pages 247–263. Springer, 2018.
- [48] I. Lanese, N. Nishida, A. Palacios, and G. Vidal. A theory of reversibility for Erlang. *Journal of Logical and Algebraic Methods in Programming*, 100:71–97, 2018.

- [49] I. Lanese, A. Palacios, and G. Vidal. Causal-consistent replay debugging for message passing programs. In *FORTE*, volume 11535 of *Lecture Notes in Computer Science*, pages 167–184. Springer, 2019.
- [50] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, 1987.
- [51] B. Lewis. Debugging backwards in time. *CoRR*, 2003.
- [52] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now?: An empirical study of bug characteristics in modern open source software. In *ASID*, pages 25–33. ACM, 2006.
- [53] J. Liu. Parallel discrete-event simulation. *Wiley Online Library*, 35:101, 2009.
- [54] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, pages 329–339. ACM, 2008.
- [55] C. Lutz. Janus: A time-reversible language. A letter to Dr. Landauer. 1986. <http://tetsuo.jp/ref/janus.pdf>.
- [56] H. C. Melgratti, C. A. Mezzina, and I. Ulidowski. Reversing P/T nets. In *COORDINATION 2019*, volume 11533 of *Lecture Notes in Computer Science*, pages 19–36. Springer, 2019.
- [57] C. A. Mezzina. On reversibility and broadcast. In *Reversible Computation*, volume 11106 of *Lecture Notes in Computer Science*, pages 67–83. Springer, 2018.
- [58] C. A. Mezzina, R. Schlatte, R. Glück, T. Haulund, J. Hoey, M. H. Cservenka, I. Lanese, T. Æ. Mogensen, H. Siljak, U. P. Schultz, and I. Ulidowski. Software and reversible systems: A survey of recent activities. In *Reversible Computation: Theory and Applications*, volume 12070 of *Lecture Notes in Computer Science*. Springer, 2020. To appear.
- [59] D. Mosberger. Memory consistency models. *ACM SIGOPS Operating Systems Review*, 27(1):18–26, 1993.
- [60] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *OSDI*, volume 8, pages 267–280, 2008.

- [61] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously recording program execution for deterministic replay debugging. *ACM SIGARCH Computer Architecture News*, 33(2):284–295, 2005.
- [62] N. Nishida, A. Palacios, and G. Vidal. A reversible semantics for Erlang. In *LOPSTR*, volume 10184 of *Lecture Notes in Computer Science*, pages 259–274. Springer, 2016.
- [63] K. Perumalla. *Introduction to Reversible Computing*. CRC Press, 2014.
- [64] I. Phillips and I. Ulidowski. Reversing algebraic process calculi. *Journal of Logic and Algebraic Programming*, 73(1):70 – 96, 2007.
- [65] I. Phillips and I. Ulidowski. Reversibility and asymmetric conflict in event structures. *Journal of Logical and Algebraic Methods in Programming*, 84(6):781 – 805, 2015. “Special Issue on Open Problems in Concurrency Theory”.
- [66] I. Phillips, I. Ulidowski, and S. Yuen. Modelling of bonding with processes and events. In *Reversible Computation*, Lecture Notes in Computer Science, pages 141–154. Springer, 2013.
- [67] I. C. C. Phillips and I. Ulidowski. Reversing algebraic process calculi. In *FOSSACS*, volume 3921 of *Lecture Notes in Computer Science*, pages 246–260. Springer, 2006.
- [68] I. C. C. Phillips, I. Ulidowski, and S. Yuen. A reversible process calculus and the modelling of the ERK signalling pathway. In *Reversible Computation*, volume 7581 of *Lecture Notes in Computer Science*, pages 218–232. Springer, 2012.
- [69] G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.
- [70] D. Quinlan and C. Liao. The ROSE Source-to-Source Compiler Infrastructure. In *Cetus Users and Compiler Infrastructure Workshop, in conjunction with PACT 2011*, October 2011.
- [71] M. Ronsse, K. D. Bosschere, and J. C. de Kergommeaux. Execution replay and debugging. In *Automated and Algorithmic Debugging workshop*, 2000.
- [72] M. Ronsse, M. Christiaens, and K. De Bosschere. Cyclic debugging using execution replay. In *Computational Science - ICCS 2001*, pages 851–860. Springer, 2001.
- [73] M. Russinovich and B. Cogswell. Replay for concurrent non-deterministic shared-memory applications. *ACM SIGPLAN Notices*, 31(5):258–266, 1996.

- [74] Y. Saito. Jockey: A user-space library for record-replay debugging. In *Automated Analysis-driven Debugging*, pages 69–76. ACM, 2005.
- [75] M. Schordan, D. R. Jefferson, P. D. B. Jr., T. Oppelstrup, and D. J. Quinlan. Reverse code generation for parallel discrete event simulation. In *Reversible Computation*, volume 9138 of *Lecture Notes in Computer Science*, pages 95–110. Springer, 2015.
- [76] M. Schordan, T. Oppelstrup, D. Jefferson, P. D. B. Jr., and D. J. Quinlan. Automatic generation of reversible C++ code and its performance in a scalable kinetic Monte-Carlo application. In *SIGSIM Principles of Advanced Discrete Simulation*, pages 111–122. ACM, 2016.
- [77] M. Schordan, T. Oppelstrup, D. R. Jefferson, and P. D. B. Jr. Generation of reversible C++ code for optimistic parallel discrete event simulation. *New Generation Computing*, 36(3):257–280, 2018.
- [78] U. P. Schultz. Reversible object-oriented programming with region-based memory management. In *Reversible Computation*, volume 11106 of *Lecture Notes in Computer Science*, pages 322–328. Springer, 2018.
- [79] U. P. Schultz and H. B. Axelsen. Elements of a reversible object-oriented language. In *Reversible Computation*, volume 9720 of *Lecture Notes in Computer Science*, pages 153–159. Springer, 2016.
- [80] M. Soeken, S. Frehse, R. Wille, and R. Drechsler. Revkit: A toolkit for reversible circuit design. *Multiple-Valued Logic and Soft Computing*, 18(1):55–65, 2012.
- [81] M. K. Thomsen and H. B. Axelsen. Interpretation and programming of the reversible functional language rfun. In *Implementation and Application of Functional Programming Languages*, pages 8:1–8:13. ACM, 2015.
- [82] M. K. Thomsen, H. B. Axelsen, and R. Glück. A reversible processor architecture and its reversible logic design. In *Reversible Computation*, pages 30–42. Springer, 2012.
- [83] T. Toffoli. Computation and construction universality of reversible cellular automata. *Journal of Computer and System Sciences*, 15(2):213 – 231, 1977.
- [84] T. Toffoli. Reversible computing. In *Automata, Languages and Programming*, pages 632–644. Springer, 1980.
- [85] I. Ulidowski, I. Phillips, and S. Yuen. Reversing event structures. *New Generation Computing*, 36(3):281–306, 2018.

- [86] Undo Software. UndoDB. Commercially available reversible debugger. <http://undo-software.com/>.
- [87] Undo Software. Increasing software development productivity with reversible debugging, 2014.
- [88] Valgrind website. URL: <http://www.valgrind.org/>.
- [89] C. J. Vieri. *Pendulum - a reversible computer architecture*. PhD thesis, Massachusetts Institute of Technology, 1995.
- [90] V. Vipindeep and P. Jalote. List of Common Bugs and Programming Practices to avoid them. 2005.
- [91] A.-M. Visan, K. Arya, G. Cooperman, and T. Denniston. URDB: A universal reversible debugger based on decomposing debugging histories. In *PLOS*, pages 1–5, 2011.
- [92] P. Vitányi. Time, space, and energy in reversible computing. In *Computing Frontiers*, pages 435–444. ACM, 2005.
- [93] G. Vulov, C. Hou, R. W. Vuduc, R. Fujimoto, D. J. Quinlan, and D. R. Jefferson. The backstroke framework for source level reverse computation applied to parallel discrete event simulation. In *WSC 2011*. IEEE, 2011.
- [94] T. Yokoyama, H. B. Axelsen, and R. Glück. Principles of a reversible programming language. In *Computing frontiers*, pages 43–54. ACM, 2008.
- [95] T. Yokoyama, H. B. Axelsen, and R. Glück. Reversible flowchart languages and the structured reversible program theorem. In *International Colloquium on Automata, Languages and Programming*, pages 258–270. Springer, 2008.
- [96] T. Yokoyama, H. B. Axelsen, and R. Glück. Towards a reversible functional language. In *Reversible Computation*, volume 7165 of *Lecture Notes in Computer Science*, pages 14–29. Springer, 2011.
- [97] T. Yokoyama and R. Glück. A reversible programming language and its invertible self-interpreter. In *Partial Evaluation and Semantics-based Program Manipulation*, pages 144–153. ACM, 2007.
- [98] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Elsevier Science, 2 edition, 2009.