SEMANTICS AND LOGICS FOR SIGNALS

by

MAXIM STRYGIN

A thesis submitted to the
University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
University of Birmingham
May 2014

Abstract

In operating systems such as Unix, processes can interact via signals. Signal handling resembles both exception handling and concurrent interleaving of processes. The handlers can be installed dynamically by the main program, but signals arrive nondeterministically; therefore, a handler may interrupt a program at any point. However, the interleaving of actions is not symmetric, in that the handler interrupts the main program, but not conversely. This thesis presents operational semantics and program logic for an idealized form of signal handling. To make signal handling logically tractable, we define handling to be block-structured. To reason about the interleaving of signal handlers, we adopt the idea of binary relations on states from rely-guarantee logics, imposing rely conditions on handlers. Given the one-way interleaving of signal handlers, the logic is less symmetric than rely-guarantee. We combine signal and exception handlers in the same language to investigate their interactions, specifically whether a handler can run more than once or is linearly used. We prove soundness of the program logic relative to a big-step operational semantics for signal handling. Then, we introduce and discuss reentrancy in various domains. Finally, we present our work towards logic with Reentrancy Linear Type System.

Dedication

I dedicate this thesis work to My Family. A special feeling of gratitude to my loving parents, Lyubov and Gennadiy, for their support and constant encouragement over the years. The warmest thanks to my precious wife, Alyona, for her patience, understanding and continuos support throughout my research and the whole life. I could not have done this without you.

Acknowledgments

I wish to thank my thesis committee members for all of their guidance, expertise and support through this process. Their discussion, ideas, and feedback have been invaluable. A special thanks to my supervisor Hayo Thielecke. His advice has been indispensable in many respects. I have been very much enjoying working with him and have always left his office encouraged and in good spirit. I would also like to acknowledge the financial support of the College of Engineering and Physical Sciences. Finally I would like to thank the School of Computer Science for allowing me to conduct my research and providing any assistance requested.

Contents

List of Figures

CHAPTER 1

INTRODUCTION

> "Interference is the essence of concurrency and it is what makes reasoning about concurrent programs difficult." [47]

In operating systems, and specifically Unix and its descendants, signals provide a simple and efficient, if rather low-level, means of interprocess communication [56, 87, 91, 88, 9, 50]. Put simply, a process can cause a branch of control in another process, causing it to run a signal handler in response to external events. A well known example is the `kill` signal telling a process to shut down (perhaps after first deallocating system resources, such as releasing memory). Signals resemble exceptions in that control jumps to a handler that can be installed by the program. Nonetheless, there are some significant differences. Whereas exceptions typically abort from the context in which they were thrown, the control flow returns to the interrupted code after a signal handler completes its execution. Whereas exceptions are triggered at specific points by the code itself, signals arrive nondeterministically. In the literature on control constructs and their semantics, signals have received less attention than exceptions, and far less than first-class continuations.

Exceptions have become amenable to semantic analysis by a focus on their key control features, while abstracting away from implementation details and restrictions (such as the entanglement of exceptions in C++ with the class hierarchy and memory management by destructors). For instance, the exceptions monad [70] gives a highly idealized account of exceptions as functions $A \to (B + E)$ that may either return normally with a B or raise an exception of type E.

One of the thesis goals was to address signal handling at a level of generality and abstraction comparable to that of other control constructs in the literature, idealizing where necessary and focusing on some key semantic features.

The main contributions of the thesis are as follows:

- We define an operational semantics for a language with both exceptions and signal handling with persistent and one-shot control flow semantics. A variant of this operational semantics has appeared in Workshop on Structured Operational Semantics 2012 [93]. The main difference between two versions is that we have weakened some restrictions on signal handlers in terms of interrupts from not blocked signals. This made our operational semantics closer to the real life implementations and gave a boost towards studying reentrancy. The most interesting feature of the operational semantics is the multiplicative way that one-shot signals are propagated, as the signal binding is split into two disjoint bindings when a semantics rule has two premises, for example in a sequential composition.

- We define an abstract stack machine for signal handlers to show the challenges one may encounter if he decides to implement a language with block structured exception and signal handling. We also compare how the idealized stack machine models features of real signal implementations in Unix like systems.

- To reason about concurrency explicitly, we define a program logic with specifications for signal handlers and exception context. In the logic, by contrast to signal binding splitting in semantics, the signal context is used additively, in that it is shared rather than split in the logic rules. We also adopt the notion of stability to address how exception and signal handlers with commands influence each other. We consider how the presented logic could be applied to address some of the idioms (such as invariant preserving, signal masking, and etc.) of signal usage.

- One of the main contributions of this thesis is a soundness proof of the logic with respect to the big-step operational semantics, because one may derive properties of

a program which do not hold if a logic is unsound. The proof proceeds by induction over the derivation of a program logic judgement. Towards the soundness proof, we impose the condition that all signal bindings respect the specification given by the whole signal context. As our language combines signal and exception handling, we introduce a form of stability condition between signal and exception contexts.

- We summarise the discussions about reentrancy from various domains and provide a glossary of the reentrancy related terms. To show how diverse the definitions of reentrancy could be, we compare the notion of reentrancy in Object Oriented and Procedural paradigms. We discuss and raise some questions about relations between reentrancy and notions like thread-safety, locks, stability, signal and exception handlers.

- We extend our language with local variables and functions to address reentrancy. Thus, we register functions as signals instead of commands what makes our language closer to the real life implementations. The argument passing in functions is imitated by use of global variables, whereas functions are classified in reentrant and non-reentrant ones.

  We define an extended logic and Reentrancy Linear Type System with functions, local variables, exception and signal handling. The designed Reentrancy Linear Type System ensures that non-reentrant functions are used at most once or not used at all in the environment with signals. We also raise some open-ended questions and discuss the variations of the logic rules updates such as potential stability checks elimination.

1.1 Block Structure and Control

Our most significant idealization of signal handlers is directly inspired by exceptions in contrast to the unstructured `longjmp` that exceptions were designed to replace [56, 87].

We define an idealized block-structured form of signal handling in which a signal handler is installed at the beginning of the block and uninstalled at the end. The idealized signal handling relates to `sigaction` the way exceptions related to `setjmp` and atomic `synchronized` blocks related to locking and unlocking. Furthermore, the big-step operational semantics perfectly suits for addressing signal handling with a block structured nature.

One may say that the big-step semantics is out of fashion, but as a counterexample to this statement year by year new papers are published [60, 58, 101, 15]. The last doubts could be dispelled by reading a short analysis provided by Charguéraud in [15].

1.2 One-side Interleaving and Concurrency

One may think of addressing one-sided interleaving with the same approach as complete interleaving. This is true to some extent, but there is important difference between them. The interaction between fully concurrent processes is symmetric, but there is no such symmetry between signals' body and handler. Only signals' handler may interrupt the body but not vice versa. This allows to simplify the approach/mechanism for addressing the signals handling. On the other hand, the general approach used for the fully concurrent interleaving might be not suitable, as interaction is nonsymmetric.

1.3 Rely-guarantee and Binary Relations

As signals are found in imperative languages, their interaction with shared state is of critical importance. The problem is interference, and so we adapt techniques from the theory of shared-variable concurrency. Figure 1.1 depicts the symmetric interleaving of concurrent processes compared to the one-sided interleaving of a process by its signal handlers. Dashed horizontal lines represent control flow; dotted vertical lines represent switches in the control flow due to interleaving. In both cases, the state σ_i that a process sees at some point could have been changed to some state σ_{i+1} by interleaved actions.

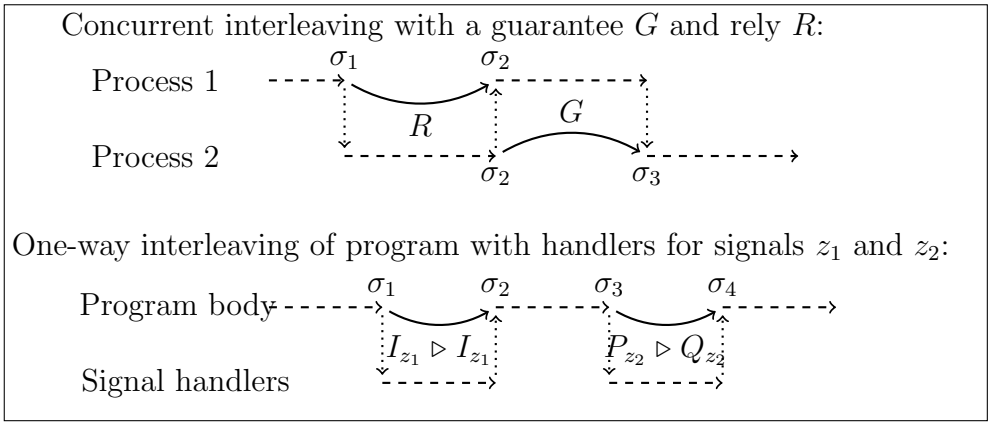

Figure 1.1: Concurrency vs signal handlers

These state changes need to be limited in some way, as otherwise no assumptions could be made by the process about the state, including resource invariants.

A key contribution of rely-guarantee logic [55, 17, 23] is to introduce *binary* relations on states, in addition to the unary predicates on states known from Hoare logic. Using such relations, we can express that a process *relies* on the interleaved state changes being contained in the relation, that is, (σ_i, σ_{i+1}). In rely-guarantee logic, the interaction between concurrent processes is symmetric, so that the rely of one process becomes (part of) the guarantee of another.

For signal handlers, by contrast, there is no such symmetry, as the handlers interrupt the body, but not conversely. In that sense, the situation is greatly simplified, and we only need rely predicates. On the other hand, the set of installed signal handlers changes over time. In Figure 1.1, a program body relies that the state change (σ_1, σ_2) will satisfy $(I_{z_1} \triangleright I_{z_1})$, and the state change (σ_3, σ_4) will satisfy $(P_{z_2} \triangleright Q_{z_2})$. That means if I_{z_1} holds in σ_1, then I_{z_1} holds in σ_2. Analogously, if P_{z_2} holds in σ_3, then Q_{z_2} holds in σ_4. Further details of the program logic, including definition of the binary relation $\triangleright$, are covered in Chapter 5.

1.4 Linear Use and Resources

Attacks on software security published by Zalewski using malicious signal handling [27, 21] were the initial motivation for logic and semantics in this thesis. A critical ingredient in Zalewski's exploits is the idea to cause the same handler to run twice and thereby corrupt a resource. In other words, some signal handlers may be safe only as long as they are "linearly used".

Signal handlers can have two different control flow semantics, which we call *persistent* and *one-shot*. A persistent signal handler can be run any number of times as long as it is installed. By contrast, a one-shot signal handler can be run at most once, as it becomes automatically uninstalled after being run the first time. In Unix, a system call for installing handlers takes a parameter that determines which of these behaviours is chosen.

In our program logic, we use a context Σ to keep track of the two kinds of signal bindings, and the difference between the two forms of signal handlers is reflected in the specifications we give for them. For a persistent signal handler, we associate an invariant I to the signal that should hold before and after the signal handler runs. This invariant is similar to a loop invariant, where we also cannot statically determine how often the loop runs. For a one-shot handler, we associate a precondition P and a postcondition Q with the signal.

Due to the one-shot semantics, we can assume that the state, the one-shot signal handler finds itself, satisfies P rather than Q, which would not hold if the handler could run multiple times, as it can for a persistent binding. For example, in attacks [27, 21] where a signal is maliciously sent twice to trigger a double free, the precondition P would state that some memory is allocated, whereas Q would state that it has been freed. The problem could be addressed by using a one-shot signal or by exiting at the end of a persistent signal handler to prevent the handler from running again after its precondition has been consumed.

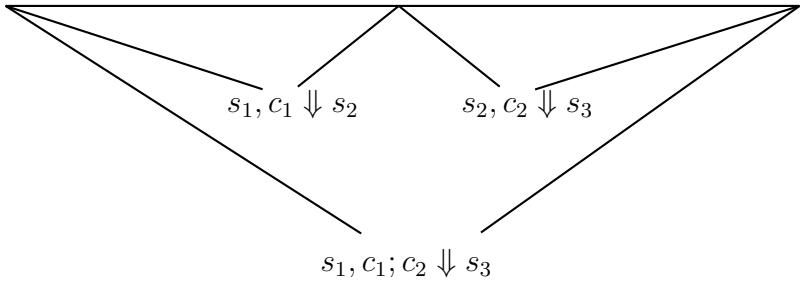

Figure 1.2: Big step trees

1.5 Big-step Semantics and Exceptions

For the operational semantics, we define a big-step semantics. This style of semantics appears particularly apt for the signals and exceptions kind of constructs. Essentially, the meaning of a block becomes a subtree of a larger derivation tree, which is convenient for keeping track of pre- and post-states (Figure 1.2).

The big-step semantics is a perfect choice to address block structured exceptions. With a notion of block, there is no need for a stack to keep track of installed exceptions. Furthermore, exception deinstallation is also handled by free as scope is clearly limited by block in the big-step semantics. Thus, the big-step semantics becomes a perfect platform for studying signals, exceptions and their interaction. Also, the big-step semantics could be perfectly related to a Hoare logic. A Hoare triple has a form $\{P\}\, c\, \{Q\}$ with pre-condition P and post-condition Q for a command c. In operational semantics, an evaluation of the command c is described as $s_1, c \Downarrow s_2$ with initial state s_1 and final state s_2. Thus, to express that state s_1 satisfies precondition P and s_2 satisfies postcondition Q, one may write $s_1 \models P$ and $s_2 \models Q$.

1.6 Motivation

The domain of control structure has been extensively studied in sequential and mostly functional languages. For instance, there are two workshops on continuations alone

(*ACM-SIGPLAN Continuation Workshop*, *Theory and Practice of Delimited Continuations Workshop*). Concurrency is another actively developing direction. For example, series of *Concurrency Workshops* since 2009 that is supported by the Cambridge, Dublin, London and Newcastle research groups. In concurrency, very little control structure is considered, typically just block structured control like `if` and `while`. One of the goals of our work was to combine and generalize these research directions thereby overcoming some of their limitations.

It is well known that the single threaded architecture and implementations are less efficient than concurrent ones. On the other hand, creating and maintaining concurrent programs is a complex task. The easiest way to handle concurrency is applying primitive blocking using locks. Unfortunately, it would not allow revealing full potential of concurrency. Therefore, many complex approaches of handling concurrency has been developed and currently supported by different research schools.

Notion of signals in concurrency is another approach that was not well addressed in the past. The most closest ideas to this approach has been addressed by Zhong Shao *et al.* in his works regarding interrupts [30, 31]. Specifically, in [30, 32] Zhong Shao *et al.* presents program logic that allows reasoning about programs and dealing with interrupts and concurrency simultaneously. Moreover, being inspired by works regarding local-reasoning [75], Shao *et al.* also addressed a sharing of the state between threads and handlers. In another work [31] authors studied the real application of early developed methodology. It is mostly oriented to program verification domain, but reasoning is based on logic from [30]. We can say that the notion of interrupts is more hardware oriented and authors used a separation logic to reason about it. However, they have not adopted the notion of rely-guarantee reasoning in their work. On the other hand, we are addressing the notion of signals which is more software oriented, as they work on a level of interprocess communication.

In the book [23], de Roever relates reactive sequences and Aczel traces. In one branch of our work, we have worked on a hybrid of both approaches and defined Aczel(2) traces.

Afterwards, we have extended this model with rely-guarantee logic. One can note that there is another approach to trace semantics (where traces are sequences of states) supported by Stephen Brooks [13]. The open question is how his approach could be related to de Roever and Cliff Jones styles.

Histories and traces are rather similar concepts where traces are state oriented and histories are event oriented. The signal control structures could be expressed in terms of histories. So, the model for the signals could be based on traces and operational semantics. As we address signals in concurrency, we have to define the rely-guarantee logic for this model if we want to reason about interference explicitly.

1.7 Outline of the Thesis

We define an idealized language for signal handling and explain some design choices we made in its semantics and program logic in Chapter 2. The meaning of our language is defined via a big-step semantics in Chapter 3. An abstract machine for signal handling, as an alternative operational semantics, is shown in Chapter 4. We present a program logic in Chapter 5; it adapts the idea of binary rely predicates from concurrency to signal handling. We show the proof for Logic Soundness in Chapter 6. The notion of a nested binding and its support by the operational semantics and the logic are discussed in Chapter 7. Introduction to reentrancy is given in Chapter 8. Work towards Reentrancy Linear Type System is explained in Chapter 9. In Chapter 10, a literature review for publications in the relevant research domains is given. Chapter 11 concludes with related work and directions for future work.

CHAPTER 2

LANGUAGE DESIGN AND SIGNALS

Before giving the formal definition of our operational semantics and program logic, we introduce the language constructs with their intended meaning, as well as design choices and simplifying assumptions. In this chapter, we also present exception and signal handling conventions that determine their interaction with other language constructs and each other.

2.1 Base Language

As is usual for program logic, we start from a small imperative base language. It contains sequential composition, `while` loops and assignment.

$$
\begin{array}{rcll}
c & ::= & \texttt{while}\, E\, \texttt{do}\, c & \text{(while construct)} \\
& | & a & \text{(Atomic command)} \\
& | & x := E & \text{(Assignment)} \\
& | & c_1; c_2 & \text{(Sequential composition)}
\end{array}
$$

This language has a standard semantics in terms of how a command c changes the state s_1 into a new state s_2. In a big-step operational semantics, the form of such judgements is

$$s_1, c \Downarrow s_2$$

2.2 Validity and Soundness

The standard correctness criterion for Hoare logic relates the pre- and postconditions of the program logic to the states before and after a big-step evaluation.

A Hoare triple

$$\{P\}\, c\, \{Q\}$$

is called *valid* if the following implication holds. If s_1 and s_2 are states such that

$$s_1 \models P \text{ and } s_1, c \Downarrow s_2$$

then

$$s_2 \models Q$$

To prove soundness of a program logic, one needs to prove that each Hoare triple that can be derived using the program logic rules is in fact valid according to the operational semantics.

When we add additional structure, such as signals and exceptions, the notion of validity of triples has to be extended accordingly, and the same goes for soundness.

Various styles of small-step and big-step semantics have been defined in the literature. Big-step semantics, the style we will use, is particularly simple for structured control flow, such as sequential composition.

2.3 Adding Exceptions

Exception throwing and handling is easy to add to a big-step operational semantics. A classic example of such semantics is the Definition of Standard ML [69], whose style we will follow.

2.3.1 Exception Operations

We add exceptions to the big-step semantics via a new form of judgement: given an old state s_1, the command c throws the exception e and produces a new state s_2.

$$s_1, c \Uparrow e, s_2$$

The base language is extended with two new constructs, for which we adopt a syntax loosely based on Java. An exception e is thrown (or raised in ML terminology) by the command $\texttt{throw}\, e$. An exception is handled (or caught) by an exception block of the form $\texttt{try}\ c_B\ \texttt{handle}\ e\ \texttt{by}\ c_h$. Any e thrown inside c_B will be handled by c_h. The syntax of our language is accordingly extended:

$$c ::= \ldots \mid \texttt{throw}\, e \mid \texttt{try}\ c_B\ \texttt{handle}\ e\ \texttt{by}\ c_h$$

The idea behind the operation semantics is that $\texttt{throw}\, e$ produces judgements of the "exceptional" form using $\Uparrow e$, whereas handling may turn a judgement of that form into one that terminates normally by way of $\Downarrow$. More precisely, we have the usual rule for exception raising:

$$\frac{}{s, \texttt{throw}\, e \Uparrow e, s}$$

For exception handling, there are the following rules:

$$\frac{s_1, c_B \Uparrow e, s_2 \qquad s_2, c_h \Downarrow s_3}{s_1, \texttt{try}\ c_B\ \texttt{handle}\ e\ \texttt{by}\ c_h \Downarrow s_3} \qquad \frac{s_1, c_B \Downarrow s_2}{s_1, \texttt{try}\ c_B\ \texttt{handle}\ e\ \texttt{by}\ c_h \Downarrow s_2}$$

$$\frac{s_1, c_B \Uparrow e, s_2 \qquad s_2, c_h \Uparrow e_2, s_3}{s_1, \texttt{try}\ c_B\ \texttt{handle}\ e\ \texttt{by}\ c_h \Uparrow e_2, s_3} \qquad \frac{s_1, c_B \Uparrow e_2, s_2 \qquad e_2 \neq e}{s_1, \texttt{try}\ c_B\ \texttt{handle}\ e\ \texttt{by}\ c_h \Uparrow e_2, s_2}$$

If the body c_B of an exception block

$$\texttt{try}\ c_B\ \texttt{handle}\ e\ \texttt{by}\ c_h$$

raises the exception e, then the handler is run. Otherwise, the handler has no effect.

We use $\Uparrow$ to indicate an exceptional form of the judgement, and find it rather elegant. There is another approach, where authors [15] use only $\Downarrow$ for both normal and exceptional judgements. The trick is that expression (command in our notation) evaluates into the behaviour b, which in turn denotes if it terminates normally or with a raised exception.

2.3.2 Exception Convention

In addition to the rules for the operations themselves, we also need to specify how the propagation of exceptions interacts with the other constructs of the language: this propagation will be done with the *exception convention* from the Definition of Standard ML. If the j-th premise of a big-step rule raises an exception, and the premises to its left do not, then the conclusion of the rule raises the same exception, and with the same state.

More precisely, suppose there is a big-step rule of the form

$$\frac{\ldots c_1 \Downarrow s_1 \ldots c_j \Downarrow s_j \ldots c_n \Downarrow s_n}{\ldots c \Downarrow s}$$

Then we implicitly extend this case to propagating exception by adding a rule

$$\frac{\ldots c_1 \Downarrow s_1 \ldots c_j \Uparrow e, s_j}{\ldots c \Uparrow e, s_j}$$

To illustrate the exception convention, we consider how exceptions are propagated in a sequential composition $c_1; c_2$.

$$\frac{s_1, c_1 \Uparrow e, s_2}{s_1, (c_1; c_2) \Uparrow e, s_2} \qquad \frac{s_1, c_1 \Downarrow s_2 \qquad s_2, c_2 \Uparrow e, s_3}{s_1, (c_1; c_2) \Uparrow e, s_3}$$

Intuitively, the first command c_1 may raise an exception, in which case the second command c_2 is not run at all. Alternatively, c_1 may terminate normally, and c_2 may raise an exception. In either case, the combined command raises the same exception.

2.3.3 Exception Contexts

For a language without control constructs, program logic judgements need only address successful termination, without any thrown exceptions. If we add exceptions, the outcome of evaluations of the form

$$s_1, c \Uparrow e, s_2$$

also need to be addressed. Such cases require us to associate a postcondition Q_e that should hold after the corresponding exception e has been thrown.

We extend the Hoare logic with an *exception context* η of the form

$$\eta \;=\; e_1 : Q_1, e_2 : Q_2, \ldots, e_n : Q_n$$

The form of a program logic judgement with an exception context becomes:

$$\{P\}\, c\, \{Q\}\ \texttt{throws}\ e_1 : Q_1, e_2 : Q_2, \ldots, e_n : Q_n$$

One needs to ensure that the precondition $\eta(e_j) = Q'_j$ for an exception e_j holds immediately before the exception is thrown (throwing by itself does not change the state). That way, it holds at the beginning of the handler.

The semantics of exceptions is fairly well understood, and it is greatly simplified by the fact that exceptions are block structured. The more primitive non-local jumps in C (given via the library functions `setjmp()` and `longjmp()`) would be much harder to formalize, both operationally and logically. In fact, they would present many of the challenges of self-modifying code pointers, such as recursion through the store.

2.4 Adding Block-structured Signals

The main construct we aim to address is signal handling. Signal handling is a form of interprocess communication, so that for full generality we would have to address the

concurrent interaction between a signal sending and a signal handling process. To keep the semantics and the program logic as simple as possible, we address only the *handling* part of the signal mechanism, while the truly concurrent interaction between sender and receiver is left for future work. Rather than modelling the signal sender explicitly, only the point of view of the process receiving the signals will be assumed, so that signals arrive nondeterministically, causing handlers to run unpredictably. In the authors' view, this focus on signal *handling* still presents sufficient programming and semantics challenges. First, the nondeterministic interference by signal handlers leads to the need to preserve resource invariants, much as interference between concurrent processes. Moreover, the assumptions a programmer can make about the delivery of signals are very weak, even if there is a specification of the sender's behaviour (which there usually is not). In the worst case, the signal sender may even be malicious, sending signals with the sole intent of causing damage via the actions of the signal handlers. In that sense, a nondeterministic sender is a worst-case but realistic assumption that the signal receiver has to be able to cope with.

Signal implementation make it possible to specify that a handler should run at most once, becoming uninstalled after running for the first (and only) time. Therefore, we will make a distinction between one-shot signal handlers and persistent ones.

As implemented in Unix, a signal handler is (a pointer to) a function, which may be associated to a signal name via a system call. The association between the signal and the function pointer remains until it is overwritten by another such system call. Semantically, it behaves like a pointer assignment rather than a block-structured binding. While it would certainly be possible to construct an operational semantics for this behaviour, pointer assignment makes the logic much more difficult to handle due to the recursion through the store. Such recursion can be handled logically, but requires sophisticated techniques, such as those developed by Reus and coauthors [83].

Moreover, this logical complexity is largely extraneous to the control flow presented by signals. Hence we assume a simplified and structured form of signal handling, where

a signal handler is only ever installed at the beginning of a block and uninstalled at the end of it, just as an exception handler is.

A few C examples of what may be addressed with block structured signals is given below. Please note that in the current C implementation there is no implicit signal handler uninstallation. Thus, the signal handler is not uninstalled but reset to the default value for a particular signal when it is no longer needed.

Preparation: We add two signal handlers.

```
void signal_handler_1(int signal) {
    printf("First handler received signal %d\n", signal);
}
void signal_handler_2(int signal) {
    printf("Second handler received signal %d\n", signal);
}
```

Case 1: Two signals are installed and uninstalled one after another

```
signal(SIGTERM, signal_handler_1); // install signal_handler_1
// a block of code
signal(SIGTERM, SIG_DFL); // reset (uninstall) signal_handler_1
signal(SIGINT, signal_handler_2); // install signal_handler_2
// a block of code
signal(SIGINT, SIG_DFL); // reset (uninstall) signal_handler_2
```

Case 2: Two nested signals are installed and uninstalled

```
signal(SIGTERM, signal_handler_1); // install signal_handler_1
signal(SIGINT, signal_handler_2); // install signal_handler_2
// a block of code
signal(SIGINT, SIG_DFL); // reset (uninstall) signal_handler_2
signal(SIGTERM, SIG_DFL); // reset (uninstall) signal_handler_1
```

Code where signal installing and uninstalling do not match up (Case 3) cannot be addressed with our idealization of signal handling.

Case 3: Two signal handlers are uninstalled in the same order they were installed.

```
signal(SIGTERM, signal_handler_1); // install signal_handler_1
signal(SIGINT, signal_handler_2); // install signal_handler_2
// a block of code
signal(SIGTERM, SIG_DFL); // reset (uninstall) signal_handler_1
signal(SIGINT, SIG_DFL); // reset (uninstall) signal_handler_2
```

2.4.1 Operational Semantics

In the operational semantics, the evaluation of a command c starting from a state s_1 will now take place relative to a signal binding. Moreover, the signal binding is subdivided into two parts: persistent signals S, and one-shot signals O. Persistent handlers may run any number of times during the evaluation of the command c, whereas one-shot handlers may run at most once. The form of a big-step judgement with signal bindings is:

$$S; O \Vdash s_1, c \Downarrow s_2$$

Note that the signal binding behaves like an environment (for variables bound via `let`) rather than a mutable state (for variables updated via `:=`). The judgement produces an updated state s_2, but it does not update S or O.

Analogous to binding an exception handler, we have two binding constructs for signals: one for persistent and one for one-shot handlers, where z is a signal name, c_B is a command, and c_h is a handler command.

$$\texttt{bind}\, z\, \texttt{to}\, c_h\, \texttt{in}\, c_B \quad \text{and} \quad \texttt{bind/1}\, z\, \texttt{to}\, c_h\, \texttt{in}\, c_B$$

To support signal disabling in a scope, we introduce a blocking construct for signals:

$$\texttt{block}\, z\, \texttt{in}\, c_B$$

Note that there is no need for an analogue of $\texttt{throw}\, e$ (a command that throws an exception e), as we assume that signals arrive nondeterministically from other, unspecified processes. The idea of using two contexts with a binder for each is loosely inspired by Barber and Plotkin's Dual Intuitionistic Linear Logic (DILL) [6].

2.4.2 Signal Handling Convention

Signals may arrive at any time. Thus, signal handler might be processed before

$$\frac{S(z) = c_h \qquad S - z; O \Vdash s_1, c_h \Downarrow s_2 \qquad S; O \Vdash s_2, c_B \Downarrow s_3}{S; O \Vdash s_1, c_B \Downarrow s_3}$$

or after the command c_B

$$\frac{S; O \Vdash s_1, c_B \Downarrow s_2 \qquad S(z) = c_h \qquad S - z; O \Vdash s_2, c_h \Downarrow s_3}{S; O \Vdash s_1, c_B \Downarrow s_3}$$

2.4.3 Program Logic with Specifications for Signal Handlers

When we augment our language with signal handling, the program logic needs to be extended with specifications of all the handlers that may interfere with the given command. These specifications limit how the handlers can interfere with the body of the code, making them the analogue of a rely condition in rely-guarantee logic. The format of a judgement becomes

$$\Sigma \vdash \{P\}\, c\, \{Q\}$$

Here Σ contains the specification of persistent signal handlers, which may run any number of times, and specification for one-shot signal handlers, which may run at most once. Further details for the program logic are given in Chapter 5.

A key difference between persistent and one-shot handler specifications is given by transitivity. Whereas persistent signal handler specifications are automatically transitive, one-shot handlers need not be transitive. As the one-shot handler can run at most once, its action may invalidate its precondition, so that the relation between states is not transitive. For example, a handler h changes the initial state in which P holds in to the state in which Q holds. As the precondition P is invalidated, the handler h cannot run anymore.

$$P \xrightarrow{h} Q$$

Transitivity is vital for the persistent signal handlers, because it implies that multiple executions of a handler does not invalidate the associated invariant. For example, a handler h changes the initial state in which I holds in to the state in which I still holds.

$$I \xrightarrow{h} I$$

Thus, multiple runs of the handler h could be presented as follows.

$$I \xrightarrow{h^*} I$$

CHAPTER 3

OPERATIONAL SEMANTICS

In this chapter, we define an operational semantics for our base language with both signal and exception handling. We explain in detail auxiliary definitions for the operational semantics, evaluation of expressions and big-step rules in particular. To show how the big-step rules could be used and to represent capabilities of the defined operational semantics, a set of examples with explanation is given. In this chapter we also discuss an interaction between exception and signal handling in terms of their priority, which is technically a design choice.

3.1 Block Structured Signals and Exceptions

The details of an operational semantics first appeared in [93], where we compare semantics in a big-step style to an abstract machine, which is closer to implementations of signal and exception handlers using stacks. It should be noted, that in previous work [93] all signal handlers were uninterruptible. In the current version, a restriction has been weakened; thus, only a signal linked to the running signal handler is blocked, where the rest could interrupt the running handler. In terms of signal handlers, they are not reentrant, as they cannot interrupt themselves. However, the reentrancy of code is not limited in any way, as two signals could be bound to the same signal handler; a particular block of code in the main program could be interrupted by a signal handler with the same block of code.

Definition 3.1.1 The syntax of the language with signal and exception handling is given in Figure 3.1, where e ranges over the primitive set of exceptions.

Some auxiliary definitions will be required for the operational semantics. For a partial function f, we write $f\,[\,x \mapsto v\,]$ for the function that maps x to v and coincides with f on all other arguments. In particular, we use this notation for updating states or signal bindings. We write $\mathrm{dom}(f)$ for the domain of definition of a partial function. For $x \in \mathrm{dom}(f)$, we write $f - x$ for the restriction of f to $(\mathrm{dom}(f) \setminus \{x\})$. A signal binding is a finite partial function from signal names z to commands c. We will need a partial operation on signal bindings. In fact, this definition is the same as the separating conjunction from separation logic [86].

Please note, that the assignment is an instance of the atomic command. If one would like to add a new atomic command to the language, that command should be lifted into the signal context. Despite the general requirements for the atomic commands, they must not manipulate with the signal context in any way.

$$\frac{s_1, a \Downarrow s_2}{S; O \Vdash s_1, a \Downarrow s_2}$$

Definition 3.1.2 Given two signal bindings O_1 and O_2, we define a partial operation $*$ as follows:

- If $\mathrm{dom}(O_1) \cap \mathrm{dom}(O_2) = \emptyset$, we write $O_1 * O_2$ for $O_1 \cup O_2$.
- If $\mathrm{dom}(O_1) \cap \mathrm{dom}(O_2) \neq \emptyset$, then $O_1 * O_2$ is undefined.

It is this splitting of a signal binding, analogous to the heap-splitting of separation logic, that gives one-shot behaviour to signals. Specifically, in a sequential composition $(c_1; c_2)$, the one-shot signals are split non-deterministically between the commands c_1 and c_2. Moreover, every time a one-shot signal arrives and is handled, it is removed from the one-shot binding O. Thus a one-shot signal may never be handled twice.

Commands

c	::=	$\texttt{while}\, E_B\, \texttt{do}\, c$	(while construct)
	|	$\texttt{if}\, E_B\, \texttt{then}\, c_1\, \texttt{else}\, c_2$	(if else construct)
	|	a	(Atomic command)
	|	$x := E$	(Assignment)
	|	$x{+}{+}$	(Increment)
	|	$x{-}{-}$	(Decrement)
	|	$c_1; c_2$	(Sequential composition)
	|	$\texttt{throw}\, e$	(Exception throwing)
	|	$\texttt{try}\, c_B\, \texttt{handle}\, e\, \texttt{by}\, c_h$	(Exception handling)
	|	$\texttt{block}\, z\, \texttt{in}\, c$	(Blocking of the signals)
	|	$\texttt{bind}\, z\, \texttt{to}\, c_z\, \texttt{in}\, c_B$	(Binding of the persistent signal handler)
	|	$\texttt{bind/1}\, z\, \texttt{to}\, c_z\, \texttt{in}\, c_B$	(Binding of the one-shot signal handler)

Expressions

E	::=	x	(Variables)
	|	$E_I \quad \mid \quad E_B$	(Int and Bool expressions)
E_I	::=	n	(Integers)
	|	$E_I + E_I \mid E_I - E_I \mid \ldots$	(Basic arithmetic operations)
E_B	::=	$\texttt{true} \mid \texttt{false}$	(Booleans)
	|	$E_I \leq E_I \mid E_I > E_I \mid \ldots$	(Basic arithmetic operations)

Figure 3.1: The syntax of the language

Definition 3.1.3 Given two signal bindings S and O, the form of a big-step judgement is either

$$S; O \Vdash s_1, c \Downarrow s_2$$

for normal termination, or

$$S; O \Vdash s_1, c \Uparrow e, s_2$$

for exception throwing.

The full list of big-step rules is given in Figure 3.2, Figure 3.3 and Figure 3.4. The exception convention could be assumed implicitly; therefore, the list of big-step rules might be shortened.

Standard Hoare logic uses only a first order, but Turing complete, programming language, with constructs like sequential composition and while. Functions could be added, but are to some extent orthogonal to our aims. The problem of interleaving of a handler and its body already arises even if the body is just a sequential composition of assignments and the handler is a single assignment. For example:

$$\texttt{bind}\, z\, \texttt{to}\, (x := 0)\, \texttt{in}\, (x := 1\, ;\, y := x)$$

3.2 Evaluation of Expressions

In our language, evaluation of expressions is fairly standard. Variables are evaluated via the following rule:

$$\frac{s_1(x) = v}{s_1 \Vdash x \Downarrow v}$$

A corresponding value for the variable x is stored in a state s_1. Rules for the arithmetic operations are analogous to each other. The rule for addition is given below:

$$\frac{s_1 \Vdash E_I \Downarrow v_1 \qquad s_1 \Vdash E'_I \Downarrow v_2 \qquad v = v_1 + v_2}{s_1 \Vdash E_I + E'_I \Downarrow v}$$

$$\frac{}{S; O \Vdash s, \texttt{throw}\, e \Uparrow e, s}\ (\textsc{Throw})$$

$$\frac{S; O_1 \Vdash s_1, c_B \Uparrow e_k, s_2 \quad S; O_2 \Vdash s_2, c_h \Downarrow s_3}{S; O_1 * O_2 \Vdash s_1, \texttt{try}\ c_B\ \texttt{handle}\ e_k\ \texttt{by}\ c_h \Downarrow s_3}\ (\textsc{Handl})$$

$$\frac{S\,[\,z \mapsto c_h\,]; O \Vdash s_1, c_B \Downarrow s_2}{S; O \Vdash s_1, \texttt{bind}\, z\, \texttt{to}\, c_h\, \texttt{in}\, c_B \Downarrow s_2}\ (\textsc{PerSigBind})$$

$$\frac{S\,[\,z \mapsto c_h\,]; O \Vdash s_1, c_B \Uparrow e_1, s_2}{S; O \Vdash s_1, \texttt{bind}\, z\, \texttt{to}\, c_h\, \texttt{in}\, c_B \Uparrow e_1, s_2}\ (\text{PSB2})$$

$$\frac{S; O\,[\,z \mapsto c_h\,] \Vdash s_1, c_B \Downarrow s_2}{S; O \Vdash s_1, \texttt{bind/1}\, z\, \texttt{to}\, c_h\, \texttt{in}\, c_B \Downarrow s_2}\ (\textsc{OneSigBind})$$

$$\frac{S; O\,[\,z \mapsto c_h\,] \Vdash s_1, c_B \Uparrow e_1, s_2}{S; O \Vdash s_1, \texttt{bind/1}\, z\, \texttt{to}\, c_h\, \texttt{in}\, c_B \Uparrow e_1, s_2}\ (\text{OSB2})$$

$$\frac{S - z; O - z \Vdash s_1, c \Downarrow s_2}{S; O \Vdash s_1, \texttt{block}\, z\, \texttt{in}\, c \Downarrow s_2}\ (\textsc{SignBlock})$$

$$\frac{S - z; O - z \Vdash s_1, c \Uparrow e_1, s_2}{S; O \Vdash s_1, \texttt{block}\, z\, \texttt{in}\, c \Uparrow e_1, s_2}\ (\text{SB2})$$

$$\frac{S; O_1 \Vdash s_1, c_1 \Downarrow s_2 \quad S(z) = c_h \quad S - z; O_2 \Vdash s_2, c_h \Downarrow s_3}{S; O_1 * O_2 \Vdash s_1, c_1 \Downarrow s_3}\ (\text{PSH-v2})$$

$$\frac{S(z) = c_h \quad S - z; O_1 \Vdash s_1, c_h \Downarrow s_2 \quad S; O_2 \Vdash s_2, c_1 \Downarrow s_3}{S; O_1 * O_2 \Vdash s_1, c_1 \Downarrow s_3}\ (\text{PSH2-v2})$$

$$\frac{S; O_1 - z \Vdash s_1, c_1 \Downarrow s_2 \quad O_1 * O_2(z) = c_h \quad S; O_2 - z \Vdash s_2, c_h \Downarrow s_3}{S; O_1 * O_2 \Vdash s_1, c_1 \Downarrow s_3}\ (\text{OSH-v2})$$

$$\frac{O_1 * O_2(z) = c_h \quad S; O_1 - z \Vdash s_1, c_h \Downarrow s_2 \quad S; O_2 - z \Vdash s_2, c_1 \Downarrow s_3}{S; O_1 * O_2 \Vdash s_1, c_1 \Downarrow s_3}\ (\text{OSH22})$$

Figure 3.2: Big-step semantic rules for exceptions and signal handling Part1

$$\frac{S; O_1 \Vdash s_1, c_1 \Downarrow s_2 \quad S; O_2 \Vdash s_2, c_2 \Downarrow s_3}{S; O_1 * O_2 \Vdash s_1, (c_1 \,;\, c_2) \Downarrow s_3}\ (\textsc{SeqComp})$$

$$\frac{S; O \Vdash s_1, c_1 \Uparrow e_1, s_2}{S; O \Vdash s_1, (c_1 \,;\, c_2) \Uparrow e_1, s_2}\ (\text{SC3})$$

$$\frac{S; O_1 \Vdash s_1, c_1 \Downarrow s_2 \quad S; O_2 \Vdash s_2, c_2 \Uparrow e_1, s_3}{S; O_1 * O_2 \Vdash s_1, (c_1 \,;\, c_2) \Uparrow e_1, s_3}\ (\text{SC2})$$

$$\frac{s_1 \Vdash E \Downarrow v}{S; O \Vdash s_1, x := E \Downarrow s_1\,[\,x \mapsto v\,]}\ (\textsc{Assignment})$$

$$\frac{s_1, a \Downarrow s_2}{S; O \Vdash s_1, a \Downarrow s_2}\ (\textsc{Atomic})$$

$$\frac{s_1 \models \neg E_B}{S; O \Vdash s_1, \texttt{while}\, E_B\, \texttt{do}\, c \Downarrow s_1}\ (\textsc{WhileFalse})$$

$$\frac{s_1 \models E_B \qquad S; O \Vdash s_1, c \Uparrow e, s_2}{S; O \Vdash s_1, \texttt{while}\, E_B\, \texttt{do}\, c \Uparrow e, s_2}\ (\textsc{WhileTrue2})$$

$$\frac{s_1 \models E_B \qquad S; O_1 \Vdash s_1, c \Downarrow s_2 \qquad S; O_2 \Vdash s_2, \texttt{while}\, E_B\, \texttt{do}\, c \Downarrow s_3}{S; O_1 * O_2 \Vdash s_1, \texttt{while}\, E_B\, \texttt{do}\, c \Downarrow s_3}\ (\textsc{WTrue})$$

$$\frac{s_1 \models E_B \qquad S; O \Vdash s_1, c_1 \Downarrow s_2}{S; O \Vdash s_1, \texttt{if}\, E_B\, \texttt{then}\, c_1\, \texttt{else}\, c_2 \Downarrow s_2}\ (\textsc{IfTrue})$$

$$\frac{s_1 \models \neg E_B \qquad S; O \Vdash s_1, c_2 \Downarrow s_3}{S; O \Vdash s_1, \texttt{if}\, E_B\, \texttt{then}\, c_1\, \texttt{else}\, c_2 \Downarrow s_3}\ (\textsc{IfFalse})$$

$$\frac{s_1 \models E_B \qquad S; O \Vdash s_1, c_1 \Uparrow e_1, s_2}{S; O \Vdash s_1, \texttt{if}\, E_B\, \texttt{then}\, c_1\, \texttt{else}\, c_2 \Uparrow e_1, s_2}\ (\text{IfT2})$$

$$\frac{s_1 \models \neg E_B \qquad S; O \Vdash s_1, c_2 \Uparrow e_1, s_3}{S; O \Vdash s_1, \texttt{if}\, E_B\, \texttt{then}\, c_1\, \texttt{else}\, c_2 \Uparrow e_1, s_3}\ (\text{IfF2})$$

Figure 3.3: Big-step semantic rules for exceptions and signal handling Part2

$$\frac{S; O_1 \Vdash s_1, c_1 \Downarrow s_2 \quad S(z) = c_h \quad S - z; O_2 \Vdash s_2, c_h \Uparrow e_1, s_3}{S; O_1 * O_2 \Vdash s_1, c_1 \Uparrow e_1, s_3} \text{(PSH1B)}$$

$$\frac{S(z) = c_h \quad S - z; O \Vdash s_1, c_h \Uparrow e_1, s_2}{S; O \Vdash s_1, c_1 \Uparrow e_1, s_2} \text{(PSH2A)}$$

$$\frac{S(z) = c_h \quad S - z; O_1 \Vdash s_1, c_h \Downarrow s_2 \quad S; O_2 \Vdash s_2, c_1 \Uparrow e_1, s_3}{S; O_1 * O_2 \Vdash s_1, c_1 \Uparrow e_1, s_3} \text{(PSH2B)}$$

$$\frac{S; O - z \Vdash s_1, c_1 \Uparrow e_1, s_2}{S; O \Vdash s_1, c_1 \Uparrow e_1, s_2} \text{(OSH1)}$$

$$\frac{S; O_1 - z \Vdash s_1, c_1 \Downarrow s_2 \quad O_1 * O_2(z) = c_h \quad S; O_2 - z \Vdash s_2, c_h \Uparrow e_1, s_3}{S; O_1 * O_2 \Vdash s_1, c_1 \Uparrow e_1, s_3} \text{(OS2)}$$

$$\frac{O(z) = c_h \quad S; O - z \Vdash s_1, c_h \Uparrow e_1, s_2}{S; O \Vdash s_1, c_1 \Uparrow e_1, s_2} \text{(OSH3)}$$

$$\frac{O_1 * O_2(z) = c_h \quad S; O_1 - z \Vdash s_1, c_h \Downarrow s_2 \quad S; O_2 - z \Vdash s_2, c_1 \Uparrow e_1, s_3}{S; O_1 * O_2 \Vdash s_1, c_1 \Uparrow e_1, s_3} \text{(OS4)}$$

Figure 3.4: Big-step semantic rules for exceptions and signal handling Part3

Addition or any other arithmetic operation of two expressions is straightforward. Both expressions (E_I and E'_I) are evaluated to the values (v_1 and v_2) one by one in a state s_1. Then, the arithmetic operation is performed on v_1 and v_2. The result value v is a return value of the arithmetic operation of two expressions E_I and E'_I. Please note that evaluation of expressions does not change the state.

3.3 Big-step Rules in Detail

There are few semantic rules (`WhileTrue`,`SeqComp`, `Exception handling`, and `Signal handling`) that require a one-shot signal binding splitting, which is described in the Definition 3.1.2.

3.3.1 Assignment

$$\frac{s_1 \Vdash E \Downarrow v}{S; O \Vdash s_1, x := E \Downarrow s_1 [x \mapsto v]}$$

An expression E evaluates to a value v in a state s_1. The result of an assignment is an update of the state s_1, such that a variable x points to the value v.

3.3.2 Sequential Composition

$$\frac{S; O_1 \Vdash s_1, c_1 \Downarrow s_2 \quad S; O_2 \Vdash s_2, c_2 \Downarrow s_3}{S; O_1 * O_2 \Vdash s_1, (c_1 \,;\, c_2) \Downarrow s_3}$$

The one-shot signal binding is split (Definition 3.1.2) non-deterministically between commands c_1 and c_2. It means that if the one-shot signal arrives during execution of c_1, it can't be handled during c_2, and vice versa. There are no such limitations for the persistent signals. Therefore, a persistent signal binding S is copied for every branch.

3.3.3 Repetitive Construct `while`

$$\frac{s_1 \models E_B \qquad S; O_1 \Vdash s_1, c \Downarrow s_2 \qquad S; O_2 \Vdash s_2, \texttt{while}\, E_B\, \texttt{do}\, c \Downarrow s_3}{S; O_1 * O_2 \Vdash s_1, \texttt{while}\, E_B\, \texttt{do}\, c \Downarrow s_3}$$

$$\frac{s_1 \models \neg E_B}{S; O \Vdash s_1, \texttt{while}\, E_B\, \texttt{do}\, c \Downarrow s_1}$$

If E_B evaluates to the value `true` in a state s_1, then a body of the `while` c is executed. Then, the same rule is applied recursively to the right branch. If E_B evaluates to the value `false` in a state s_1, then a body of the `while` c never runs; thus, the sate s_1 remains unchanged. Splitting of the signal binding is analogous to the one explained in subsection 3.3.2.

3.3.4 Exception Handling

$$\frac{}{S; O \Vdash s, \texttt{throw}\, e \Uparrow e, s}$$

$$\frac{S; O_1 \Vdash s_1, c_B \Uparrow e_k, s_2 \quad S; O_2 \Vdash s_2, c_h \Downarrow s_3}{S; O_1 * O_2 \Vdash s_1, \texttt{try}\ c_B\ \texttt{handle}\ e_k\ \texttt{by}\ c_h \Downarrow s_3}$$

In a first rule, a command `throw` e doesn't change a state itself. It raises an exception, so the exception propagation takes place. A signal could be handled before the command `throw` e, or before the exception handler (just after exception propagation terminates).

Our language is capable of supporting signal handling during the exception propagation, but our design choice is to exclude this possibility. In the real-life implementations, handling a signal after the raised exception leads to the unpredictable outcomes, because a raised exception may indicate a memory corruption. In our language, programmer has a control over when and what kind of exception to throw; however, we stick to the real-life convention.

The second rule defines a block for an exception, where it could be handled. Any e_k thrown inside c_B will be handled by c_h. This rule requires a signal splitting (Definition 3.1.2).

3.3.5 Conditional Construct `if`

$$\frac{s_1 \models E_B \qquad S; O \Vdash s_1, c_1 \Downarrow s_2}{S; O \Vdash s_1, \texttt{if}\ E_B\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \Downarrow s_2}$$

$$\frac{s_1 \models \neg E_B \qquad S; O \Vdash s_1, c_2 \Downarrow s_3}{S; O \Vdash s_1, \texttt{if}\ E_B\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \Downarrow s_3}$$

There are two rules for two cases. If E_B evaluates to the value `true` in a state s_1, then the first branch of the `if-else` structure is executed. If E_B evaluates to the value `false`, the second branch is executed. There is no splitting of the signal binding as only one branch is executed.

3.3.6 Signal Binding

$$\frac{S\,[\,z \mapsto c_h\,]; O \Vdash s_1, c_B \Downarrow s_2}{S; O \Vdash s_1, \texttt{bind}\ z\ \texttt{to}\ c_h\ \texttt{in}\ c_B \Downarrow s_2}$$

$$\frac{S; O\,[\,z \mapsto c_h\,] \Vdash s_1, c_B \Downarrow s_2}{S; O \Vdash s_1, \texttt{bind/1}\ z\ \texttt{to}\ c_h\ \texttt{in}\ c_B \Downarrow s_2}$$

We have two rules for the signal binding: one for the persistent signals and one for the one-shot signals. The rules are straightforward, binding commands add a new signal z to the corresponding signal binding context (S for persistent and O for one-shot) with a link to a signal handler c_h. Then a command c_B runs in a scope with an extended signal binding $S\,[\,z \mapsto c_h\,]; O$ or $S; O\,[\,z \mapsto c_h\,]$.

3.3.7 Signal Handling

$$\frac{S; O_1 \Vdash s_1, c_1 \Downarrow s_2 \quad S(z) = c_h \quad S - z; O_2 \Vdash s_2, c_h \Downarrow s_3}{S; O_1 * O_2 \Vdash s_1, c_1 \Downarrow s_3}$$

$$\frac{S(z) = c_h \quad S - z; O_1 \Vdash s_1, c_h \Downarrow s_2 \quad S; O_2 \Vdash s_2, c_1 \Downarrow s_3}{S; O_1 * O_2 \Vdash s_1, c_1 \Downarrow s_3}$$

$$\frac{S; O_1 - z \Vdash s_1, c_1 \Downarrow s_2 \quad O_1 * O_2(z) = c_h \quad S; O_2 - z \Vdash s_2, c_h \Downarrow s_3}{S; O_1 * O_2 \Vdash s_1, c_1 \Downarrow s_3}$$

$$\frac{O_1 * O_2(z) = c_h \quad S; O_1 - z \Vdash s_1, c_h \Downarrow s_2 \quad S; O_2 - z \Vdash s_2, c_1 \Downarrow s_3}{S; O_1 * O_2 \Vdash s_1, c_1 \Downarrow s_3}$$

During a signal execution, only the signal of a corresponding handler is blocked inside the handler. Thus, the signal handler becomes interruptible by other signals, except by itself.

One may ask, why do we require a binding splitting even for the persistent handling? It should be noted, that whenever rule assumes a resource sharing (one-shot signals in our case), the one-shot signal binding splitting is enforced by default.

In a particular case of the persistent signal handling, the one-shot binding is split nondeterministically between the main command and the handler, as the one-shot handler may arrive during the execution of each of them.

3.3.8 Signal Blocking

$$\frac{S - z; O - z \Vdash s_1, c \Downarrow s_2}{S; O \Vdash s_1, \texttt{block}\, z\, \texttt{in}\, c \Downarrow s_2}$$

There is only one signal blocking rule for both types of signals. A signal z belongs to the only one domain, $\mathrm{dom}(S)$ or $\mathrm{dom}(O)$. Thus, if $z \in \mathrm{dom}(S)$, then $z \notin \mathrm{dom}(O)$; thus, $(\mathrm{dom}(O) \setminus \{z\}) = \mathrm{dom}(O)$. Therefore, excluding z from both domains results in a required outcome. Analogously, it works when $z \notin \mathrm{dom}(S)$ and $z \in \mathrm{dom}(O)$.

3.4 `skip` command

In many languages (e.g.: used in [77]), one may observe a `skip` command that technically does nothing. In our language, we do not have a separate command for such behaviour. For example, there is no explicit rule such as

$$\frac{}{S; O \Vdash s_1, \texttt{skip} \Downarrow s_1}$$

Instead, one may use a `while` command with a `false` boolean condition $\texttt{while}\, E_B\, \texttt{do}\, c$.

$$\frac{s_1 \models \neg E_B}{S; O \Vdash s_1, \texttt{while}\, E_B\, \texttt{do}\, c \Downarrow s_1}$$

Thus, if E_B evaluates to the value `false` in a state s_1, then a body of the `while` c never runs and the sate s_1 remains unchanged. That has an analogous effect to the command `skip` defined in other languages.

3.5 Interaction between Signal and Exception Handling

There is potentially a pitfall in combining signals and jumps (such as exceptions). Consider the following pseudo-code:

```
install(handler);
goto L;
uninstall(handler);
L:
```

Here the signal handler would not be removed correctly if control jumps past the uninstalling command. In fact the problem is quite general, and arises whenever resource management is combined with jumping.

In our language as defined in Definition 3.1.1, such a potential problem case is presented by the following code:

$$\texttt{try}\ (\texttt{bind}\, z\, \texttt{to}\, h\, \texttt{in}\, \texttt{throw}\, e)\ \texttt{handle}\ e\ \texttt{by}\ g$$

The intended meaning is that the signal z is bound locally inside the body of an exception block. The signal handler may run immediately before the $\texttt{throw}\, e$ command. However, once the exception has propagated to the exception handler, it has left the scope of the signal binding, so that the signal handler should not be able to run. To see that the big-step semantics (Figure 3.2, Figure 3.3 and Figure 3.4) correctly handles this case, consider the derivation tree in Figure 3.5.

In a big-step semantics, block structure is handled correctly "for free". The extended signal binding $S\,[\,z \mapsto h\,]$ is confined to the subtree of the body of the binding. When the body is left, the evaluation is resumed with the old S, which is what used in the evaluation of g. Even when control leaves the signal block abruptly via an exception, there is no danger that a signal handler escapes from its scope. By contrast, if we use a small-step semantics, and in particular an abstract machine, the uninstalling of signal handlers needs to be performed explicitly [93].

$$\dfrac{\dfrac{\dfrac{S\,[\,z \mapsto h\,](z) = h \quad S - z; O \Vdash s_1, h \Downarrow s \qquad \overline{S\,[\,z \mapsto h\,]; O \Vdash s, \texttt{throw}\, e \Uparrow e, s}}{S\,[\,z \mapsto h\,]; O \Vdash s_1, \texttt{throw}\, e \Uparrow e, s}}{S; O \Vdash s_1, (\texttt{bind}\, z\, \texttt{to}\, h\, \texttt{in}\, \texttt{throw}\, e) \Uparrow e, s} \qquad S; O \Vdash s, g \Downarrow s_3}{S; O \Vdash s_1, \texttt{try}\ (\texttt{bind}\, z\, \texttt{to}\, h\, \texttt{in}\, \texttt{throw}\, e)\ \texttt{handle}\ e\ \texttt{by}\ g \Downarrow s_3}$$

Figure 3.5: A signal binding inside of the exception block

3.5.1 Question of Priority - Design Choice

In any language, combination of different control structures with different features and conventions leads to subtle questions. It is crucial to have a clear understanding of how

$$\frac{S\,[\,z \mapsto h\,](z) = h \qquad S - z; O \Vdash s, h \Downarrow s_3 \qquad S\,[\,z \mapsto h\,]; O \Vdash s_3, g \Downarrow s_4}{S\,[\,z \mapsto h\,]; O \Vdash s, g \Downarrow s_4}$$

$$\frac{S\,[\,z \mapsto h\,](z) = h \quad S - z; O \Vdash s_1, h \Downarrow s \quad \overline{S\,[\,z \mapsto h\,]; O \Vdash s, \texttt{throw}\,e \Uparrow e, s}}{S\,[\,z \mapsto h\,]; O \Vdash s_1, \texttt{throw}\,e \Uparrow e, s}$$

$$\frac{\dfrac{S\,[\,z \mapsto h\,]; O \Vdash s_1, \texttt{throw}\,e \Uparrow e, s \qquad S\,[\,z \mapsto h\,]; O \Vdash s, g \Downarrow s_4}{S\,[\,z \mapsto h\,]; O \Vdash s_1, \texttt{try}\ (\texttt{throw}\,e)\ \texttt{handle}\ e\ \texttt{by}\ g \Downarrow s_4}}{S; O \Vdash s_1, \texttt{bind}\,z\,\texttt{to}\,h\,\texttt{in}\,(\texttt{try}\ (\texttt{throw}\,e)\ \texttt{handle}\ e\ \texttt{by}\ g) \Downarrow s_4}$$

Figure 3.6: Derivation tree for the combined signals and exceptions

two constructs interact or influence each other at any possible situation that is permitted by the language. Sometimes there is no right answer and language designer should decide which construct has higher priority, which constructs could be used as a part of another construct, and etc. We design logic rules and semantics models for the language based on a while-language that is extended with exception and signal handling constructs. Signals and exceptions obey different conventions, and both constructs have privileges of other code interruptions. Thus, we had to decide which construct has a right to interrupt, and which should be blocked while other is running. Exception convention was explained in Section 2.3, and signals were introduced in Section 2.4.

In our operational semantics, exception propagation has higher priority than exception handling. Thus, signal might be handled only before exception has been *thrown* and after it has been caught (Figure 3.6). The command `throw` does not change a state itself; thus, the state remains unchanged until an exception is caught, then there are few options. If no signal arrives then an exception handler runs. If any signal arrives then the corresponding signal handler runs and only then the exception handler proceeds (Figure 3.6).

However, one can design implementation where signal handling has higher priority. In this scenario, a signal handler should be processed even if exception propagation takes place (Figure 3.7). In configuration with signal priority, the state could be changed by the signal handler even during the exception propagation. Thus, while the exception

$$\dfrac{\dfrac{\dfrac{}{S\,[\,z \mapsto h\,]; O \Vdash s, \texttt{throw}\, e \Uparrow e, s} \qquad S\,[\,z \mapsto h\,](z) = h \quad S - z; O \Vdash s, h \Downarrow s_2}{S\,[\,z \mapsto h\,]; O \Vdash s, \texttt{throw}\, e \Uparrow e, s_2}}{S; O \Vdash s, (\texttt{bind}\, z\, \texttt{to}\, h\, \texttt{in}\, \texttt{throw}\, e) \Uparrow e, s_2} \qquad \dfrac{S; O \Vdash s_1, (\texttt{bind}\, z\, \texttt{to}\, h\, \texttt{in}\, \texttt{throw}\, e) \Uparrow e, s_2 \qquad S; O \Vdash s_2, g \Downarrow s_3}{S; O \Vdash s_1, \texttt{try}\, (\texttt{bind}\, z\, \texttt{to}\, h\, \texttt{in}\, \texttt{throw}\, e)\, \texttt{handle}\, e\, \texttt{by}\, g \Downarrow s_3}$$

Figure 3.7: Signal handled after the `throw`

$$\dfrac{\dfrac{[\,z \mapsto c_h\,](z) = c_h \qquad S; \emptyset \Vdash s_1, c_h \Downarrow s_2 \qquad S; \emptyset \Vdash s_2, c_1 \Downarrow s_3}{S; [\,z \mapsto c_h\,] \Vdash s_1, c_B \Downarrow s_3}}{S; \emptyset \Vdash s_1, \texttt{bind/1}\, z\, \texttt{to}\, c_h\, \texttt{in}\, c_B \Downarrow s_3}$$

Figure 3.8: One-shot signal handling

propagates down in the tree, leaving block scopes one after another, different signal handlers could be registered in every block. Therefore, the choice of handler will be rather unpredictable and implementation dependant. This approach might be further investigated from a security point of view.

3.6 Examples for Operational Semantics

To show how the big-step rules could be applied we present a set of examples.

3.6.1 Basic Examples

In Figure 3.8, the one-shot signal z arrives before the main command c_1 even ran, thus registered handler c_h runs and only then command c_1 proceeds. Please note that the one-shot signal context no longer contains binding for the signal z while c_B runs. Therefore, if signal z arrives again, it will be ignored.

In Figure 3.9, the persistent signal z arrives after the main command c_B ran, and then registered handler c_h runs. In contrast with one-shot signal handlers, persistent handler bindings are not removed from the signal context for c_B. However, it is excluded from the signal context for the handler run.

$$\frac{S\,[\,z \mapsto c_h\,];O \Vdash s_1, c_B \Downarrow s_2 \qquad S\,[\,z \mapsto c_h\,](z) = c_h \qquad S;O \Vdash s_2, c_h \Downarrow s_3}{\dfrac{S\,[\,z \mapsto c_h\,];O \Vdash s_1, c_B \Downarrow s_3}{S;O \Vdash s_1, \texttt{bind}\, z\, \texttt{to}\, c_h\, \texttt{in}\, c_B \Downarrow s_3}}$$

Figure 3.9: Persistant signal handling

$$\frac{S\,[\,z \mapsto c_h\,](z) = c_h \qquad S;O \Vdash s_2, c_h \Downarrow s_3 \qquad S\,[\,z \mapsto c_h\,];O \Vdash s_3, c_B \Downarrow s_4}{\dfrac{S\,[\,z \mapsto c_h\,];O \Vdash s_2, c_B \Downarrow s_4}{\dfrac{S\,[\,z \mapsto c_h\,](z) = c_h \qquad S;O \Vdash s_1, c_h \Downarrow s_2 \qquad S\,[\,z \mapsto c_h\,];O \Vdash s_2, c_B \Downarrow s_4}{\dfrac{S\,[\,z \mapsto c_h\,];O \Vdash s_1, c_B \Downarrow s_4}{S;O \Vdash s_1, \texttt{bind}\, z\, \texttt{to}\, c_h\, \texttt{in}\, c_B \Downarrow s_4}}}$$

Figure 3.10: Multiple persistant signal handling

Figure 3.10 extends the example given in Figure 3.9. As it was mentioned before, calling the persistent handler does not remove a corresponding signal binding from the context of c_B. Therefore, the persistent handler will run again and again if corresponding signal arrives. In Figure 3.10, two signals z arrive one after another before the main command c_B ran. In the current version of operational semantics, all signals are unblocked while handler runs, except the signal that calls that handler. Example in Figure 3.10, shows a case when the next signal triggers a handler after the previous handler finishes. Please note that the signals may arrive after the command c_B and multiple handlers will be triggered then. This will end, when control flow leaves the scope of the signal binding.

3.6.2 Interruptible Signal Handlers

In Figure 3.11, a subtree for the signal handlers may grow up until all signals arrive; thus, at the top of the tree a signal binding will be empty. If we compare this example to the example in Figure 3.12 or Figure 3.8, we may observe that according to the nature of the one-shot signals, a signal handler could be used only once. In Figure 3.12, when a signal handler is called, the one-shot signal is excluded from the binding of a signal handler and the binding of a body.

$$\frac{S - z(z') = c'_h \qquad (S - z) - z'; O \Vdash s_1, c'_h \Downarrow s_2 \qquad S - z; O \Vdash s_2, c_h \Downarrow s_3}{S - z; O \Vdash s_1, c_h \Downarrow s_3}$$

$$\frac{S(z) = c_h \qquad S - z; O \Vdash s_1, c_h \Downarrow s_3 \qquad S; O \Vdash s_3, c_B \Downarrow s_4}{S; O \Vdash s_1, c_B \Downarrow s_4}$$

Figure 3.11: Interruptible signal handlers

$$\frac{O - z(z') = c'_h \qquad S; O' \Vdash s_1, c'_h \Downarrow s_2 \qquad S; O' \Vdash s_2, c_h \Downarrow s_3}{S; O - z \Vdash s_1, c_h \Downarrow s_3}$$

$$\frac{O(z) = c_h \qquad S; O - z \Vdash s_1, c_h \Downarrow s_3 \qquad S; O' \Vdash s_3, c_B \Downarrow s_4}{S; O \Vdash s_1, c_B \Downarrow s_4}$$

where $O' = (O - z) - z'$

Figure 3.12: Interruptible signal handlers

3.6.3 One-Shot & Persistent Signals Bindings

Having $O_1\,[\,z \mapsto c_h\,]$, is not enough to decide whether O_1 and $O_1 - z$ are equivalent or not. If $z \in \mathrm{dom}(O_1)$, then O_1 and $O_1 - z$ are obviously not equivalent. Also, $O_1\,[\,z \mapsto c_h\,] - z$ and O_1 are not equivalent if $z \in \mathrm{dom}(O_1)$, as $-z$ in first place excludes z completely. Its value (original or updated) becomes unimportant. However, $O_1\,[\,z \mapsto c_h\,] - z$ and $O_1 - z$ are equivalent, no matter if $z \in \mathrm{dom}(O_1)$ or not, as $-z$ was applied on both sides.

The aim of the Figure 3.13 and Figure 3.14 is to show how one-shot and persistent signal bindings are "shared" between sequentially composed commands, and highlight the core difference between them (splitting & copying). According to the operational semantics (rules are given in Figure 3.2, Figure 3.3 and Figure 3.4), the signal handlers run uninterruptedly as signal bindings are kept empty ($\emptyset; \emptyset \vdash \ldots$) during the execution.

In Figure 3.13, the one-shot signal binding O is split non-deterministically between commands c_1 and c_2. Thus, we write $O = O_1 * O_2$ (Definition 3.1.2). When the new signal z is registered it becomes an element of the domain O. However, $z \in \mathrm{dom}(O_1)$ or $z \in \mathrm{dom}(O_2)$ will be determined during the run time only. In this particular example, the signal z arrives in "scope" of the command c_1 ($z \in \mathrm{dom}(O'_1)$) and bound handler runs.

$$\frac{O_1\,[\,z \mapsto c_h\,](z) = c_h \qquad S; O_1 - z \Vdash s_1, c_h \Downarrow s_2 \qquad S; O_1 - z \Vdash s_2, c_1 \Downarrow s_3}{S; O_1\,[\,z \mapsto c_h\,] \Vdash s_1, c_1 \Downarrow s_3}$$

$$\frac{\dfrac{S; O_1\,[\,z \mapsto c_h\,] \Vdash s_1, c_1 \Downarrow s_3 \qquad S; O_2 \Vdash s_3, c_2 \Downarrow s_4}{S; (O_1 * O_2)\,[\,z \mapsto c_h\,] \Vdash s_1, (c_1\,;c_2) \Downarrow s_4}}{S; O_1 * O_2 \Vdash s_1, \texttt{bind/1}\; z\; \texttt{to}\; c_h\; \texttt{in}\; (c_1\,;c_2) \Downarrow s_4}$$

Figure 3.13: Splitting of the O binding in seq. composed commands

$$\frac{S\,[\,z \mapsto c_h\,](z) = c_h \qquad S; O \Vdash s_4, c_h \Downarrow s_5 \qquad S\,[\,z \mapsto c_h\,]; O \Vdash s_5, c_2 \Downarrow s_6}{S\,[\,z \mapsto c_h\,]; O \Vdash s_4, c_2 \Downarrow s_6}$$

$$\frac{S\,[\,z \mapsto c_h\,](z) = c_h \qquad S; O \Vdash s_3, c_h \Downarrow s_4 \qquad S\,[\,z \mapsto c_h\,]; O \Vdash s_4, c_2 \Downarrow s_6}{S\,[\,z \mapsto c_h\,]; O \Vdash s_3, c_2 \Downarrow s_6}$$

$$\frac{S\,[\,z \mapsto c_h\,](z) = c_h \qquad S; O \Vdash s_1, c_h \Downarrow s_2 \qquad S\,[\,z \mapsto c_h\,]; O \Vdash s_2, c_1 \Downarrow s_3}{S\,[\,z \mapsto c_h\,]; O \Vdash s_1, c_1 \Downarrow s_3}$$

$$\frac{\dfrac{S\,[\,z \mapsto c_h\,]; O \Vdash s_1, c_1 \Downarrow s_3 \qquad S\,[\,z \mapsto c_h\,]; O \Vdash s_3, c_2 \Downarrow s_6}{S\,[\,z \mapsto c_h\,]; O \Vdash s_1, (c_1\,;c_2) \Downarrow s_6}}{S; O \Vdash s_1, \texttt{bind}\; z\; \texttt{to}\; c_h\; \texttt{in}\; (c_1\,;c_2) \Downarrow s_6}$$

Figure 3.14: Multiple persistant signal handling in seq. composed commands

According to the one-shot signal binding nature, the binding for z is removed from the O_1' and consequently from the $O\,[\,z \mapsto c_h\,]$ as $O_1' \subseteq O\,[\,z \mapsto c_h\,]$. Therefore, $z \notin \mathrm{dom}(O_2)$ and if signal z arrives during the execution of the command c_2, it will be ignored.

In Figure 3.14, we focus on the persistent signal binding. The key difference with the one-shot binding is that binding just copied to the every command without splitting or modification. Thus, the same signal handler may run any number of times during the execution of the commands c_1 and c_2. This behaviour is possible because triggering persistent signal handler does not invalidate the corresponding binding.

3.6.4 Signals & Exceptions

Suppose that the signal handler relies on some resource (valid pointer, open socket, active connection, and etc.) available in a particular scope. However, as a side effect of the

$$\frac{O\,[\,z \mapsto c_h\,](z) = c_h \qquad S; O - z \Vdash s_1, c_h \Downarrow s_2 \qquad S; O - z \Vdash s_2, c_B \Downarrow s_3}{\dfrac{S; O\,[\,z \mapsto c_h\,] \Vdash s_1, c_B \Downarrow s_3}{S; O \Vdash s_1, \mathtt{bind/1}\; z\; \mathtt{to}\; c_h\; \mathtt{in}\; c_B \Downarrow s_3}}$$

Figure 3.15: One-shot signal handling before the command

$$\frac{S; O - z \Vdash s_1, c_B \Downarrow s_2 \qquad O\,[\,z \mapsto c_h\,](z) = c_h \qquad S; O - z \Vdash s_2, c_h \Downarrow s_3}{\dfrac{S; O\,[\,z \mapsto c_h\,] \Vdash s_1, c_B \Downarrow s_3}{S; O \Vdash s_1, \mathtt{bind/1}\; z\; \mathtt{to}\; c_h\; \mathtt{in}\; c_B \Downarrow s_3}}$$

Figure 3.16: One-shot signal handling after the command

$$\frac{S_1(z) = (h\,;\mathtt{throw}\,e) \qquad \dfrac{S{-}z; O \Vdash s_1, h \Downarrow s_2 \qquad \dfrac{}{S{-}z; O \Vdash s_2, \mathtt{throw}\,e \Uparrow e, s_2}}{S{-}z; O \Vdash s_1, (h\,;\mathtt{throw}\,e) \Uparrow e, s_2}}{\dfrac{S_1; O \Vdash s_1, c_B \Uparrow e, s_2}{S; O \Vdash s_1, (\mathtt{bind}\; z\; \mathtt{to}\; (h\,;\mathtt{throw}\,e)\; \mathtt{in}\; c_B) \Uparrow e, s_2}}$$

$$\frac{S; O \Vdash s_1, (\mathtt{bind}\; z\; \mathtt{to}\; (h\,;\mathtt{throw}\,e)\; \mathtt{in}\; c_B) \Uparrow e, s_2 \qquad S; O \Vdash s_2, g \Downarrow s_3}{S; O \Vdash s_1, \mathtt{try}\; (\mathtt{bind}\; z\; \mathtt{to}\; (h\,;\mathtt{throw}\,e)\; \mathtt{in}\; c_B)\; \mathtt{handle}\; e\; \mathtt{by}\; g \Downarrow s_3}$$

where $S_1 = S\,[\,z \mapsto (h\,;\mathtt{throw}\,e)\,]$

Figure 3.17: Persistent handler with an exception triggered before the command

$$\frac{\dfrac{S - z; O \Vdash s_2, h \Downarrow s_3 \qquad \dfrac{}{S - z; O \Vdash s_3, \mathtt{throw}\,e \Uparrow e, s_3}}{S - z; O \Vdash s_2, (h\,;\mathtt{throw}\,e) \Uparrow e, s_3} \qquad = \mathcal{F}}{\dfrac{\dfrac{S_1; O \Vdash s_1, c_B \Downarrow s_2 \qquad S_1(z) = (h\,;\mathtt{throw}\,e) \qquad \mathcal{F}}{S_1; O \Vdash s_1, c_B \Uparrow e, s_3}}{S; O \Vdash s_1, (\mathtt{bind}\; z\; \mathtt{to}\; (h\,;\mathtt{throw}\,e)\; \mathtt{in}\; c_B) \Uparrow e, s_3} \qquad S; O \Vdash s_3, g \Downarrow s_4}{S; O \Vdash s_1, \mathtt{try}\; (\mathtt{bind}\; z\; \mathtt{to}\; (h\,;\mathtt{throw}\,e)\; \mathtt{in}\; c_B)\; \mathtt{handle}\; e\; \mathtt{by}\; g \Downarrow s_4}$$

where $S_1 = S\,[\,z \mapsto (h\,;\mathtt{throw}\,e)\,]$

Figure 3.18: Persistent handler with an exception triggered after the command

handler execution, resource becomes unavailable (freed pointer, closed socket, inactive connection). In this situation, multiple handler execution will lead to the program failure. Obviously, one-shot signal handlers are perfectly fit for purpose. In Figure 3.15, the one-shot signal handler c_h runs before the command c_B. Thus, when control flow returns to c_B, the signal context no longer contains a binding for the handler c_h. In Figure 3.16, the one-shot signal handler c_h runs after the command c_B, so (as handler definitely runs) signal binding does not contain a binding for c_h.

One the other hand, a persistent handler combined with an exception imitates one-shot signal handler to some extent. The key trick is in adding of a "`throw`" command to the end of the persistent handler. As a result of a thrown exception, control leaves the signal block, so the persistent signal handler will not run again.

In Figure 3.17, the persistent signal handler runs and throws an exception. As exception propagation takes place, the command c_B does not run. In Figure 3.18, the command c_B runs before the persistent signal handler has been triggered. Thus, the rise of an exception does not influence the command c_B at that point.

Comparing derivation trees from Figure 3.16 and Figure 3.18, we may observe some similarities. In both cases, the main command runs first and then signal handler runs only once. The only difference is that handler's singular run is achieved by two different approaches.

Comparing derivation trees from Figure 3.15 and Figure 3.17, we observe the next situation: in both cases the strict conditions for the signal handlers are satisfied, but as a "side effect" of an exception propagation, the command c_B will be skipped.

More complex example (where signal handled before and after the `throw`) could be found in Figure 3.19.

$$\frac{\overline{S';O \Vdash s_2, \mathtt{throw}\, e \Uparrow e, s_2} \qquad S'(z) = h \qquad S - z; O \Vdash s_2, h \Downarrow s_3}{S';O \Vdash s_2, \mathtt{throw}\, e \Uparrow e, s_3}$$

$$\frac{\dfrac{S'(z) = h \qquad S - z; O \Vdash s_1, h \Downarrow s_2 \qquad S';O \Vdash s_2, \mathtt{throw}\, e \Uparrow e, s_3}{S';O \Vdash s_1, \mathtt{throw}\, e \Uparrow e, s_3}}{S;O \Vdash s_1, (\mathtt{bind}\, z\, \mathtt{to}\, h\, \mathtt{in}\, \mathtt{throw}\, e) \Uparrow e, s_3}$$

$$\frac{S;O \Vdash s_1, (\mathtt{bind}\, z\, \mathtt{to}\, h\, \mathtt{in}\, \mathtt{throw}\, e) \Uparrow e, s_3 \qquad S;O \Vdash s_3, g \Downarrow s_4}{S;O \Vdash s_1, \mathtt{try}\ (\mathtt{bind}\, z\, \mathtt{to}\, h\, \mathtt{in}\, \mathtt{throw}\, e)\ \mathtt{handle}\ e\ \mathtt{by}\ g \Downarrow s_4}$$

where $S' = S\,[\,z \mapsto h\,]$

Figure 3.19: Signal handled before and after the `throw`

CHAPTER 4

ABSTRACT MACHINE

In this chapter, we review some basic design decisions for abstract machines and define a stack machine for signal handlers. The form of a machine configuration includes two stacks: one for the exception and signal bindings, and another for continuations. We explain in detail why it is important to keep track of signal and exception handlers on the same stack to achieve required interaction between them. Application of the transition steps is shown in a set of examples. Finally, we discuss some issues that need to be solved as part of the correctness proof of the signal machine with respect to the big-step operational semantics.

4.1 From Big-step to Abstract Machine

Suppose we have a language with only atomic commands a and sequential composition $c_1; c_2$. We define a machine with a command, a current state and a continuation:

$$\begin{aligned} \langle c_1; c_2,\ s,\ k\rangle &\rightsquigarrow \langle c_1,\ s,\ c_2; k\rangle \\ \langle a,\ s_1,\ c; k\rangle &\rightsquigarrow \langle c,\ s_2,\ k\rangle \end{aligned}$$

For a command c, the initial state of the machine has some initial state s_0. The initial continuation is a special instruction `return`. When an atomic command runs, it may modify the state s_1 to a new state s_2. At the same time, the next command c is popped from the continuation. This last feature is similar to the way a real CPU increments the

instruction pointer to the next instruction after it executes an instruction.

The relation to big-step semantics is as follows:

$$s_1, c \Downarrow s_2$$

if and only if for all c' and k:

$$\langle c,\, s_1,\, c'; k\rangle \overset{*}{\rightsquigarrow} \langle c',\, s_2,\, k\rangle$$

Proof By induction over the length of the run. We make a case analysis of the first step. Then we apply the induction hypothesis to the middle of the run. Then we find a step matching the first one that will get us back to the same k.

For the atomic command, assume the initial state s_1, continuation $c'; k$, and a command a at the evaluation position

$$\langle a,\, s_1,\, c'; k\rangle$$

According to the transition rules, we proceed to another configuration

$$\langle a,\, s_1,\, c'; k\rangle \quad \rightsquigarrow \quad \langle c',\, s_2,\, k\rangle$$

In the big-step, we have

$$s_1, a \Downarrow s_2$$

as required.

For the sequential composition, assume sequentially composed commands $(c_1; c_2)$, initial state s_1 and continuation $c'; k$

$$\langle (c_1; c_2),\, s_1,\, c'; k\rangle$$

According to the transition rule we proceed as follows

$$\begin{aligned}\langle (c_1; c_2),\ s_1,\ c'; k\rangle &\rightsquigarrow \langle c_1,\ s_1,\ c_2; c'; k\rangle \\ &\overset{*}{\rightsquigarrow} \langle c_2,\ s_2,\ c'; k\rangle \\ &\overset{*}{\rightsquigarrow} \langle c',\ s_3,\ k\rangle\end{aligned}$$

We apply the induction hypothesis to the second transition sequence, which implies

$$s_1, c_1 \Downarrow s_2$$

Then we apply induction hypothesis again for the third transition sequence, which implies

$$s_2, c_2 \Downarrow s_3$$

Finally, the last machine configuration matches with the first one in terms of continuation component k. Thus,

$$s_1, (c_1; c_2) \Downarrow s_3$$

□

One of the advantages of big-step semantics is that is has built-in support for block-structure. Suppose we have some construct `block` c `end`. In a big-step semantics, we can define rules that use the semantics of c. With a machine, it is more complex. The machine needs to enter the block, run the command c, which may involve pushing and popping the continuation, and then leave the block. We may have to define explicit instructions for entering and leaving the block-structured contruct. Moreover, if we also have exceptions, the possibility of remove part of the continuation further complicates the machine, as we need to make sure that the block structure is handled correctly.

Definition 4.1.1 (Stuck machine configuration) For a machine configuration m, we write

$$m \not\rightsquigarrow$$

when there is no configuration m' such that $m \rightsquigarrow m'$.

4.2 Stack Machine for Signal Handlers

We define an abstract machine in order to highlight some of the issues that may arise in possible implementations of block-structured signals, such as managing the stack. The implementation of signal handlers in our abstract machine was inspired by the real implementations of exceptions in contrast to the unstructured `longjmp` that exceptions were designed to replace.

The defined block-structured form of signal handling requires a signal handler to be installed at the beginning of the block and uninstalled at the end. Therefore, to keep track of signal handlers in a particular scope, we use a signal stack. However, the addition of exceptions complicates the scoping of signal handlers. When control leaves a signal scope via a raised exception, the handler should be uninstalled. Thus, to implement the desired interaction between signal and exception scope, we keep track of signal handlers and exception handlers on the same stack. When an exception is raised, the stack is popped until the nearest enclosing handler for the exception name is found. The same popping of the common handler stack also removes any intervening signal handlers.

A machine configuration is of the form $\langle c \mid s \mid \beta \mid J \mid K \rangle$, where c is the expression that the machine is currently trying to evaluate, s is a state. The bit vector component β is used for keeping track of installed (not blocked) signals. J is a stack, which holds the signal and exception bindings. K is a continuation, which tells the machine what to do when it is finished with the current command c. The initial continuation is a special instruction `return`. The special symbol $\blacksquare$ is used to represent an empty stack in the components J and K. When we get $\langle \texttt{return} \mid s \mid \beta \mid \blacksquare \mid \blacksquare \rangle$, program execution is

finished. The full list of transition steps is given in Figure 4.1 and Figure 4.2. To evaluate expression E in a state s, we apply the function *eval* (Defintion 4.2.1), which returns a value v.

β^0 stands for a null bit vector (which means blocking or ignoring of all signals). The system instruction `pop-upd(`β'`)` removes the top element from a stack J and updates β to β'. The system instruction `update(`β'`)` updates β to β'. We define J as a data structure that follows stack discipline except in the case of one-shot signal handling. The J stack is manipulated by the system instructions that are pushed in and popped out from the continuation stack K.

β is a function from signal names z to `Booleans`. For each signal name z, $\beta(z)$ tells us whether the signal is currently enabled. Then $\beta{+}z$ is a shorthand for $\beta[z \mapsto true]$ and $\beta{-}z$ stands for $\beta[z \mapsto false]$.

For a `throw` e_1 command, where $e_1 \in \mathrm{dom}(J)$, we apply the `unwind` function (Definition 4.2.2), which returns a quadruple that is used to construct the next machine configuration. If $e_1 \notin \mathrm{dom}(J)$, then the machine gets stuck with an unhandled exception, in the sense that there is no transition for this configuration, so that

$$\langle\, \texttt{throw}\, e_1 \;\mid s \mid \beta \mid J \mid K \,\rangle \not\mapsto$$

An exception binding tag has the form of (e, h), where e is an exception identifier, and h is a handler. A persistent signal binding tag has the form of (z, h), where z is a signal name, and h is a handler. A one-shot signal binding tag has the form of (z, h, u), where z is a signal name, h is a handler, and u is a bit indicating that the handler has been used once (u=1) or not (u=0). Handling of the one-shot signals requires update of the J stack; to be more precise, the bit u in (z, h, u) is updated. Please note, the `pop-upd(`β^0`)`^{2} stands for `pop-upd(`β^0`);pop-upd(`β^0`)`.

$$\langle\, c_1; c_2 \;\mid s_1 \mid \beta_1 \mid J_1 \mid K_1 \,\rangle$$
$$\quad \rightsquigarrow \langle\, c_1 \;\mid s_1 \mid \beta_1 \mid J_1 \mid c_2; K_1 \,\rangle$$

$$\langle\, x := E \;\mid s_1 \mid \beta_1 \mid J_1 \mid c'; K_1 \,\rangle$$
$$\quad \rightsquigarrow \langle\, c' \;\mid s_1\,[\,x \mapsto v\,] \mid \beta_1 \mid J_1 \mid K_1 \,\rangle \qquad \text{(where } \texttt{eval(}E,\ s_1\texttt{)} = v\text{)}$$

$$\langle\, \texttt{bind}\, z\, \texttt{to}\, h\, \texttt{in}\, c \;\mid s \mid \beta \mid J \mid K \,\rangle$$
$$\quad \rightsquigarrow \langle\, c \;\mid s \mid \beta{+}z \mid (z, h), J \mid \texttt{pop-upd(}\beta\texttt{)}; K \,\rangle$$

$$\langle\, \texttt{bind/1}\, z\, \texttt{to}\, h\, \texttt{in}\, c \;\mid s \mid \beta \mid J \mid K \,\rangle$$
$$\quad \rightsquigarrow \langle\, c \;\mid s \mid \beta{+}z \mid (z, h, 0), J \mid \texttt{pop-upd(}\beta\texttt{)}; K \,\rangle$$

$$\langle\, \texttt{pop-upd(}\beta_1\texttt{)} \;\mid s \mid \beta_2 \mid (z, h), J \mid c; K \,\rangle$$
$$\quad \rightsquigarrow \langle\, c \;\mid s \mid \beta_1 \mid J \mid K \,\rangle$$

$$\langle\, c \;\mid s \mid \beta \mid J_1, (z, h), J_2 \mid K \,\rangle$$
$$\quad \rightsquigarrow \langle\, h \;\mid s \mid \beta^0 \mid J_1, (z, h), J_2 \mid \texttt{update(}\beta\texttt{)}; c; K \,\rangle$$
$$\quad \text{(handling of the persistent signal)}$$

$$\langle\, c \;\mid s \mid \beta \mid J_1, (z, h, 0), J_2 \mid K \,\rangle$$
$$\quad \rightsquigarrow \langle\, h \;\mid s \mid \beta^0 \mid J_1, (z, h, 1), J_2 \mid \texttt{update(}\beta{-}z\texttt{)}; c; K \,\rangle$$
$$\quad \text{(handling of the one-shot signal)}$$

$$\langle\, \texttt{update(}\beta_1\texttt{)} \;\mid s \mid \beta_2 \mid J \mid c; K \,\rangle$$
$$\quad \rightsquigarrow \langle\, c \;\mid s \mid \beta_1 \mid J \mid K \,\rangle$$

Figure 4.1: Transition steps - Part 1

$$\langle \texttt{block}\, z\, \texttt{in}\, c \;\mid s \mid \beta_1 \mid J \mid K \rangle$$
$$\rightsquigarrow \langle c \;\mid s \mid \beta_1{-}z \mid J \mid \texttt{update}(\beta_1); K \rangle$$

$$\langle \texttt{block/1}\, z\, \texttt{in}\, c \;\mid s \mid \beta_1 \mid J \mid K \rangle$$
$$\rightsquigarrow \langle c \;\mid s \mid \beta_1{-}z \mid J \mid \texttt{update}(\beta_1); K \rangle$$

$$\langle \texttt{try}\; c_b\; \texttt{handle}\; e\; \texttt{by}\; h \;\mid s \mid \beta \mid J \mid K \rangle$$
$$\rightsquigarrow \langle c_b \;\mid s \mid \beta \mid (e,h), J \mid \texttt{pop-upd}(\beta); K \rangle$$

$$\langle \texttt{throw}\, e_1 \;\mid s \mid \beta \mid J_1, (e_1,h), J_2 \mid K_1 \rangle$$
$$\rightsquigarrow \langle h \;\mid s \mid \beta' \mid J_2 \mid K_2 \rangle$$
$$(\text{where } \texttt{unwind}(e_1,\, (J_1, (e_1,h), J_2),\, K_1) = (h, \beta', J_2, K_2))$$

Figure 4.2: Transition steps - Part 2

Definition 4.2.1 (eval function)

$$\begin{aligned} \texttt{eval}(x,\, s) &= s(x) \\ \texttt{eval}(E_1 + E_2,\, s) &= \texttt{eval}(E_1,\, s) + \texttt{eval}(E_2,\, s) \end{aligned}$$

Definition 4.2.2 (unwind function)

$$\begin{aligned} \texttt{unwind}(e_1,\, J,\, c; K) &= \texttt{unwind}(e_1,\, J,\, K) \\ \texttt{unwind}(e_1,\, J,\, \texttt{update}(\beta); K) &= \texttt{unwind}(e_1,\, J,\, K) \\ \texttt{unwind}(e_1,\, ((z,h), J),\, \texttt{pop-upd}(\beta); K) &= \texttt{unwind}(e_1,\, J,\, K) \\ \texttt{unwind}(e_1,\, ((e_1,h), J),\, \texttt{pop-upd}(\beta); K) &= (h, \beta, J, K) \end{aligned}$$

Function `unwind` has three input parameters: name of the exception, J and K stacks.

1. If there is a non-system instruction on top of K, it is discarded.

2. If there is a system instruction `update` on top of K, it is discarded.

3. If there is a system instruction `pop-upd` on top of K, and if there is a signal binding or an exception binding for another exception on top of J, both are discarded.

4. If there is a system instruction `pop-upd` on top of K and if there is an exception binding for the required exception name on top of J, we get a corresponding signal handler from the signal binding and β from the system instruction. Then we discard exception binding from J and `pop-upd` from K to get required J and K stacks.

4.3 Examples of the Machine Runs

We have already seen in previous examples (e.g.: Figure 3.17 and Figure 3.6) that the big-step semantics gives us block structure for free. This becomes very useful in studying block structured constructs and their interactions. On the contrary, the machine needs to manage block structure explicitly with a help of the stack. The examples of corresponding machine runs are given in Figure 4.3 and Figure 4.4. Please note, the `pop-upd(`β^0`)`^{2} stands for `pop-upd(`β^0`);pop-upd(`β^0`)`.

The example in Figure 3.13 shows how the big-step syntax makes it easy to address one-shot signals with splitting the bindings. By contrast, the machine needs to perform extra administrative work with the binding tags and the stack to implement one-shot signal handling (Figure 4.5). One may observe that the abstract machine is more complex than the big-step semantics, as machine needs to deal with many details explicitly. Overall, we see that the machine is closer to implementations, whereas the big-step semantics is more convenient for abstract reasoning.

4.4 Towards Signal Machine Correctness

Before we can move towards correctness of the machine relative to the big-step semantics, an issue with exception handling should be discussed. Exceptions give rise to non-local control that violates the simple stack discipline. In order to reason about an exception

$\langle \texttt{try}\ (\texttt{bind}\, z\, \texttt{to}\, (h\,;\texttt{throw}\, e)\, \texttt{in}\, c)\ \texttt{handle}\ e\ \texttt{by}\ g \ \mid s_1 \mid \beta^0 \mid \blacksquare \mid \texttt{return} \rangle$

$\rightsquigarrow \langle \texttt{bind}\, z\, \texttt{to}\, (h\,;\texttt{throw}\, e)\, \texttt{in}\, c \ \mid s_1 \mid \beta^0 \mid (e, g) \mid \texttt{pop-upd}(\beta^0) \rangle$

$\rightsquigarrow \langle c \ \mid s_1 \mid \beta^0{+}z \mid (z, (h\,;\texttt{throw}\, e)), (e, g) \mid \texttt{pop-upd}(\beta^0)^2 \rangle$

$\rightsquigarrow \langle c \ \mid s_1 \mid \beta^0{+}z \mid (z, (h\,;\texttt{throw}\, e)), (e, g) \mid \texttt{pop-upd}(\beta^0)^2 \rangle$

$\rightsquigarrow \langle (h\,;\texttt{throw}\, e) \ \mid s_1 \mid \beta^0 \mid (z, (h\,;\texttt{throw}\, e)), (e, g) \mid \texttt{update}(\beta^0{+}z); c; \texttt{pop-upd}(\beta^0)^2 \rangle$

$\rightsquigarrow \langle h \ \mid s_1 \mid \beta^0 \mid (z, (h\,;\texttt{throw}\, e)), (e, g) \mid \texttt{throw}\, e; \texttt{update}(\beta^0{+}z); c; \texttt{pop-upd}(\beta^0)^2 \rangle$

$\rightsquigarrow \langle \texttt{throw}\, e \ \mid s_2 \mid \beta^0 \mid (z, (h\,;\texttt{throw}\, e)), (e, g) \mid \texttt{update}(\beta^0{+}z); c; \texttt{pop-upd}(\beta^0)^2 \rangle$

$\rightsquigarrow \langle g \ \mid s_2 \mid \beta^0 \mid \blacksquare \mid \texttt{return} \rangle$

$\rightsquigarrow \langle \texttt{return} \ \mid s_3 \mid \beta^0 \mid \blacksquare \mid \blacksquare \rangle$

Figure 4.3: Binding inside of the try block

$$\langle \texttt{bind}\, z\, \texttt{to}\, h\, \texttt{in}\, (\texttt{try}\ (\texttt{throw}\, e)\ \texttt{handle}\ e\ \texttt{by}\ g) \mid s_1 \mid \beta^0 \mid \blacksquare \mid \texttt{return} \rangle$$

$$\rightsquigarrow \langle \texttt{try}\ (\texttt{throw}\, e)\ \texttt{handle}\ e\ \texttt{by}\ g \mid s_1 \mid \beta^0{+}z \mid (z,h) \mid \texttt{pop-upd}(\beta^0) \rangle$$

$$\rightsquigarrow \langle \texttt{throw}\, e \mid s_1 \mid \beta^0{+}z \mid (e,g),(z,h) \mid \texttt{pop-upd}(\beta^0{+}z); \texttt{pop-upd}(\beta^0) \rangle$$

$$\rightsquigarrow \langle h \mid s_1 \mid \beta^0 \mid (e,g),(z,h) \mid \texttt{update}(\beta^0{+}z); \texttt{throw}\, e; \texttt{pop-upd}(\beta^0{+}z); \texttt{pop-upd}(\beta^0) \rangle$$

$$\rightsquigarrow \langle \texttt{update}(\beta^0{+}z) \mid s_2 \mid \beta^0 \mid (e,g),(z,h) \mid \texttt{throw}\, e; \texttt{pop-upd}(\beta^0{+}z); \texttt{pop-upd}(\beta^0) \rangle$$

$$\rightsquigarrow \langle \texttt{throw}\, e \mid s_2 \mid \beta^0{+}z \mid (e,g),(z,h) \mid \texttt{pop-upd}(\beta^0{+}z); \texttt{pop-upd}(\beta^0) \rangle$$

$$\rightsquigarrow \langle g \mid s_2 \mid \beta^0{+}z \mid (z,h) \mid \texttt{pop-upd}(\beta^0) \rangle$$

$$\rightsquigarrow \langle h \mid s_2 \mid \beta^0 \mid (z,h) \mid \texttt{update}(\beta^0{+}z); g; \texttt{pop-upd}(\beta^0) \rangle$$

$$\rightsquigarrow \langle \texttt{update}(\beta^0{+}z) \mid s_3 \mid \beta^0 \mid (z,h) \mid g; \texttt{pop-upd}(\beta^0) \rangle$$

$$\rightsquigarrow \langle g \mid s_3 \mid \beta^0{+}z \mid (z,h) \mid \texttt{pop-upd}(\beta^0) \rangle$$

$$\rightsquigarrow \langle \texttt{pop-upd}(\beta^0) \mid s_4 \mid \beta^0{+}z \mid (z,h) \mid \texttt{return} \rangle$$

$$\rightsquigarrow \langle \texttt{return} \mid s_4 \mid \beta^0 \mid \blacksquare \mid \blacksquare \rangle$$

Figure 4.4: Exception handling inside of the binding

$$
\begin{aligned}
& \langle \texttt{bind/1}\ z\ \texttt{to}\ h_1\ \texttt{in}\ (c_1\ ;\ c_2)\ \mid s_1 \mid \beta^0 \mid \blacksquare \mid \texttt{return} \rangle \\
\rightsquigarrow\ & \langle c_1\ ;\ c_2\ \mid s_1 \mid \beta^0{+}z \mid (z, h_1, 0) \mid \texttt{pop-upd}(\beta^0); \texttt{return} \rangle \\
\rightsquigarrow\ & \langle c_1\ \mid s_1 \mid \beta^0{+}z \mid (z, h_1, 0) \mid c_2; \texttt{pop-upd}(\beta^0); \texttt{return} \rangle \\
\rightsquigarrow\ & \langle h_1\ \mid s_1 \mid \beta^0 \mid (z, h_1, 1) \mid \texttt{update}(\beta^0); c_1; c_2; \texttt{pop-upd}(\beta^0); \texttt{return} \rangle \\
\rightsquigarrow\ & \langle \texttt{update}(\beta^0)\ \mid s_2 \mid \beta^0 \mid (z, h_1, 1) \mid c_1; c_2; \texttt{pop-upd}(\beta^0); \texttt{return} \rangle \\
\rightsquigarrow\ & \langle c_1\ \mid s_2 \mid \beta^0 \mid (z, h_1, 1) \mid c_2; \texttt{pop-upd}(\beta^0); \texttt{return} \rangle \\
\rightsquigarrow\ & \langle c_2\ \mid s_3 \mid \beta^0 \mid (z, h_1, 1) \mid \texttt{pop-upd}(\beta^0); \texttt{return} \rangle \\
\rightsquigarrow\ & \langle \texttt{pop-upd}(\beta^0)\ \mid s_4 \mid \beta^0 \mid (z, h_1, 1) \mid \texttt{return} \rangle \\
\rightsquigarrow\ & \langle \texttt{return}\ \mid s_4 \mid \beta^0 \mid \blacksquare \mid \blacksquare \rangle
\end{aligned}
$$

Figure 4.5: Signal binding and seq. composed commands

raised somewhere deep inside its corresponding try block, we need to keep track of the exception handlers on the stack in terms of an invariant.

Definition 4.4.1 (Handler relation) We define the relation $\ggg$ as follows. The relation

$$\{e\}, \{J\}, \{K\} \ggg \{h\}, \{J'\}, \{K'\}$$

holds if the exception e thrown inside stacks J and K goes in one step to a configuration running the handler h, and the stacks J' and K'. More formally, the relation holds if for all s and β, there is a step

$$\langle \texttt{throw}\, e\ \mid s \mid \beta \mid J \mid K \rangle \rightsquigarrow \langle h\ \mid s \mid \beta' \mid J' \mid K' \rangle$$

where a bit vector β' is extracted from the stack component J.

All machine runs either return to the same stack or lead to a machine configuration with raised exception. All steps other than exception handling and exception raising preserve handler invariant. We need to show that for every machine run there is a

corresponding big-step derivation. A proof of the abstract machine correctness with respect to the big-step operational semantics is left for further work.

4.5 Notes about Signals Implementation

We compare how our idealized stack machine models features of real signal implementations.

4.5.1 Core Idea

Real-life implementations of Unix signals differ from our idealized block-structured construct. Our implementation of a block-structured signal handling is as follows. At the beginning of a signal block, we update a bit vector of installed signals and store corresponding handler binding into a stack structure called J. At the same time we remember the previous bit vector and add special instruction to the continuation, which will restore initial bit vector and J. At the end of a signal block, special instruction is pulled out of continuation and executed, that reinstates previous bit vector. The addition of exceptions complicates the scoping of signal handlers. When control leaves a signal scope via a raised exception, the handler should be uninstalled. To implement the desired interaction between signal and exception scope, we keep track of signal handlers and exception handlers on the same stack. When an exception is raised, the stack is popped until the nearest enclosing handler for the exception name is found. The same popping of the common handler stack also removes any intervening signal handlers.

4.5.2 Bit Vector

In our machine, β stands for the bit vector of installed not currently blocked signals; and β^0 stands for a null bit vector that may be interpreted as "all signals are blocked" or "no signals are installed". The use of this bit vector almost directly corresponds to the bit maps used in real implementations. There are bit maps of installed and blocked signals.

To define which signals should run, two bit maps are xored. In real implementations, every signal has a default pre-assigned handler. To imitate the same behaviour, in our implementation it is possible to run a command inside of nested blocks in which all signals are bound to their default handlers.

4.5.3 Exceptions and Signals

In real implementations (as explained in [22], ISO/IEC 14882 [51, 50]), exception throwing inside of signal handlers is not recommended, due to implementation restrictions. Moreover, the existing implementation of signals is not block structured. On the contrary, our abstract machine and big-step semantics deal with block structured signals and allow signal handlers to throw exceptions.

Special Case

Consider (`try` ... `handle` e `by` h) block nested into (`bind/1` z `to` h `in` ...) block. Assume that no signals arrive before exception handling block. We know that at the very beginning of exception handling block the current continuation and a bit vector will be stored, which indicates that one-shot signal is installed. The special tag will be also added to the J stack. That special tag keeps a note if one-shot signal handler, if it has run or not. Then, inside of exception handling block, signal arrives and signal handler is called. The bit vector and tag in J will be updated. Then exception inside of exception handling block is thrown. When exception is handled, the previous bit vector is updated by a stored one. Here is the problem, stored bit vector doesn't have information that signal already has run once. However, the tag in a J still keeps a note that signal has run. Thus, it adds extra search for a tag, but it doesn't result in a multiple run of one-shot signals. Extra search results in an overhead, as if signal is not in a bit vector, search over J even doesn't start. To mitigate possible overhead, an extra rule could be added that will fix bit vector if inconsistent situation is found.

4.5.4 Implementation of Exception Handling

In real implementations (e.g.: Itanium [28], and as described in [56, 22, 10]), exception handling is implemented by use of stack unwinding. The process consists of two phases. In first phase, the stack is searched for corresponding installed exception handler. If nothing is found, the running application is terminated with risen unhandled exception. If corresponding handler is found, the second phase begins. In the second phase, the stack unwinding takes place. During unwinding, special instructions (they were added at time of creation of the frame) are called from every stack frame. That ensures that register will remain in consistent state when stack unwinding finishes and handler runs.

In our current implementation, the first phase of exception handling resembles the first phase from real implementation. The stack J is searched until exception binding with exception handler is found. If search fails, machine terminates with raised exception. If search succeeds, the second phase starts from opening exception binding. The exception binding contains exception handler, bit vector and continuation. Thus machine proceeds its execution with handler at a call position, new bit vector and new continuation. This process implements exception handling using continuations, but continuation that is stored in exception binding was added at the point of entering to the exception block. Thus, such implementation could be related to stack unwinding. It is possible to adjust the current implementation of an abstract machine, to make it work more close to real implementation. The continuation and bit vector from exception binding will be removed. Thus in a search phase, after the exception binding will be found in a stack J at some particular position, the same number of system instructions will be popped from a continuation K and executed. Those instruction will ensure the consistency of bit vector and stack J. Exception handling in our implementation resembles handling in real implementations, except the fact that the abstract machine uses the extra stack J to keep track of block structures (including nesting), and the J is manipulated by special instructions in the continuation K.

CHAPTER 5

LOGIC REASONING

In this chapter, we present a program logic for our base language with both signal and exception handling. First of all, we define a binary relation $\triangleright$ that is required for the signal program logic. Then, we consequently define signal specification, signal and exception contexts. The notion of stability is crucial for our logic. We define it formally and informally, and then explain how it is used in the program logic. Supporting lemmas are presented with corresponding proofs and most of the logic rules are explained in detail in the dedicated sections. Then we introduce and discuss the notion of the ghost variables. And finally, we consider real life situations that could be addressed with our logic.

5.1 Program Logic with Specifications for Signal Handlers

The format of a program logic judgement with signal handling is as follows:

$$\Sigma \vdash \{P\}\, c\, \{Q\}$$

Here Σ contains a specification of persistent signal handlers (which may run any number of times) and the specification of one-shot signal handlers (which may run at most once) that may interfere with the command c. Thus, the specifications stored in Σ limit how the handlers can interfere with the body of c. The rest of the judgement is a standard

Hoare triple $\{P\}\, c\, \{Q\}$ with precondition P and postcondition Q for the command c.

Some auxiliary definitions will be required for the signal program logic. For two unary relations P and Q, we write $(P \triangleright Q)$ for a binary relation on a set of states relating pre-states satisfying P to post-states satisfying Q. Thus, the binary relation $\triangleright$ is a set of pairs (s_1, s_2) where $s_1 \models P$ implies $s_2 \models Q$.

Definition 5.1.1 (Binary relation $\triangleright$) The formal definition of the $P \triangleright Q$ is of the next form:

$$(s_1, s_2) \models P \triangleright Q \text{ iff } s_1 \models P \text{ implies } s_2 \models Q$$

Definition 5.1.2 (Signal specification) Signal specifications consist of a precondition and a postcondition, with a possible quantification over ghost variables α. The syntax of signal specifications R is as follows:

$$R \;::=\; (P \triangleright Q) \mid \forall \alpha.R$$

Examples of the signal specifications are as follows: $\forall \alpha.((x = \alpha) \triangleright (x = \alpha))$, $\forall \alpha.\beta.((x = \alpha \wedge y = \beta) \triangleright (x = \beta + 1 \wedge y = \alpha))$, $(z = 1 \wedge x = 2) \triangleright (z > 1 \wedge x = 0)$.

We define ghost variables as variables that do not occur in the body of commands but only occur in command's specifications and assertions. For more details on ghost variables, please see Section 5.7.

Definition 5.1.3 (Signal context) The signal context Σ has the next form:

$$z_1 : R_1, z_2 : R_2, \ldots, z_n : R_n$$

where $z_1, \ldots, z_n$ are signal names and $R_1, \ldots, R_n$ are corresponding signal specifications. Thus, assuming that R_1 and R_n employ the optional quantification then $z_1 : R_1, z_2 : R_2, \ldots, z_n : R_n$ could be rewritten as:

$$z_1 : \forall \alpha.(P_1 \triangleright Q_1), z_2 : (P_2 \triangleright Q_2), \ldots, z_n : \forall \alpha.(P_n \triangleright Q_n)$$

For persistent signals, we use a handler invariant I_z as both the precondition and the postcondition of the specification $I_z \triangleright I_z$. This use of an invariant for the handler is analogous to the invariant of a loop body in the standard Hoare logic rule for a `while` loop. For one-shot signals, the specification of the handler consists of a precondition P_z and a possibly different postcondition Q_z.

5.2 Exception Context

For a language without control constructs, program logic judgements needs only address successful termination, without any thrown exceptions. If we add exceptions, the outcome of evaluation that results in a raised exception also needs to be addressed. Such cases require us to associate a postcondition Q'_j that should hold after the corresponding exception e_j has been thrown. We extend the Hoare logic with an *exception context* η of the form $\eta = e_1 : Q'_1, e_2 : Q'_2, \ldots, e_n : Q'_n$. The form of a program logic judgement with an exception context is as follows:

$$\{P\}\, c\, \{Q\}\ \texttt{throws}\ e_1 : Q'_1, e_2 : Q'_2, \ldots, e_n : Q'_n$$

The first part of the judgement is a standard Hoare triple $\{P\}\, c\, \{Q\}$. What follows is a specification for all exceptions that may be raised by the command c. Our syntax for these exception contexts is analogous to `throws`-clauses in Java methods. In the program logic for exceptions, one needs to ensure that the precondition $\eta(e_j) = Q'_j$ for an exception e_j holds immediately before the exception is thrown. That way, it holds at the beginning of the handler.

5.3 Stability

We adopt a notion of stability to address how various handlers and commands influence each other. For example, satisfied stability assumptions for a signal handler ensure that

the signal's invariants would not be corrupted by the actions of the main command and other signals. Simply, an action is stable under possible interference with another action, if preconditions were satisfied by the initial state and possible interference took place, but the final state nevertheless satisfies postcondition.

The general definition of stability [97, 17] in rely-guarantee logic is as follows. A binary relation R_1 is stable under a binary relation R_2 if and only if $(R_2; R_1) \implies R_1$ and $(R_1; R_2) \implies R_1$. And by definition of the relational composition that could be rewritten into the next form:

$$\exists s'.(s_1, s') \models R_2 \wedge (s', s_2) \models R_1 \implies (s_1, s_2) \models R_1$$

$$\exists s'.(s_1, s') \models R_1 \wedge (s', s_2) \models R_2 \implies (s_1, s_2) \models R_1$$

The above is also written as R_1 **stable** R_2. An unary predicate P is *stable* under a binary relation R if for any program states s_1 and s_2:

$$s_1 \models P \text{ and } (s_1, s_2) \models R \text{ implies } s_2 \models P$$

The above is also written as P **stable** R.

We conjecture that the unary stability implies the binary stability, and we will need some auxiliary definitions of various forms of stability for our program logic.

Definition 5.3.1 (Stability conditions)

1. For a signal context Σ we write P **stable** Σ if for all $z_j \in \mathrm{dom}(\Sigma)$ with $\Sigma(z_j) = (P_j \triangleright Q_j)$, it is the case that P **stable** $(P_j \triangleright Q_j)$.

2. We write Σ **stable** $(P \triangleright Q)$ if for all z_j in $\mathrm{dom}(\Sigma)$ with $\Sigma(z_j) = (P_j \triangleright Q_j)$, we have P_j **stable** $(P \triangleright Q)$ and Q_j **stable** $(P \triangleright Q)$.

3. We write Σ **pairstable** if the signal specifications in Σ are pairwise stable, in the following sense: for any signals $z_1, z_2 \in \mathrm{dom}(\Sigma)$ such that $z_1 \neq z_2$, $\Sigma(z_1) = (P_1 \triangleright Q_1)$

and $\Sigma(z_2) = (P_2 \triangleright Q_2)$ it is the case that

$$P_1 \textbf{ stable } (P_2 \triangleright Q_2) \qquad \text{and} \qquad Q_1 \textbf{ stable } (P_2 \triangleright Q_2)$$

4. If P_1 **stable** $(P_2 \triangleright P_2)$, we may write P_1 **stable** P_2.

5. We write $(P \triangleright Q)$ **stable** Σ,
   if $\forall z.\Sigma(z) = z : \forall \alpha.P_z \triangleright Q_z$ we have $(P \triangleright Q)$ **stable** $(\forall \alpha.P_z \triangleright Q_z)$.

Our original definition of stability relates an unary predicate with a binary relation. Therefore, the next form P **stable** I may look incomplete, because one may read it as stability of the unary predicate P under the unary predicate I. However, this is just a short form of a stability relation where the binary relation $I \triangleright I$ takes the form of the unary predicate I, solely for the purpose of space saving. Whenever one meet a form that may look as an unary predicate is stable under another unary predicate (e.g.: P **stable** I, P **stable** Σ, or even Σ **stable** I), that form can always be expanded (Definition 5.3.1) into the standard form P_1 **stable** $P_2 \triangleright Q_2$. It should be noted, that if stability holds for a bigger Σ, then it always holds for a smaller Σ, as stability conditions remain satisfied. For example, if P_1 **stable** $(\Sigma, z : P_2 \triangleright Q_2)$ holds, then P_1 **stable** Σ trivially holds.

The stability of the pre- and postcondition under all signals is built into the meaning of judgements by way of the big-step semantics. In a big-step rule, all signal handlers could be run after the main command has terminated, or the signal handlers could be run before the main command is even begun. Hence in sequential composition $c_1; c_2$, there is no need for explicitly stating that the postcondition of the first command c_1 could be subject to interference by the signal handlers. The judgement for c_1 already takes that interference, and the need for stability, into account. The price one has to pay for the stability of the implicit pre- and postconditions is that when an atomic command is lifted into a signal context, all stability assumptions need to be established. The signals may happen before or after the atomic command, thus we need to ensure stability of P and Q under all handlers.

5.4 Program Logic for Signal and Exception Handling

Combining the above, we now define a program logic for our language with both signal and exception handling. For signal handling, the judgements contain a signal context, written on the left of the Hoare triple as

$$\Sigma \vdash \ldots$$

The exception context is written to the right of the Hoare triple as

$$\ldots \texttt{throws}\, \eta$$

It specifies the precondition that needs to hold for each exception before it can be thrown.

Definition 5.4.1 Program logic judgements are of the form

$$\Sigma \vdash \{P\}\, c\, \{Q\}\, \texttt{throws}\, \eta$$

where

- c is a command
- P and Q are unary predicates on states
- Σ is a signal context of the form $z_1 : R_1, \ldots, z_n : R_n$ or $z_1 : P_1 \triangleright Q_1, \ldots, z_n : P_n \triangleright Q_n$
- η is an exception context of the form $e_1 : Q'_1, \ldots, e_k : Q'_k$

The rules of the program logic are listed in Figure 5.1 and Figure 5.2. They are explained in detail in Section 5.6. It is assumed that there are some atomic commands a, together with valid Hoare logic axioms for them $\{P\}\, a\, \{Q\}$.

$$\frac{\Sigma, z : \forall\alpha.P_z \triangleright Q_z \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta \qquad \Sigma \vdash \forall\alpha.\{P_z\}\, c_h\, \{Q_z\}\ \texttt{throws}\ \eta \qquad \Sigma\ \textbf{stable}\ \forall\alpha.P_z \triangleright Q_z}{\Sigma \vdash \forall\alpha.\{P \wedge P_z\}\ \texttt{bind/1}\ z\ \texttt{to}\ c_h\ \texttt{in}\ c_B\, \{Q \wedge (P_z \vee Q_z)\}\ \texttt{throws}\ \eta}$$

$$\frac{\Sigma, z : \forall\alpha.I_z \triangleright I_z \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta \qquad \Sigma \vdash \forall\alpha.\{I_z\}\, c_h\, \{I_z\}\ \texttt{throws}\ \eta \qquad \Sigma\ \textbf{stable}\ \forall\alpha.I_z \triangleright I_z}{\Sigma \vdash \forall\alpha.\{P \wedge I_z\}\ \texttt{bind}\ z\ \texttt{to}\ c_h\ \texttt{in}\ c_B\, \{Q \wedge I_z\}\ \texttt{throws}\ \eta}$$

$$\frac{\eta\ \textbf{stable}\ \Sigma \qquad Q\ \textbf{stable}\ \Sigma}{\Sigma \vdash \{\eta(e_j)\}\ \texttt{throw}\ e_j\, \{Q\}\ \texttt{throws}\ \eta}\ (\textsc{Throw})$$

$$\frac{\Sigma \vdash \{P\}\, c_B\, \{Q_b\}\ \texttt{throws}\ \eta, e_k : Q_k \qquad \Sigma \vdash \{Q_k\}\, c_h\, \{Q_h\}\ \texttt{throws}\ \eta}{\Sigma \vdash \{P\}\ \texttt{try}\ c_B\ \texttt{handle}\ e_k\ \texttt{by}\ c_h\, \{Q_b \vee Q_h\}\ \texttt{throws}\ \eta}\ (\text{EH})$$

$$\frac{\Sigma \vdash \{P\}\, c\, \{Q\}\ \texttt{throws}\ \eta \qquad P\ \textbf{stable}\ P_z \triangleright Q_z \qquad Q\ \textbf{stable}\ P_z \triangleright Q_z \qquad \eta\ \textbf{stable}\ P_z \triangleright Q_z}{\Sigma, z : P_z \triangleright Q_z \vdash \{P\}\ \texttt{block}\ z\ \texttt{in}\ c\, \{Q\}\ \texttt{throws}\ \eta}\ (\text{SB})$$

$$\frac{\Sigma \vdash \{P_1\}\, c_1\, \{P_2\}\ \texttt{throws}\ \eta \qquad \Sigma \vdash \{P_2\}\, c_2\, \{P_3\}\ \texttt{throws}\ \eta}{\Sigma \vdash \{P_1\}\, c_1\, ;\, c_2\, \{P_3\}\ \texttt{throws}\ \eta}\ (\textsc{Seq})$$

Figure 5.1: Hoare logic rules for exception and signal handling

$$\frac{\{P\}\, a\, \{Q\} \qquad (P \triangleright Q)\ \mathbf{stable}\ \Sigma \qquad \eta\ \mathbf{stable}\ \Sigma \qquad \forall z.\Sigma(z) = z : \forall \alpha. P_z \triangleright Q_z \qquad \exists E. Q \implies P_z\,[\,\alpha \mapsto E\,]}{\Sigma \vdash \{P\}\, a\, \{Q\}\ \texttt{throws}\ \eta}\ (\textsc{Atomic})$$

$$\frac{\{P\}\, x := E\, \{Q\} \qquad (P \triangleright Q)\ \mathbf{stable}\ \Sigma \qquad \eta\ \mathbf{stable}\ \Sigma \qquad \forall z.\Sigma(z) = z : \forall \alpha. P_z \triangleright Q_z \qquad \exists E. Q \implies P_z\,[\,\alpha \mapsto E\,]}{\Sigma \vdash \{P\}\, x := E\, \{Q\}\ \texttt{throws}\ \eta}\ (\textsc{Assignment})$$

$$\frac{\Sigma \vdash \{I \wedge E_B\}\, c\, \{I\}\ \texttt{throws}\ \eta \qquad \neg E_B\ \mathbf{stable}\ \Sigma}{\Sigma \vdash \{I\}\ \texttt{while}\ E_B\ \texttt{do}\ c\, \{I \wedge \neg E_B\}\ \texttt{throws}\ \eta}\ (\textsc{WhileRule})$$

$$\frac{\Sigma \vdash \{P\}\, c\, \{Q\}\ \texttt{throws}\ \eta \quad P' \Rightarrow P \quad Q \Rightarrow Q' \quad \eta \Rightarrow \eta' \qquad P'\ \mathbf{stable}\ \Sigma \qquad Q'\ \mathbf{stable}\ \Sigma \qquad \eta'\ \mathbf{stable}\ \Sigma}{\Sigma \vdash \{P'\}\, c\, \{Q'\}\ \texttt{throws}\ \eta'}\ (\textsc{Conseq})$$

$$\frac{\Sigma \vdash \{P_1\}\, c\, \{Q_1\}\ \texttt{throws}\ \eta_1 \qquad \Sigma \vdash \{P_2\}\, c\, \{Q_2\}\ \texttt{throws}\ \eta_2}{\Sigma \vdash \{P_1 \wedge P_2\}\, c\, \{Q_1 \wedge Q_2\}\ \texttt{throws}\ \eta_1 \wedge \eta_2}\ (\textsc{Conj})$$

$$\frac{\Sigma \vdash \{E_B \wedge P\}\, c_1\, \{Q\}\ \texttt{throws}\ \eta \qquad \Sigma \vdash \{\neg E_B \wedge P\}\, c_2\, \{Q\}\ \texttt{throws}\ \eta}{\Sigma \vdash \{P\}\ \texttt{if}\ E_B\ \texttt{then}\ c_1\ \texttt{else}\ c_2\, \{Q\}\ \texttt{throws}\ \eta}\ (\text{IE})$$

Figure 5.2: Hoare logic rules for exception and signal handling 2

5.5 Supporting Lemmas

For the signal handling rule, we will need a lemma:

Lemma 5.5.1 If a judgement $\Sigma \vdash \{P\}\, c\, \{Q\}\ \texttt{throws}\ \eta$ is derivable, then P **stable** Σ, Q **stable** Σ, $(P \triangleright Q)$ **stable** Σ and η **stable** Σ.

Proof By induction over the derivation of a program logic judgement

$$\Sigma \vdash \{P\}\, c\, \{Q\}\ \texttt{throws}\ \eta$$

We consider all the cases how the proof tree could be built up (Figure 5.1 and Figure 5.2).

One-shot signal binding We consider the program logic rule for the one-shot signal binding:

$$\frac{\Sigma, z : \forall\alpha.P_z \triangleright Q_z \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta \qquad \Sigma \vdash \forall\alpha.\{P_z\}\, c_h\, \{Q_z\}\ \texttt{throws}\ \eta \qquad \Sigma\ \textbf{stable}\ \forall\alpha.P_z \triangleright Q_z}{\Sigma \vdash \forall\alpha.\{P \wedge P_z\}\ \texttt{bind/1}\ z\ \texttt{to}\ c_h\ \texttt{in}\ c_B\, \{Q \wedge (P_z \vee Q_z)\}\ \texttt{throws}\ \eta}$$

Suppose the following judgments are derivable: $\Sigma \vdash \forall\alpha.\{P_z\}\, c_h\, \{Q_z\}\ \texttt{throws}\ \eta$ and $\Sigma, z : \forall\alpha.P_z \triangleright Q_z \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta$. Also suppose that Σ **stable** $\forall\alpha.P_z \triangleright Q_z$ holds. By the induction hypothesis for c_B, we get P **stable** $\Sigma, z : \forall\alpha.P_z \triangleright Q_z$, Q **stable** $\Sigma, z : \forall\alpha.P_z \triangleright Q_z$ and η **stable** $\Sigma, z : \forall\alpha.P_z \triangleright Q_z$. Then, we infer P **stable** Σ, Q **stable** Σ and η **stable** Σ. By the induction hypothesis for c_h, we get P_z **stable** Σ and Q_z **stable** Σ. Suppose $s_1 \models Q$. Either $s_1 \models Q_z$ or $s_1 \models P_z$. So either $s_1 \models Q \wedge Q_z$ or $s_1 \models Q \wedge P_z$. Both are stable so either $s_2 \models Q \wedge Q_z$ or $s_2 \models Q \wedge (P_z \vee Q_z)$ due to signals' nondeterminism. Thus, for the judgment $\Sigma \vdash \forall\alpha.\{P \wedge P_z\}\ \texttt{bind/1}\ z\ \texttt{to}\ c_h\ \texttt{in}\ c_B\, \{Q \wedge (P_z \vee Q_z)\}\ \texttt{throws}\ \eta$ we have $P \wedge P_z$ **stable** Σ, $Q \wedge (P_z \vee Q_z)$ **stable** Σ and η **stable** Σ as required.

Persistent signal binding We consider the program logic rule for the persistent signal binding:

$$\frac{\Sigma, z : \forall\alpha.I_z \triangleright I_z \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta \qquad \Sigma \vdash \forall\alpha.\{I_z\}\, c_h\, \{I_z\}\ \texttt{throws}\ \eta \qquad \Sigma\ \textbf{stable}\ \forall\alpha.I_z \triangleright I_z}{\Sigma \vdash \forall\alpha.\{P \wedge I_z\}\ \texttt{bind}\, z\, \texttt{to}\, c_h\, \texttt{in}\, c_B\, \{Q \wedge I_z\}\ \texttt{throws}\ \eta}$$

Suppose the judgments $\Sigma \vdash \forall\alpha.\{I_z\}\, c_h\, \{I_z\}\ \texttt{throws}\ \eta$ and $\Sigma, z : \forall\alpha.I_z \triangleright I_z \vdash \{P\}\, c_B\, \{Q\}\texttt{throws}\eta$ are derivable. Also suppose that Σ **stable** $\forall\alpha.I_z \triangleright I_z$ holds. By the induction hypothesis for c_B, we get P **stable** $\Sigma, z : \forall\alpha.I_z \triangleright I_z$, Q **stable** $\Sigma, z : \forall\alpha.I_z \triangleright I_z$ and η **stable** $\Sigma, z : \forall\alpha.I_z \triangleright I_z$. Then, we infer P **stable** Σ, Q **stable** Σ and η **stable** Σ. By the induction hypothesis for c_h, we get I_z **stable** Σ. Thus, for the judgment $\Sigma \vdash \forall\alpha.\{P \wedge I_z\}\, \texttt{bind}\, z\, \texttt{to}\, c_h\, \texttt{in}\, c_B\, \{Q \wedge I_z\}\ \texttt{throws}\ \eta$ we have $P \wedge I_z$ **stable** Σ, $Q \wedge I_z$ **stable** Σ and η **stable** Σ as required.

Atomic command We consider the program logic rule for the atomic command:

$$\frac{\{P\}\, a\, \{Q\} \qquad (P \triangleright Q)\ \textbf{stable}\ \Sigma \qquad \eta\ \textbf{stable}\ \Sigma \qquad \forall z.\Sigma(z) = z : \forall\alpha.P_z \triangleright Q_z \qquad \exists E.Q \implies P_z\,[\,\alpha \mapsto E\,]}{\Sigma \vdash \{P\}\, a\, \{Q\}\ \texttt{throws}\ \eta}$$

Suppose the judgment $\{P\}\, a\, \{Q\}$ is derivable. Also suppose that $(P \triangleright Q)$ **stable** Σ and η **stable** Σ hold. Thus, for the judgment $\Sigma \vdash \{P\}\, a\, \{Q\}\ \texttt{throws}\ \eta$ we have $(P \triangleright Q)$ **stable** Σ and η **stable** Σ as required.

Sequential composition We consider the program logic rule for the sequential composition:

$$\frac{\Sigma \vdash \{P_1\}\, c_1\, \{P_2\}\ \texttt{throws}\ \eta \qquad \Sigma \vdash \{P_2\}\, c_2\, \{P_3\}\ \texttt{throws}\ \eta}{\Sigma \vdash \{P_1\}\, c_1\, ;\, c_2\, \{P_3\}\ \texttt{throws}\ \eta}$$

Suppose that the judgments $\Sigma \vdash \{P_1\}\, c_1\, \{P_2\}\, \texttt{throws}\, \eta$ and $\Sigma \vdash \{P_2\}\, c_2\, \{P_3\}\, \texttt{throws}\ \eta$ are derivable. By the induction hypothesis for c_1 and c_2, we get P_1 **stable** Σ, P_2 **stable** Σ, P_3 **stable** Σ, and η **stable** Σ.

Thus, for $\Sigma \vdash \{P_1\}\, c_1\, ;\, c_2\, \{P_3\}\ \texttt{throws}\ \eta$ we have P_1 **stable** Σ, P_3 **stable** Σ and η **stable** Σ as required.

Rule of consequence We consider the program logic rule:

$$\frac{\Sigma \vdash \{P\}\, c\, \{Q\}\ \texttt{throws}\ \eta \quad P' \Rightarrow P \quad Q \Rightarrow Q' \quad \eta \Rightarrow \eta' \qquad P' \textbf{ stable } \Sigma \qquad Q' \textbf{ stable } \Sigma \qquad \eta' \textbf{ stable } \Sigma}{\Sigma \vdash \{P'\}\, c\, \{Q'\}\ \texttt{throws}\ \eta'}$$

Suppose the judgment $\Sigma \vdash \{P\}\, c\, \{Q\}\ \texttt{throws}\ \eta$ is derivable. Also suppose that $P' \Rightarrow P$, $Q \Rightarrow Q'$, $\eta \Rightarrow \eta'$, P' **stable** Σ, Q' **stable** Σ and η' **stable** Σ hold.

Thus, for the judgment $\Sigma \vdash \{P'\}\, c\, \{Q'\}\, \texttt{throws}\, \eta'$ we have P' **stable** Σ, Q' **stable** Σ and η' **stable** Σ as they were assumed.

Rule of conjunction We consider the program logic rule:

$$\frac{\Sigma \vdash \{P_1\}\, c\, \{Q_1\}\ \texttt{throws}\ \eta_1 \qquad \Sigma \vdash \{P_2\}\, c\, \{Q_2\}\ \texttt{throws}\ \eta_2}{\Sigma \vdash \{P_1 \wedge P_2\}\, c\, \{Q_1 \wedge Q_2\}\ \texttt{throws}\ \eta_1 \wedge \eta_2}$$

Suppose that the judgments $\Sigma \vdash \{P_1\}\, c\, \{Q_1\}\, \texttt{throws}\, \eta_1$ and $\Sigma \vdash \{P_2\}\, c\, \{Q_2\}\, \texttt{throws}\ \eta_2$ are derivable.

By the induction hypothesis for c, we get P_1 **stable** Σ, Q_1 **stable** Σ, η_1 **stable** Σ. Then we apply the induction hypothesis for c once again. We get P_2 **stable** Σ, Q_2 **stable** Σ and η_2 **stable** Σ. That is equivalent to $P_1 \wedge P_2$ **stable** Σ, $Q_1 \wedge Q_2$ **stable** Σ and $\eta_1 \wedge \eta_2$ **stable** Σ.

Thus, for the judgment $\Sigma \vdash \{P_1 \wedge P_2\}\, c\, \{Q_1 \wedge Q_2\}\ \texttt{throws}\ \eta_1 \wedge \eta_2$ we have $P_1 \wedge P_2$ **stable** Σ, $Q_1 \wedge Q_2$ **stable** Σ and $\eta_1 \wedge \eta_2$ **stable** Σ as required.

Conditional `if` structure We consider the program logic rule:

$$\frac{\Sigma \vdash \{E \wedge P\}\, c_1\, \{Q\}\ \texttt{throws}\ \eta \qquad \Sigma \vdash \{\neg E \wedge P\}\, c_2\, \{Q\}\ \texttt{throws}\ \eta}{\Sigma \vdash \{P\}\ \texttt{if}\ E\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \{Q\}\ \texttt{throws}\ \eta}$$

Suppose the judgments $\Sigma \vdash \{E \wedge P\}\, c_1\, \{Q\}\ \texttt{throws}\ \eta$ and $\Sigma \vdash \{\neg E \wedge P\}\, c_2\, \{Q\}\ \texttt{throws}\ \eta$ are derivable.

By the induction hypothesis for c_1, we get $E \wedge P$ **stable** Σ, Q **stable** Σ and η **stable** Σ. Then, by the induction hypothesis for c_2, we get $\neg E \wedge P$ **stable** Σ, Q **stable** Σ and η **stable** Σ.

From $E \wedge P$ **stable** Σ and $\neg E \wedge P$ **stable** Σ we infer that P **stable** Σ.

Thus, for the judgment $\Sigma \vdash \{P\}\ \texttt{if}\ E\ \texttt{then}\ c_1\ \texttt{else}\ c_2\, \{Q\}\ \texttt{throws}\ \eta$ we have P **stable** Σ, Q **stable** Σ and η **stable** Σ as required.

Repetitive `while` command We consider the program logic rule for the `while`:

$$\frac{\Sigma \vdash \{I \wedge E\}\, c\, \{I\}\ \texttt{throws}\ \eta \qquad \neg E\ \textbf{stable}\ \Sigma}{\Sigma \vdash \{I\}\ \texttt{while}\ E\ \texttt{do}\ c\, \{I \wedge \neg E\}\ \texttt{throws}\ \eta}$$

Suppose the judgment $\Sigma \vdash \{I \wedge E\}\, c\, \{I\}\ \texttt{throws}\ \eta$ is derivable and $\neg E$ **stable** Σ holds.

By the induction hypothesis for c, we get $I \wedge E$ **stable** Σ, I **stable** Σ and η **stable** Σ. It was assumed that $\neg E$ **stable** Σ; together with I **stable** Σ it is equivalent to $I \wedge \neg E$ **stable** Σ.

Thus, for the judgment $\Sigma \vdash \{I\}\ \texttt{while}\ E\ \texttt{do}\ c\, \{I \wedge \neg E\}\ \texttt{throws}\ \eta$ we have I **stable** Σ, $I \wedge \neg E$ **stable** Σ and η **stable** Σ as required.

Signal blocking We consider the program logic rule for the signal blocking:

$$\frac{\Sigma \vdash \{P\}\, c\, \{Q\}\ \texttt{throws}\ \eta \qquad P\ \textbf{stable}\ \forall\alpha.P_z \triangleright Q_z \qquad Q\ \textbf{stable}\ \forall\alpha.P_z \triangleright Q_z \qquad \eta\ \textbf{stable}\ \forall\alpha.P_z \triangleright Q_z}{\Sigma, z : \forall\alpha.P_z \triangleright Q_z \vdash \{P\}\, \texttt{block}\, z\, \texttt{in}\, c\, \{Q\}\ \texttt{throws}\ \eta}$$

Suppose the judgment $\Sigma \vdash \{P\}\, c\, \{Q\}\ \texttt{throws}\ \eta$ is derivable. Also suppose that P **stable** $\forall\alpha.P_z \triangleright Q_z$, Q **stable** $\forall\alpha.P_z \triangleright Q_z$ and η **stable** $\forall\alpha.P_z \triangleright Q_z$. By the induction hypothesis for c, we get P **stable** Σ, Q **stable** Σ and η **stable** Σ. It was assumed that P **stable** $\forall\alpha.P_z \triangleright Q_z$, Q **stable** $\forall\alpha.P_z \triangleright Q_z$ and η **stable** $\forall\alpha.P_z \triangleright Q_z$. Thus, for the judgment $\Sigma, z : \forall\alpha.P_z \triangleright Q_z \vdash \{P\}\, \texttt{block}\, z\, \texttt{in}\, c\, \{Q\}\ \texttt{throws}\ \eta$ we have P **stable** $\Sigma, z : \forall\alpha.P_z \triangleright Q_z$, Q **stable** $\Sigma, z : \forall\alpha.P_z \triangleright Q_z$ and η **stable** $\Sigma, z : \forall\alpha.P_z \triangleright Q_z$ as required.

Exception `throw` command We consider the program rule for the `throw`; we could also use false as the postcondition of throw e, and false is automatically stable.

$$\frac{\eta\ \textbf{stable}\ \Sigma \qquad Q\ \textbf{stable}\ \Sigma}{\Sigma \vdash \{Q'_j\}\, \texttt{throw}\, e_j\, \{Q\}\ \texttt{throws}\ \eta}$$

where $\texttt{throws}\ \eta = \texttt{throws}\ e_1 : Q'_1, \ldots, e_n : Q'_n$. And $\eta(e_j) = Q'_j$ in a precondition position, means that Q'_j holds immediately before the exception is thrown.

Suppose that Q **stable** Σ holds. Also suppose that η **stable** Σ holds, which trivially implies that Q'_j **stable** Σ.

Thus, for the judgment $\Sigma \vdash \{Q'_j\}\, \texttt{throw}\, e_j\, \{Q\}\ \texttt{throws}\ \eta$ we have Q'_j **stable** Σ, Q **stable** Σ and η **stable** Σ as required.

Exception handling We consider the program rule for the exception handling:

$$\frac{\Sigma \vdash \{P\}\, c_B\, \{Q_b\}\ \texttt{throws}\ \eta, e_k : Q_k \qquad \Sigma \vdash \{Q_k\}\, c_h\, \{Q_h\}\ \texttt{throws}\ \eta}{\Sigma \vdash \{P\}\, \texttt{try}\ c_B\ \texttt{handle}\ e_k\ \texttt{by}\ c_h\, \{Q_b \vee Q_h\}\ \texttt{throws}\ \eta}$$

Suppose the judgments $\Sigma \vdash \{P\}\, c_B\, \{Q_b\}\, \texttt{throws}\, \eta, e_k : Q_k$ and
$\Sigma \vdash \{Q_k\}\, c_h\, \{Q_h\}\, \texttt{throws}\, \eta$ are derivable.

By the induction hypothesis for c_B, we get P **stable** Σ, Q_b **stable** Σ and $\eta, e_k : Q_k$ **stable** Σ. By the induction hypothesis for c_h, we get Q_k **stable** Σ, Q_h **stable** Σ and η **stable** Σ.

For the judgment $\Sigma \vdash \{P\}\, \texttt{try}\ c_B\ \texttt{handle}\ e_k\ \texttt{by}\ c_h\, \{Q_b \vee Q_h\}\, \texttt{throws}\, \eta$ we have P **stable** Σ, $Q_b \vee Q_h$ **stable** Σ and η **stable** Σ as required.

□

5.6 Logic Rules in Detail

In this section, we discuss the most interesting and important logic rules. The full list of logic rules is given in Figure 5.1 and Figure 5.2.

5.6.1 Atomic and Assignment

$$\{P\}\, a\, \{Q\}$$

$$(P \triangleright Q)\ \textbf{stable}\ \Sigma \qquad \eta\ \textbf{stable}\ \Sigma$$

$$\frac{\forall z.\Sigma(z) = z : \forall \alpha.P_z \triangleright Q_z \qquad \exists E.Q \implies P_z\,[\,\alpha \mapsto E\,]}{\Sigma \vdash \{P\}\, a\, \{Q\}\, \texttt{throws}\, \eta}$$

When an atomic command is lifted into a signal context, all stability assumptions need to be established. The signals may happen before or after the atomic command, thus we need to ensure stability of P and Q under all handlers. That is the price one has to pay for the stability of the implicit pre- and postconditions.

We also need to ensure stability of all signals' preconditions under the actions of the atomic command a. Intuitively, $\forall z.\Sigma(z) = z : \forall \alpha.P_z \triangleright Q_z \quad \exists E.Q \implies P_z\,[\,\alpha \mapsto E\,]$

means that the atomic command a is limited to change the state in a way that the preconditions of the signals still hold.

Finally, we need to ensure stability of the exception context η under the signal context Σ. Preconditions for every "registered" exception are stored in the exception context η. Therefore, the exception context η should be stable under actions of the signal handlers. Without that stability assumption we can't guarantee that the required precondition for some particular exception holds.

Stability of the signals' postconditions is not explicitly covered in the atomic rule. The importance of holding signal handlers' postconditions after an execution of the atomic command could be considered as a design choice. Still, in the rest of the logic we develop, a particular attention has been paid to that bit. For example, in the Definition 5.3.1, for `pairstability` we explicitly require some conditions to hold for the postconditions. As it is rather important to maintain conditions on the signals' postconditions in the logic rules implicitly or explicitly, we tried to design the rest of the rules in such a way, that all required limitations for the persistent signals' invariant are in place via enforcing them on the preconditions, and the postconditions become implicitly covered, as they are identical to preconditions.

$$\{P\}\, x := E' \,\{Q\}$$

$$(P \triangleright Q)\ \textbf{stable}\ \Sigma \qquad \eta\ \textbf{stable}\ \Sigma$$

$$\frac{\forall z.\Sigma(z) = z : \forall \alpha.P_z \triangleright Q_z \qquad \exists E.Q \implies P_z\,[\,\alpha \mapsto E\,]}{\Sigma \vdash \{P\}\, x := E' \,\{Q\}\ \texttt{throws}\ \eta}\ (\textsc{Assignment})$$

An assignment command is an instance of the atomic command. Signals could be handled before or after the assignment command, but not during it.

5.6.2 Sequential Composition

$$\frac{\Sigma \vdash \{P_1\}\, c_1\, \{P_2\}\ \texttt{throws}\ \eta \qquad \Sigma \vdash \{P_2\}\, c_2\, \{P_3\}\ \texttt{throws}\ \eta}{\Sigma \vdash \{P_1\}\, c_1\, ;\, c_2\, \{P_3\}\ \texttt{throws}\ \eta}$$

In sequential composition $c_1; c_2$, there is no need for explicitly stating that the postcondition of the first command c_1 could be subject to interference by the signal handlers. The judgement for c_1 already takes that interference, and the need for stability, into account.

5.6.3 One-shot Signal Binding

$$\Sigma, z : \forall\alpha.P_z \triangleright Q_z \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta \qquad \Sigma \vdash \forall\alpha.\{P_z\}\, c_h\, \{Q_z\}\ \texttt{throws}\ \eta$$

$$\frac{\Sigma\ \textbf{stable}\ \forall\alpha.P_z \triangleright Q_z}{\Sigma \vdash \forall\alpha.\{P \wedge P_z\}\ \texttt{bind/1}\ z\ \texttt{to}\ c_h\ \texttt{in}\ c_B\, \{Q \wedge (P_z \vee Q_z)\}\ \texttt{throws}\ \eta}$$

When a new handler is installed, stability needs to be checked for the new signal handler. As we bind new signal to the command block, signal's pre- post- conditions should be stable under actions of already bound signals. This requirement is embedded into the $\Sigma \vdash \forall\alpha.\{P_z\}\, c_h\, \{Q_z\}\ \texttt{throws}\ \eta$ premise, which is required to be derivable. From that premise and the Lemma 5.5.1, we infer P_z **stable** Σ, Q_z **stable** Σ, and η **stable** Σ. At the same time, already bound signals should be stable under the action of a new handler. That is the reason why program logic rules for the signal binding contain implicit and explicit stability assumptions. One can notice when a new signal binding is added, body of a command c_B should be stable under the actions of a new signal handler. When we prove a judgement for the command c_B, all the atomic commands are checked against the signal specification on the left of the $\vdash$. The Σ is passed up in the tree all the way to the atomic commands that make up the body.

5.6.4 Persistent Signal Binding

$$\Sigma, z : \forall\alpha.I_z \triangleright I_z \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta \qquad \Sigma \vdash \forall\alpha.\{I_z\}\, c_h\, \{I_z\}\ \texttt{throws}\ \eta$$

$$\frac{\Sigma\ \textbf{stable}\ \forall\alpha.I_z \triangleright I_z}{\Sigma \vdash \forall\alpha.\{P \wedge I_z\}\ \texttt{bind}\ z\ \texttt{to}\ c_h\ \texttt{in}\ c_B\, \{Q \wedge I_z\}\ \texttt{throws}\ \eta}$$

Analogously to the one-shot signal binding rule, when a new handler is installed, stability needs to be checked for the new signal handler. As we bind new signal to the command block, signal's pre- post- conditions should be stable under actions of already bound signals. One may say that the following stability assumptions are missing: I_z **stable** Σ and η **stable** Σ. They are implicitly covered by the fact that the $\Sigma \vdash \forall\alpha.\{I_z\}\, c_h\, \{I_z\}\ \texttt{throws}\ \eta$ is derivable. Thus, we infer all the above mentioned stability assumptions using Lemma 5.5.1. Satisfied stability assumptions ensure that the I_z wouldn't be corrupted by the actions of the main command c_B and the other signals.

5.6.5 Persistent Versus One-shot Signal Binding

One may ask, why are there two separate rules for the persistent and the one-shot signal bindings? Technically, a rule for the persistent signal binding could be constructed from the one-shot rule, by taking $P = Q$. In logic, we could verify the same programs by replacing `bind` by `bind/1`. However, we would lose expressivity since the `bind/1` programs omit some behaviours of the `bind`. That becomes clear in Figure 5.3 and Figure 5.4, where we compare `bind` and `bind/1` in operational semantics. A signal handler for a signal z bound with `bind` may appear in both subtrees for c_1 and c_2. On the other hand, a signal handler bound with `bind/1` may appear only in one of the subtrees for c_1 or c_2.

$$\frac{\dfrac{S\,[\,z \mapsto c_h\,]; O_1 \Vdash s_1, c_1 \Downarrow s_2 \qquad S\,[\,z \mapsto c_h\,]; O_2 \Vdash s_2, c_2 \Downarrow s_3}{S\,[\,z \mapsto c_h\,]; O_1 * O_2 \Vdash s_1, (c_1\ ;\ c_2) \Downarrow s_3}}{S; O_1 * O_2 \Vdash s_1, \texttt{bind}\ z\ \texttt{to}\ c_h\ \texttt{in}\ (c_1\ ;\ c_2) \Downarrow s_3}$$

$$\frac{\dfrac{S; O_1\,[\,z \mapsto c_h\,] \Vdash s_1, c_1 \Downarrow s_2 \qquad S; O_2 \Vdash s_2, c_2 \Downarrow s_3}{S; O_1 * O_2\,[\,z \mapsto c_h\,] \Vdash s_1, (c_1\ ;\ c_2) \Downarrow s_3}}{S; O_1 * O_2 \Vdash s_1, \texttt{bind/1}\ z\ \texttt{to}\ c_h\ \texttt{in}\ (c_1\ ;\ c_2) \Downarrow s_3}$$

$$\frac{\dfrac{S; O_1 \Vdash s_1, c_1 \Downarrow s_2 \qquad S; O_2\,[\,z \mapsto c_h\,] \Vdash s_2, c_2 \Downarrow s_3}{S; O_1 * O_2\,[\,z \mapsto c_h\,] \Vdash s_1, (c_1\ ;\ c_2) \Downarrow s_3}}{S; O_1 * O_2 \Vdash s_1, \texttt{bind/1}\ z\ \texttt{to}\ c_h\ \texttt{in}\ (c_1\ ;\ c_2) \Downarrow s_3}$$

Figure 5.3: Persistent and one-shot binding derivations

$$\frac{\dfrac{[\,z \mapsto c_h\,]; \emptyset \Vdash s_1, c_1 \Downarrow s_2 \qquad [\,z \mapsto c_h\,]; \emptyset \Vdash s_2, c_2 \Downarrow s_3}{[\,z \mapsto c_h\,]; \emptyset \Vdash (c_1\ ;\ c_2), s_3 \Downarrow}}{\emptyset; \emptyset \Vdash \texttt{bind}\ z\ \texttt{to}\ c_h\ \texttt{in}\ (c_1\ ;\ c_2), s_3 \Downarrow}$$

$$\frac{\dfrac{\emptyset; [\,z \mapsto c_h\,] \Vdash s_1, c_1 \Downarrow s_2 \qquad \emptyset; \emptyset \Vdash s_2, c_2 \Downarrow s_3}{\emptyset; [\,z \mapsto c_h\,] \Vdash s_1, (c_1\ ;\ c_2) \Downarrow s_3}}{\emptyset; \emptyset \Vdash s_1, \texttt{bind/1}\ z\ \texttt{to}\ c_h\ \texttt{in}\ (c_1\ ;\ c_2) \Downarrow s_3}$$

$$\frac{\dfrac{\emptyset; \emptyset \Vdash s_1, c_1 \Downarrow s_2 \qquad \emptyset; [\,z \mapsto c_h\,] \Vdash s_2, c_2 \Downarrow s_3}{\emptyset; [\,z \mapsto c_h\,] \Vdash s_1, (c_1\ ;\ c_2) \Downarrow s_3}}{\emptyset; \emptyset \Vdash s_1, \texttt{bind/1}\ z\ \texttt{to}\ c_h\ \texttt{in}\ (c_1\ ;\ c_2) \Downarrow s_3}$$

Figure 5.4: Persistent and one-shot binding examples

5.6.6 Signal Blocking

$$\Sigma \vdash \{P\}\, c\, \{Q\}\, \texttt{throws}\, \eta$$

$$\frac{P\ \textbf{stable}\ \forall\alpha.P_z \triangleright Q_z \qquad Q\ \textbf{stable}\ \forall\alpha.P_z \triangleright Q_z \qquad \eta\ \textbf{stable}\ \forall\alpha.P_z \triangleright Q_z}{\Sigma, z : \forall\alpha.P_z \triangleright Q_z \vdash \{P\}\, \texttt{block}\, z\, \texttt{in}\, c\, \{Q\}\, \texttt{throws}\, \eta}$$

One may ask what the blocking has achieved if we still need the stability assumption above the line. What blocking does is to relieve us from having to check all atomic commands inside the body of the block against the blocked signal. Let consider an example c_1 ; $\texttt{block}\, z\, \texttt{in}\, (c_2$; $c_3)$; c_4. The signal cannot jump into the middle of the command c_2 ; c_3 when it is blocked, but it could still be handled right at the beginning or at the end. The point is that the condition P between c2 and c3 does not have to be stable under the blocked signal. Without `block z`, we would have to check that it is. The rule for blocking is intended for both types of signals. For the persistent signals, the rule has a special form where $P = I_z$, $Q = I_z$, and $z : I_z \triangleright I_z$. A signal context Σ, contains specifications of persistent and one-shot signal handlers. Thus, the same signal name cannot be used for both types of signals simultaneously.

5.6.7 Exception Throwing/Raising and Handling

$$\frac{\eta\ \textbf{stable}\ \Sigma \qquad Q\ \textbf{stable}\ \Sigma}{\Sigma \vdash \{\eta(e_j)\}\, \texttt{throw}\, e_j\, \{Q\}\, \texttt{throws}\, \eta}$$

We could also use **false** as the postcondition of $\texttt{throw}\, e_j$, and **false** is automatically stable.

Preconditions for every "registered" exception are stored in the exception context η. Therefore, the exception context η should be stable under actions of the signal handlers. Without that stability assumption we can't guarantee that the required precondition for some particular exception holds.

$$\frac{\Sigma \vdash \{P\}\, c_B\, \{Q_b\}\ \texttt{throws}\ \eta, e_k : Q_k \qquad \Sigma \vdash \{Q_k\}\, c_h\, \{Q_h\}\ \texttt{throws}\ \eta}{\Sigma \vdash \{P\}\ \texttt{try}\ c_B\ \texttt{handle}\ e_k\ \texttt{by}\ c_h\, \{Q_b \vee Q_h\}\ \texttt{throws}\ \eta}$$

When a new exception is registered, stability needs to be checked for it. Stability is embedded into the requirement of $\Sigma \vdash \{P\}\, c_B\, \{Q_b\}\ \texttt{throws}\ \eta, e_k : Q_k$ to be derivable.

If c_B doesn't throw an exception e_k, then the final state satisfies Q_b. If c_B actually throws an exception, it is covered by the right branch of the rule. Firs of all, a corresponding precondition Q_k should hold before exception is raised. Then, the state after execution of the exception handler should satisfy Q_h. This rule of exception registering and handling covers both outcomes. Therefore, there is the next postcondition $Q_b \vee Q_h$.

5.6.8 Repetitive `while` Construct

$$\frac{\Sigma \vdash \{I \wedge E_B\}\, c\, \{I\}\ \texttt{throws}\ \eta \qquad \neg E_B\ \textbf{stable}\ \Sigma}{\Sigma \vdash \{I\}\ \texttt{while}\ E_B\ \texttt{do}\ c\, \{I \wedge \neg E_B\}\ \texttt{throws}\ \eta}$$

A `while` command is not atomic. Thus, boolean expressions E_B and $\neg E_B$ could be corrupted by the signals, after their values have been changed but before the next check in loop. Therefore, our rule enforces stability assumptions for the control expressions.

The next stability assumption E_B **stable** Σ is embedded into the premise $\Sigma \vdash \{I \wedge E_B\}\, c\, \{I\}\ \texttt{throws}\ \eta$. And $\neg E_B$ **stable** Σ is added explicitly.

5.6.9 Conditional `if` Construct

$$\frac{\Sigma \vdash \{E_B \wedge P\}\, c_1\, \{Q\}\ \texttt{throws}\ \eta \qquad \Sigma \vdash \{\neg E_B \wedge P\}\, c_2\, \{Q\}\ \texttt{throws}\ \eta}{\Sigma \vdash \{P\}\ \texttt{if}\ E_B\ \texttt{then}\ c_1\ \texttt{else}\ c_2\, \{Q\}\ \texttt{throws}\ \eta}$$

Stability of control booleans is embedded into the premises. Left branch covers the case when a control boolean is `true`; and the right branch is for the case when the control boolean is `false`.

5.6.10 Rule of Consequence

$$\Sigma \vdash \{P\}\, c\, \{Q\}\ \texttt{throws}\ \eta \quad P' \Rightarrow P \quad Q \Rightarrow Q' \quad \eta \Rightarrow \eta'$$

$$\frac{P' \textbf{ stable } \Sigma \qquad Q' \textbf{ stable } \Sigma \qquad \eta' \textbf{ stable } \Sigma}{\Sigma \vdash \{P'\}\, c\, \{Q'\}\ \texttt{throws}\ \eta'}$$

If P' implies P, command c changes the state which satisfies P into the state satisfying Q, and Q implies Q', then we write $\{P'\}\, c\, \{Q'\}$. Stability assumptions for P and Q are embedded into the $\Sigma \vdash \{P\}\, c\, \{Q\}\ \texttt{throws}\ \eta$. Stability assumptions for P' and Q' are given explicitly, to ensure that they are not corrupted by the signals.

5.6.11 Rule of Conjunction

$$\frac{\Sigma \vdash \{P_1\}\, c\, \{Q_1\}\ \texttt{throws}\ \eta_1 \qquad \Sigma \vdash \{P_2\}\, c\, \{Q_2\}\ \texttt{throws}\ \eta_2}{\Sigma \vdash \{P_1 \wedge P_2\}\, c\, \{Q_1 \wedge Q_2\}\ \texttt{throws}\ \eta_1 \wedge \eta_2}$$

A command c runs from an initial state to the final state and may terminate normally of with an exception e. If we know that for the command run, the initial state satisfies P_1, and the final state satisfies Q_1 or $\eta_1(e)$. If we also know that the initial state satisfies P_2, and the final state satisfies Q_2 or $\eta_2(e)$. Then we conclude that the initial state satisfies $P_1 \wedge P_2$ and the final state satisfies $Q_1 \wedge Q_2$ or $(\eta_1 \wedge \eta_2)(e)$.

5.7 Ghost Variables

P, Q and I are unary predicates; therefore, they can not describe the relationship between different states on their own. We introduce ghost variables to explicitly relate pre- and post- states. Ghost variables are also known as "logical variables", but should not be confused with "auxiliary variables" [97]. The main requirement is that they do not appear in the program body; they are opposed to the program variables. In our notation, we use Greek alphabet to represent ghost variables. Analogously to [85], we define ghost

variables as variables that occur in command's specification but do not occur in the body of the specified command. For example, $\alpha \notin \mathcal{FV}(c)$. One may note that there are two ways of relating pre and post states:

- $P \triangleright Q$, where both P is interpreted in the pre and Q in the post state (Definition 5.1.1)
- a single predicate R with primed variables x', where x is interpreted in the pre state and x' in the post state

Ghost variables can be used to translate between the two forms. For instance, $x' = x + 1$ could be translated to $\forall\alpha.((x = \alpha) \triangleright (x = \alpha + 1))$.

5.7.1 Quantification and Instantiation of the Ghost Variables

In our logic, quantification of the ghost variables is both implicit and explicit. Thus, for $\forall\alpha.(\{x = \alpha\}\, c\, \{y = \alpha\})$ we may write $\{x = \alpha\}\, c\, \{y = \alpha\}$. It should be noted that quantification never appears inside of the Hoare triples.

For the ghost variable instantiation one may use one of the following rules.

$$\frac{P \textbf{ stable } \forall\alpha.R}{P \textbf{ stable } (R\,[\,\alpha \mapsto E\,])} \qquad \frac{P \textbf{ stable } R \qquad \alpha \notin \mathcal{FV}(P)}{P \textbf{ stable } (R\,[\,\alpha \mapsto E\,])}$$

For the stability assumption P **stable** R, we can replace all occurrences of α in R with an expression E. It is implicitly assumed that the ghost variable α is not free in P.

$$\frac{\forall\alpha.\{P\}\, c\, \{Q\}}{\{P\,[\,\alpha \mapsto E\,]\}\, c\, \{Q\,[\,\alpha \mapsto E\,]\}} \qquad \frac{\{P\}\, c\, \{Q\} \qquad \alpha \notin \mathcal{FV}(c)}{\{P\,[\,\alpha \mapsto E\,]\}\, c\, \{Q\,[\,\alpha \mapsto E\,]\}}$$

If the ghost variable α is not free in the command c, then we can replace all its occurrences in precondition P and postcondition Q with a required expression.

5.7.2 Example with Quantified Ghost Variables

Let's inspect the next example

$$z : R_1 \vdash \{(x = \beta)\}\, c_B\, \{(x = \beta)\}$$

where both c_b and c_h are equal to $(x{+}{+}; x{-}{-};)$, and R_1 stays for $\forall\alpha.(x = \alpha) \triangleright (x = \alpha)$.

$$\frac{\forall\alpha.(x = \alpha) \triangleright (x = \alpha) \vdash \{(x = \beta)\}\, x{+}{+};\, \{(x = \beta + 1)\} \qquad \forall\alpha.(x = \alpha) \triangleright (x = \alpha) \vdash \{(x = \beta + 1)\}\, x{-}{-};\, \{(x = \beta)\}}{\forall\alpha.(x = \alpha) \triangleright (x = \alpha) \vdash \{(x = \beta)\}\, x{+}{+}; x{-}{-};\, \{(x = \beta)\}}$$

We will consider every branch in a separate tree.

Left branch:

$$\{(x = \beta)\}\, x{+}{+};\, \{(x = \beta + 1)\}$$

$$((x = \beta) \triangleright (x = \beta + 1))\ \textbf{stable}\ (\forall\alpha.(x = \alpha) \triangleright (x = \alpha))$$

$$\frac{\forall z.\Sigma(z) = z : \forall\alpha.(x = \alpha) \triangleright (x = \alpha) \qquad \exists E.(x = \beta + 1) \implies (x = \alpha)\,[\,\alpha \mapsto E\,]}{\forall\alpha.(x = \alpha) \triangleright (x = \alpha) \vdash \{(x = \beta)\}\, x{+}{+};\, \{(x = \beta + 1)\}}$$

Right branch:

$$\{(x = \beta + 1)\}\, x{-}{-};\, \{(x = \beta)\}$$

$$((x = \beta + 1) \triangleright (x = \beta))\ \textbf{stable}\ (\forall\alpha.(x = \alpha) \triangleright (x = \alpha))$$

$$\frac{\forall z.\Sigma(z) = z : \forall\alpha.(x = \alpha) \triangleright (x = \alpha) \qquad \exists E.(x = \beta) \implies (x = \alpha)\,[\,\alpha \mapsto E\,]}{\forall\alpha.(x = \alpha) \triangleright (x = \alpha) \vdash \{(x = \beta + 1)\}\, x{-}{-};\, \{(x = \beta)\}}$$

5.8 Idioms of Signal Usage - Logic Examples

In this section, we discuss situations that could be addressed with our logic.

5.8.1 Invariant Preserving

The general practice in C programs is to implement error handling by setting the error code using a special variable `errno`, which is global, and returning `-1` [56, 87]. However, the function call that reports an error can be interrupted by the signal handler just before a return. Therefore, there is a possibility that `errno` may be overwritten inside the signal handler. For example, a handler may call a function that results in another error. As `errno` is overwritten, an incorrect error will be reported when interrupted function returns. Therefore, the general advice is to use signal handlers in a safe way, such that they do not interfere with error handling mechanism of the programs. However, this is not always possible; therefore, the more practical advice would be saving and restoring the `errno` value inside the handler if it has access to `errno`.

In our logic, we model this situation as part of the signal handling mechanism, where an invariant that holds before the handler should also hold after it. Assume that $h_1 \,;\, h_2$ are the components of a persistent signal handler. We know that the handler's code is not atomic in a way that it can be interrupted by other handlers except by itself. We also know that it must satisfy an invariant (Definition 6.1.1), let's call it I. However, there is no limitation on invalidating invariant I inside the signal handler. The handler's code may consist of many nested or sequentially composed commands, including command $\texttt{throw}\, e$.

$$\frac{\emptyset \vdash \{I\}\, h_1\, \{P_2\}\, \texttt{throws}\, \emptyset \qquad \emptyset \vdash \{P_2\}\, h_2\, \{I\}\, \texttt{throws}\, \emptyset}{\emptyset \vdash \{I\}\, h_1 \,;\, h_2\, \{I\}\, \texttt{throws}\, \emptyset}$$

An example above, represents a branch of a derivation tree for the sequentially composed commands $h_1 \,;\, h_2$ in the handler. To focus on the idea of invalidating and revalidating of

```
void phandler(int signo) {
   int tmp = errno;
   /* some code that potentially may invalidate errno */
   errno = tmp;
}
```

Figure 5.5: Save and restore `errno`

the invariant I in the handler, exception context is kept empty.

An example in Figure 5.5, shows a signal handler (*phandler*) that saves the value of `errno` on entry and restores it on return. This technique ensures that the correct error reported when interrupted call returns after the handler.

The following example represents how to address the code from Figure 5.5 in our logic. An invariant I for this handler could be an equality of values of the global variable `errno` before and after the handler. Assume that *gvar* (ghost variable) stands for the correct value of `errno`, then I stands for $(gvar == errno)$. In our language, the code of *phandler* will have the next form $t := errno; h; errno := t;$.

$$\frac{\emptyset \vdash \{I\}\, t := errno;\ \{I\}\ \texttt{throws}\ \emptyset \qquad \emptyset \vdash \{I\}\, h\, \{P_2\}\ \texttt{throws}\ \emptyset}{\emptyset \vdash \{I\}\, t := errno; h\, \{P_2\}\ \texttt{throws}\ \emptyset}$$

$$\frac{\emptyset \vdash \{I\}\, t := errno; h\, \{P_2\}\ \texttt{throws}\ \emptyset \quad \emptyset \vdash \{P_2\}\, errno := t;\ \{I\}\ \texttt{throws}\ \emptyset}{\emptyset \vdash \{I\}\, t := errno; h; errno := t;\ \{I\}\ \texttt{throws}\ \emptyset}$$

More abstract version is given below.

Assume that $phandler = h_1; h_2; ...; h_{n-1}; h_n$ and I stands for $(gvar == errno)$.

$$\frac{\emptyset \vdash \{I\}\, h_1; ...; h_{n-1}\, \{P_n\}\ \texttt{throws}\ \emptyset \qquad \emptyset \vdash \{P_n\}\, h_n\, \{I\}\ \texttt{throws}\ \emptyset}{\emptyset \vdash \{I\}\, phandler\, \{I\}\ \texttt{throws}\ \emptyset}$$

Remark 5.8.1 It should be noted that even async-signal safe functions deal with `errno` in non-reentrant way [56]. Thus, one can conclude that async-signal safe functions are not completely safe if error handling mechanism is considered as part of that functions.

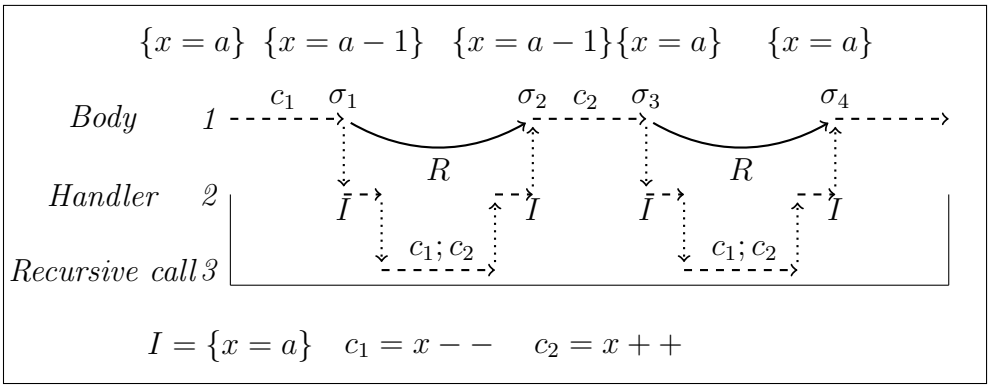

Figure 5.6: Invariant and recursive calls

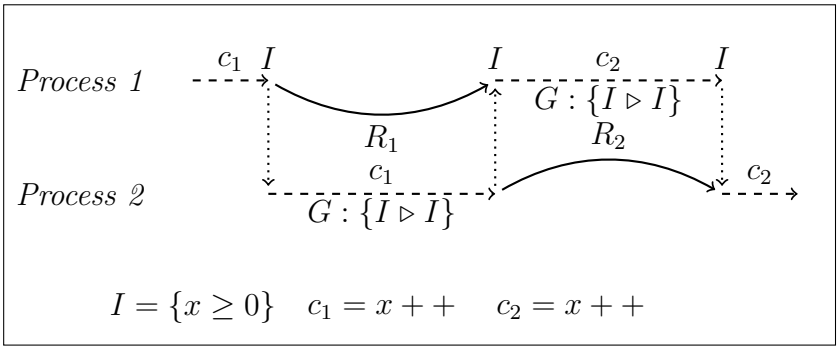

Figure 5.7: Invariant for concurrent processes

Examples of invariants for handlers and concurrent processes are presented in Figure 5.6 and Figure 5.7.

5.8.2 Signal Masks in Unix-like Systems

In Unix-like systems, the set of blocked signals is stored in a signal mask. In some OS (e.g.: Linux) the signal mask is not treated carefully when long jumps are performed [56]. Therefore, signals may remain blocked even in a scope where it is no longer required. We define a special rule for the signal blocking (Figure 5.1 and Figure 5.2), which perfectly fits the language and does not violate the signal bindings in the presence of exceptions.

Signals arrives nondeterministically, thus signal handler may run at any time. The interaction between program and signals is the classic example of shared memory concurrency. The signal handler may corrupt some global variables or resources that program code relies on and uses (in other words, interfere destructively). This may or not result

$$\frac{\Sigma \vdash \{P_1\}\, c_1\, \{P_2\}\ \texttt{throws}\ \eta \quad \Sigma \vdash \{P_2\}\, (c_{2a}; c_{2b}; c_{2c})\, \{P_3\}\ \texttt{throws}\ \eta}{\Sigma \vdash \{P_1\}\, c_1\, ;\, (c_{2a}; c_{2b}; c_{2c})\, \{P_3\}\ \texttt{throws}\ \eta}$$

$$\frac{\Sigma \vdash \{P_1\}\, c_1\, ;\, (c_{2a}; c_{2b}; c_{2c})\, \{P_3\}\ \texttt{throws}\ \eta \qquad \Sigma \vdash \{P_3\}\, c_3\, \{P_4\}\ \texttt{throws}\ \eta}{\Sigma \vdash \{P_1\}\, c_1\, ;\, (c_{2a}; c_{2b}; c_{2c})\, ;\, c_3\, \{P_4\}\ \texttt{throws}\ \eta}$$

Figure 5.8: Three sequentially composed commands

$$\frac{\dfrac{\emptyset \vdash \{P_2\}\, c_{2a}\, \{I_b\} \qquad \emptyset \vdash \{I_b\}\, c_{2b}\, \{I_b\}}{\emptyset \vdash \{P_2\}\, c_{2a}; c_{2b}\, \{I_b\}} \qquad \emptyset \vdash \{I_b\}\, c_{2c}\, \{P_3\}}{\emptyset \vdash \{P_2\}\, (c_{2a}; c_{2b}; c_{2c})\, \{P_3\}}$$

$$\frac{\Sigma \vdash \{P_1\}\, c_1\, \{P_2\} \qquad \dfrac{\emptyset \vdash \{P_2\}\, (c_{2a}; c_{2b}; c_{2c})\, \{P_3\}}{\Sigma \vdash \{P_2\}\, \texttt{block}\, z\, \texttt{in}\, (c_{2a}; c_{2b}; c_{2c})\, \{P_3\}}}{\Sigma \vdash \{P_1\}\, c_1\, ;\, \texttt{block}\, z\, \texttt{in}\, (c_{2a}; c_{2b}; c_{2c})\, \{P_3\}}$$

$$\frac{\Sigma \vdash \{P_1\}\, c_1\, ;\, \texttt{block}\, z\, \texttt{in}\, (c_{2a}; c_{2b}; c_{2c})\, \{P_3\} \qquad \Sigma \vdash \{P_3\}\, c_3\, \{P_4\}}{\Sigma \vdash \{P_1\}\, c_1\, ;\, \texttt{block}\, z\, \texttt{in}\, (c_{2a}; c_{2b}; c_{2c})\, ;\, c_3\, \{P_4\}}$$

where $\Sigma = z : P_z \triangleright Q_z$

Figure 5.9: Three sequentially composed commands and blocking

in a program crash, but the result becomes unreliable. Thus, a potential interaction between the signal handler and a program code that interact through shared resources should be treated carefully. To prevent unwanted interaction (destructive interference) with signal handlers, signals might be temporarily blocked while program operates with data (data structures, variables, resources, etc.) that are sensible to interference.

Assume we have three sequentially composed commands $c_1\, ;\, (c_{2a}; c_{2b}; c_{2c})\, ;\, c_3$ as given in Figure 5.8. What if $(c_{2a}; c_{2b}; c_{2c})$ has a stronger rely in comparison with the other commands c_1 and c_3? For example, $(c_{2a}; c_{2b}; c_{2c})$ has more strict conditions for signals, such that interference with a signal z may lead to an incorrect outcome of the command $(c_{2a}; c_{2b}; c_{2c})$. Thus, we can satisfy stronger rely of $(c_{2a}; c_{2b}; c_{2c})$ by blocking signal z during its execution. Then, the derivation tree has the next form ($\texttt{throws}\eta$ is excluded for clarity) as given in Figure 5.9. Assume that

- c_1, c_2, and c_3 are of the next form $x{+}{+}$;

$$\frac{\dfrac{\emptyset \vdash \{P\}\, c\, \{Q'_k\}\ \texttt{throws}\ e_k : Q'_k \qquad \emptyset \vdash \{Q'_k\}\ \texttt{throw}\ e_k\, \{Q\}\ \texttt{throws}\ e_k : Q'_k}{\emptyset \vdash \{P\}\, (c\,;\texttt{throw}\, e_k)\, \{Q\}\ \texttt{throws}\ e_k : Q'_k}}{\Sigma \vdash \{P\}\, (\texttt{block}\, z\, \texttt{in}\, (c\,;\texttt{throw}\, e_k))\, \{Q\}\ \texttt{throws}\ e_k : Q'_k}$$

$$\frac{\Sigma \vdash \{P\}\, b\, \{Q\}\ \texttt{throws}\ e_k : Q'_k \qquad \Sigma \vdash \{Q'_k\}\, c_k\, \{Q_k\}\ \texttt{throws}\ \emptyset}{\Sigma \vdash \{P\}\ \texttt{try}\ b\ \texttt{handle}\ e_k\ \texttt{by}\ c_k\, \{Q \vee Q_k\}\ \texttt{throws}\ \emptyset}$$

where $\Sigma = z : P_z \triangleright Q_z$

Figure 5.10: Interaction of blocking and exceptions

- signal's z code is of the form $x--; y--;$
- $P_z = \{x = A\}$ and $Q_z = \{x = A - 1\}$
- $P_1 = \{x > 2\}$
- P_2, P_3 and P_4 are of the next form $\{x > 0\}$
- I_b is of the next form $\{y = B\}$

In the next derivation, in Figure 5.10, we show that the blocking rule and exceptions interact in a clear way. When the scope of blocking is left via a raised exception, the signal context is restored; thus, c_k runs with nonempty signal context. Let b stand for $(\texttt{block}\, z\, \texttt{in}\, (c\,;\texttt{throw}\, e_k))$. Placing a single atomic command inside of the scope with blocked signals, does not look too useful, as when control flow leaves the scope with blocked signals, atomic command's postcondition still has to be stable under signals' context. However, if we put at least two sequentially composed commands inside of the scope with blocked signals, then the usefulness of the blocking rule becomes visible. One can use the first command to set all prerequisites before a critical operation and perform it in the second command. The trick is that predicate between these two commands cannot be invalidated by harmful interference with the signal handlers, as they remain blocked in that place. Let's examine $z : P_z \triangleright Q_z \vdash \{P_2\}\, \texttt{block}\, z\, \texttt{in}\, (c_a\,;c_b)\, \{P_3\}\, \texttt{throws}\, \eta$ and $z : P_z \triangleright Q_z \vdash \{P_2\}\, (c_a\,;c_b)\, \{P_3\}\, \texttt{throws}\, \eta$ in detail. Assume that P_z is $\texttt{true}$, Q_z equals $(y = 0)$, P_2 equals $(x = \alpha)$, P_3 equals $(x = \alpha)$, $c_a = (y := 1;)$ and $c_b = (x := x/y;)$. We

$$\frac{\begin{array}{c}\{(x=\alpha)\}\, y := 1;\, \{(x=\alpha)\wedge(y=1)\} \\ ((x=\alpha)\triangleright((x=\alpha)\wedge(y=1)))\ \mathbf{stable}\ (\mathit{true}\triangleright(y=0)) \\ \forall z.\Sigma(z)=z:\mathit{true}\triangleright(y=0) \qquad \exists E.((x=\alpha)\wedge(y=1))\implies \mathit{true}\end{array}}{\begin{array}{c}z:\mathit{true}\triangleright(y=0)\vdash\{(x=\alpha)\}\, y := 1;\, \{(x=\alpha)\wedge(y=1)\} \\ z:\mathit{true}\triangleright(y=0)\vdash\{(x=\alpha)\wedge(y=1)\}\, x := x/y;\, \{(x=\alpha)\}\end{array}}$$
$$z:\mathit{true}\triangleright(y=0)\vdash\forall\alpha.\{(x=\alpha)\}\,(y := 1; x := x/y;)\,\{(x=\alpha)\}$$

Figure 5.11: Sequential composition without block construct

$$\frac{\begin{array}{c}\emptyset\vdash\{x=\alpha\}\, y := 1;\, \{(x=\alpha)\wedge(y=1)\} \quad \emptyset\vdash\{(x=\alpha)\wedge(y=1)\}\, x := x/y;\, \{x=\alpha\} \\ \emptyset\vdash\{(x=\alpha)\}\,(y := 1; x := x/y;)\,\{(x=\alpha)\} \\ (x=\alpha)\ \mathbf{stable}\ \mathit{true}\triangleright(y=0) \qquad (x=\alpha)\ \mathbf{stable}\ \mathit{true}\triangleright(y=0)\end{array}}{z:\mathit{true}\triangleright(y=0)\vdash\forall\alpha.\{(x=\alpha)\}\,\mathtt{block}\, z\,\mathtt{in}\,(y := 1; x := x/y;)\,\{(x=\alpha)\}}$$

Figure 5.12: Sequential composition with block construct

omit $\mathtt{throws}\eta$ in the following examples.

Sequential composition with and without block structure

To show that blocking is a powerful construct, let's consider an example where the signal blocking has been excluded in Figure 5.11. We observe that without the blocking construct our program is no longer safe; that is indicated by stability assumption $((x=\alpha)\triangleright((x=\alpha)\wedge(y=1)))$ **stable** $(\mathit{true}\triangleright(y=0))$, which is obviously false. On the contrary, the program in Figure 5.12 is safe.

Blocking and exceptions

In the next derivation, in Figure 5.13, we show that the blocking rule and exceptions interact in a clear way. Let b stand for $(\mathtt{block}\, z\,\mathtt{in}\,(c\,;\mathtt{throw}\, e_k))$. When the scope of blocking is left via a raised exception, the signal context is restored; thus, c_k runs with nonempty signal context.

$$\frac{\dfrac{\emptyset \vdash \{P\}\, c\, \{Q'_k\}\ \texttt{throws}\ e_k : Q'_k \qquad \emptyset \vdash \{Q'_k\}\ \texttt{throw}\, e_k\, \{Q\}\ \texttt{throws}\ e_k : Q'_k}{\emptyset \vdash \{P\}\, (c\,;\texttt{throw}\, e_k)\, \{Q\}\ \texttt{throws}\ e_k : Q'_k}}{z : P_z \triangleright Q_z \vdash \{P\}\, (\texttt{block}\, z\, \texttt{in}\, (c\,;\texttt{throw}\, e_k))\, \{Q\}\ \texttt{throws}\ e_k : Q'_k}$$

$$\frac{z : P_z \triangleright Q_z \vdash \{P\}\, b\, \{Q\}\ \texttt{throws}\ e_k : Q'_k \qquad z : P_z \triangleright Q_z \vdash \{Q'_k\}\, c_k\, \{Q_k\}\ \texttt{throws}\ \emptyset}{z : P_z \triangleright Q_z \vdash \{P\}\ \texttt{try}\ b\ \texttt{handle}\ e_k\ \texttt{by}\ c_k\, \{Q \vee Q_k\}\ \texttt{throws}\ \emptyset}$$

Figure 5.13: Blocking construct and exceptions

5.8.3 Double Free and Linear Use of Resources

Assume that the signal handler has the next form:

```
/* signal handler*/
void sighandler(int sigid) {
// ...
free(pointer);
// ...
}
```

And the next code appears in the main program:

```
pointer = (int *)malloc(256);


/* signal handler binding*/
signal(SIGINT,sighandler);
```

Such scenario contains a security flaw, because if a signal arrives twice, it will result in a double free error. A memory free command shouldn't run more than once, and the memory pointer is a resource that should be used linearly. In our language we have one-shot signals, which might be used to ensure linear use of resources. Alternatively, a combination of persistent signals and exception throwing mechanism could be used to achieve linear use of resources.

Separation Logic

To describe this situation in detail (closer to the real-life implementations), one need to combine a separation logic with our logic. Embedding with the separation logic is out of scope of this thesis, and is left for the future work. What is important, is that we generalise a memory allocation and deallocation as a resource that could be used in a linear way only. Suppose we have a handler with precondition P and postcondition Q:

$$\{P\}\, c\, \{Q\}$$

Moreover, suppose Q does not imply P. For instance, it could be that some resource, say a pointer, is available if P is satisfied, but not when Q is satisfied. In separation logic, the triple could be

$$\{p \mapsto -\}\, \texttt{free}(p)\, \{\texttt{empty}\}$$

We can still use c as a one-shot handler, provided the body of the signal block does not rely on P. As the handler is one-shot, it can safely invalidate its own precondition. However, we could not prove anything about the handler if it is used as a persistent handler. A persistent handler must satisfy an invariant. One the other hand, a persistent handler combined with an exception may be used. As when control flow leaves the scope of bound signal via exception, the signal becomes automatically uninstalled; thus, the handler will not run again. Let's consider an example, in Figure 5.14 and Figure 5.15, where b stands for $\texttt{bind}\, z\, \texttt{to}\, (c_h\, ;\, \texttt{throw}\, e)\, \texttt{in}\, c$. Please note that the postcondition of c_h is not the invariant I_z. This is due to the possibility of the invariant invalidation in c_h. Moreover, the invariant I_z need not to be respected as the exception is caught in the outer scope.

$$\frac{\emptyset \vdash \{I_z\}\, c_h\, \{Q'_e\}\ \texttt{throws}\ e : Q'_e \qquad \emptyset \vdash \{Q'_e\}\ \texttt{throw}\, e\, \{I_z\}\ \texttt{throws}\ e : Q'_e}{\emptyset \vdash \forall\alpha.\{I_z\}\, (c_h\, ;\, \texttt{throw}\, e)\, \{I_z\}\ \texttt{throws}\ e : Q'_e}$$

$$\frac{\begin{array}{c} z : \forall\alpha.I_z \triangleright I_z \vdash \{P\}\, c\, \{Q\}\ \texttt{throws}\ e : Q'_e \\ \emptyset \vdash \forall\alpha.\{I_z\}\, (c_h\, ;\, \texttt{throw}\, e)\, \{I_z\}\ \texttt{throws}\ e : Q'_e \end{array}}{\emptyset \vdash \forall\alpha.\{P \wedge I_z\}\ \texttt{bind}\, z\, \texttt{to}\, (c_h\, ;\, \texttt{throw}\, e)\, \texttt{in}\, c\, \{Q \wedge I_z\}\ \texttt{throws}\ e : Q'_e}$$

$$\frac{\emptyset \vdash \forall\alpha.\{P \wedge I_z\}\, b\, \{Q \wedge I_z\}\ \texttt{throws}\ e : Q'_e \qquad \emptyset \vdash \{Q'_e\}\, c_e\, \{Q_e\}\ \texttt{throws}\ \emptyset}{\emptyset \vdash \forall\alpha.\{P \wedge I_z\}\ \texttt{try}\ b\ \texttt{handle}\ e\ \texttt{by}\ c_e\, \{(Q \wedge I_z) \vee Q_e\}\ \texttt{throws}\ \emptyset}$$

Figure 5.14: Binding is nested in exception handling

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{S_1(z) = (h\, ;\, \texttt{throw}\, e) \qquad \dfrac{S{-}z; O \Vdash s_1, h \Downarrow s_2 \qquad \dfrac{}{S{-}z; O \Vdash s_2, \texttt{throw}\, e \Uparrow e, s_2}}{S{-}z; O \Vdash s_1, (h\, ;\, \texttt{throw}\, e) \Uparrow e, s_2}}{S_1; O \Vdash s_1, c \Uparrow e, s_2}}{S; O \Vdash s_1, (\texttt{bind}\, z\, \texttt{to}\, (h\, ;\, \texttt{throw}\, e)\, \texttt{in}\, c_B) \Uparrow e, s_2} \qquad S; O \Vdash s_2, g \Downarrow s_3}{S; O \Vdash s_1, \texttt{try}\ (\texttt{bind}\, z\, \texttt{to}\, (h\, ;\, \texttt{throw}\, e)\, \texttt{in}\, c_B)\ \texttt{handle}\ e\ \texttt{by}\ g \Downarrow s_3}$$

where $S_1 = S\,[\, z \mapsto (h\, ;\, \texttt{throw}\, e)\,]$

Figure 5.15: Binding is nested in exception handling

CHAPTER 6

LOGIC SOUNDNESS

In this chapter, we show a soundness proof of our logic with respect to the big-step operational semantics defined in Chapter 3.

Perhaps the most interesting feature of the operational semantics is the multiplicative way that one-shot signals are propagated, as the signal binding is split into two disjoint bindings $O_1 * O_2$ when a semantics rule has two premises, for example in a sequential composition. In the logic, by contrast, the signal context Σ is used additively, in that it is shared rather than split in the logic rules. Intuitively, that is due to the fact that signals arrive non-deterministically, so that we cannot determine statically which way the splitting will go at runtime, and Σ only specifies what signals *may* be handled.

In the big-step semantics, the evaluation of a command c is relative to some signal binding $S; O$ for persistent and one-shot signal handlers. These handlers can run at any time during the evaluation of c. So to reason about the behaviour of c relative to some Hoare logic specification $\{P\}\, c\, \{Q\}$, we need to impose a specification on the signal contexts as well. If they could interfere in the evaluation of c with arbitrary behaviour, there would be no way of proving the correctness of c. Moreover, the signal handlers must also respect the specification of other handlers in the sense that they should not invalidate the assumptions of handlers that may run after or before them, respectively. For one-shot signals, the assumptions are a precondition P and postcondition Q, while for persistent handlers, they are invariants I.

6.1 Signal Binding and Signal Context

We need to impose the condition that all signal bindings respect the specification given by the whole signal context, in the following sense:

Definition 6.1.1 (Signal bindings respect signal contexts) Let S and O be signal bindings, and let Σ be a signal context. We say that $S; O \models \Sigma$ if the following conditions hold:

1. If the one-shot binding $c = O(z)$ is defined for some signal name z, then $\Sigma(z) = (P_z \triangleright Q_z)$ is also defined for some precondition P_z and postcondition Q_z. Moreover, c behaves as specified by P_z and Q_z, in the following sense: if there are states s_1 and s_2 such that

   $$\emptyset; \emptyset \Vdash s_1, c \Downarrow s_2$$

   and $s_1 \models P_z$, then $s_2 \models Q_z$.

2. If the persistent signal $c = S(z)$ is defined for some signal name z, then $\Sigma(z) = I_z \triangleright I_z$ is also defined for some invariant I_z. Moreover, c preserves the invariant in the following sense: if there are states s_1 and s_2 such that

   $$\emptyset; \emptyset \Vdash s_1, c \Downarrow s_2$$

   and $s_1 \models I_z$, then $s_2 \models I_z$.

We also need a form of stability condition between signal and exception contexts.

Definition 6.1.2 For an exception context η such that $\eta = e_1 : Q'_1, \ldots, e_n : Q'_n$ and a signal context Σ, we write η **stable** Σ if and only if for all $e_j \in \mathrm{dom}(\eta)$ such that $\eta(e_j) = Q'_j$ we have Q'_j **stable** Σ.

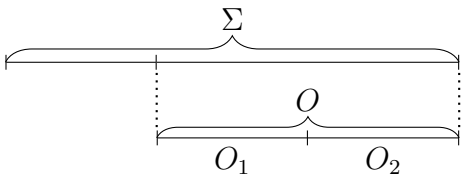

Figure 6.1: Splitting of the one-shot signal binding

6.2 Supporting Lemmas

Lemma 6.2.1 (Splitting one-shot bindings) If $S; O \models \Sigma$ and O can be split as $O = O_1 * O_2$, then $S; O_1 \models \Sigma$ and $S; O_2 \models \Sigma$.

One may visualize the splitting as in Figure 6.1.

Proof

If $S; O \models \Sigma$, then according to the Definition 6.1.1, for every signal z defined in the signal bindings there is a corresponding definition in the signal contexts, and for every signal $z \in \mathrm{dom}(O)$ all respect conditions are satisfied.

We know that $O = O_1 * O_2$. According to the Definition 3.1.2 for " $*$ ", we know that $O_1 \cap O_2 = \emptyset$, and that both O_1 and O_2 are subsets of O. Thus, for every signal z such that originally $z \in \mathrm{dom}(O)$, after the splitting, the signal z will be either $z \in \mathrm{dom}(O_1)$ or $z \in \mathrm{dom}(O_2)$.

$S; O_1 \models \Sigma$ and $S; O_2 \models \Sigma$ if and only if all respect conditions are satisfied for every signal from both domains O_1 and O_2. Assume that $z \in \mathrm{dom}(O_1)$, and as $O_1 \subseteq O$ then $z \in \mathrm{dom}(O)$. We know that $S; O \models \Sigma$, therefore all respect conditions for the signal $z \in \mathrm{dom}(O)$ are met. Thus, for every signal $z \in \mathrm{dom}(O_1)$ all conditions are met because for every signal $z \in \mathrm{dom}(O_1)$ implies $z \in \mathrm{dom}(O)$. Analogously, for every $z \in \mathrm{dom}(O_2)$ implies $z \in \mathrm{dom}(O)$, therefore all respect conditions for the signal $z \in \mathrm{dom}(O_2)$ are met.

Furthermore, a persistent signal bindings S is copied to the both split parts, and we know that for all signals $z \in \mathrm{dom}(S)$ respect conditions are satisfied.

Therefore, $S; O_1 \models \Sigma$ and $S; O_2 \models \Sigma$. □

Lemma 6.2.2 (Reducing signals' bindings and context) Assume that
$S; O \models \Sigma, z : P_z \triangleright Q_z$ and $\Sigma, z : P_z \triangleright Q_z$ **pairstable** then
Σ **pairstable** and $S - z; O - z \models \Sigma$.

Proof

Assume $S; O \models \Sigma, z : P_z \triangleright Q_z$ and $\Sigma, z : P_z \triangleright Q_z$ **pairstable** hold. Thus, for any z_1, z_2 such that $z_1 \in \mathrm{dom}(O)$ or $z_1 \in \mathrm{dom}(S)$, $z_2 \in \mathrm{dom}(O)$ or $z_2 \in \mathrm{dom}(S)$ we have $\Sigma(z_1) = P_1 \triangleright Q_1$, $\Sigma(z_2) = P_2 \triangleright Q_2$, and P_1 **stable** $(P_2 \triangleright Q_2)$, Q_1 **stable** $(P_2 \triangleright Q_2)$, P_2 **stable** $(P_1 \triangleright Q_1)$, Q_2 **stable** $(P_1 \triangleright Q_1)$. Please note that for $z \in \mathrm{dom}(S)$ we have $\Sigma(z) = P_z \triangleright Q_z$ where $P_z = Q_z$.

Then, we simultaneously remove signal z from the signal context and bindings. Thus, $S - z; O - z \models \Sigma$, as for any z_1, z_2 such that $z_1 \neq z$, $z_2 \neq z$, $z_1 \in \mathrm{dom}(O - z)$ or $z_1 \in \mathrm{dom}(S - z)$, $z_2 \in \mathrm{dom}(O - z)$ or $z_2 \in \mathrm{dom}(S - z)$ all stability assumptions are satisfied. Furthermore, removing of $z : P_z \triangleright Q_z$ from the signal context is safe, as for any $z_1 \in \mathrm{dom}(O - z)$ or $z_1 \in \mathrm{dom}(S - z)$, $z_1 \neq z$. Thus, consistency of the signal context remains satisfied. Therefore, we conclude $S - z; O - z \models \Sigma$ and Σ **pairstable**. □

Lemma 6.2.3 (Extending one-shot signals' binding and context) If
Σ **pairstable** and $S; O \models \Sigma$. Also suppose that $[\, z \mapsto c_h \,]$ and $z : P_z \triangleright Q_z$. Furthermore, assume that P_z **stable** Σ, Q_z **stable** Σ and Σ **stable** $(P_z \triangleright Q_z)$ hold. Then $S; O\,[\, z \mapsto c_h \,] \models \Sigma, z : P_z \triangleright Q_z$ and $\Sigma, z : P_z \triangleright Q_z$ **pairstable**.

Proof

Assume that Σ **pairstable** and $S; O \models \Sigma$ hold. To extend a signal context Σ and a binding O with the new signal z, one should ensure that the respect and stability conditions remain satisfied. Thus, pre- and post conditions of the signal z should be stable under other signals, which already are elements of the signal context and binding. Moreover, pre- and postconditions of all that one-shot signals and the invariants of all that persistent signals should be preserved under the actions of z.

As all stability assumptions are assumed we can safely extend the signal binding and the context. Thus, $S; O\,[\,z \mapsto c_h\,] \models \Sigma, z : P_z \triangleright Q_z$ and $\Sigma, z : P_z \triangleright Q_z$ **pairstable**. $\square$

Lemma 6.2.4 (Extending persistent signals' binding and context) If Σ **pairstable** and $S; O \models \Sigma$. Also suppose that $[\,z \mapsto c_h\,]$ and $z : I_z \triangleright I_z$. Furthermore, assume that I_z **stable** Σ and Σ **stable** $I_z \triangleright I_z$.

Then $S\,[\,z \mapsto c_h\,]; O \models \Sigma, z : I_z \triangleright I_z$ and $\Sigma, z : I_z \triangleright I_z$ **pairstable**.

Proof

Assume that Σ **pairstable** and $S; O \models \Sigma$ holds. To extend a signal context Σ and a binding S with the new signal z, one should ensure that respect and stability conditions remain satisfied. Thus, an invariant of the signal z should be stable under other signals, which already are elements of signal context and binding. Moreover, pre- and postconditions of all that one-shot signals and the invariants of all that persistent signals should be preserved under the actions of z.

As all stability assumptions are assumed we can safely extend the signal binding and the context. Thus, $S\,[\,z \mapsto c_h\,]; O \models \Sigma, z : I_z \triangleright I_z$ and $\Sigma, z : I_z \triangleright I_z$ **pairstable**. $\square$

6.3 Proof of Soundness

For the signal handling rule, we will need supporting Lemma 5.5.1.

Definition 6.3.1 (Validity with signals and exceptions) A judgement

$$\Sigma \vdash \{P\}\, c\, \{Q\}\ \texttt{throws}\ e_1 : Q'_1, \ldots, e_n : Q'_n$$

is called *valid* if the following holds. Suppose Σ **pairstable** holds, and that S and O are signal bindings such that $S; O \models \Sigma$. Let s_1 and s_2 be states. Then all evaluations (which can be either a normal termination or a raised exception) satisfy:

1. If $S; O \Vdash s_1, c \Downarrow s_2$ then $s_1 \models P$ implies $s_2 \models Q$

2. If $S; O \Vdash s_1, c \Uparrow e_j, s_2$ then $s_1 \models P$ implies $s_2 \models Q'_j$

Moreover, in both of these cases, the states s_1 and s_2 in the evaluation satisfy the following conditions regarding Σ:

- for any signal name z with $\Sigma(z) = (P_z \triangleright Q_z)$ and $z \in \mathrm{dom}(O)$, $s_1 \models P_z$ implies $s_2 \models P_z \vee Q_z$
- for any signal name z with $\Sigma(z) = (P_z \triangleright Q_z)$ and $z \notin \mathrm{dom}(O)$, $s_1 \models P_z$ implies $s_2 \models P_z$ and $s_1 \models Q_z$ implies $s_2 \models Q_z$

Theorem 6.3.2 (Soundness) Each Hoare triple that can be derived using the program logic rules (Figure 5.1 and Figure 5.2) is valid in the sense of Definition 6.3.1.

Proof The proof proceeds by induction over the derivation of a program logic judgement

$$\Sigma \vdash \{P\}\, c\, \{Q\}\ \texttt{throws}\ \eta$$

We consider all the cases how the proof tree could be built up, and reason about the possible big-step evaluations (Figure 3.2, Figure 3.3, Figure 3.4).

$$S; O \Vdash s_1, c \Downarrow s_2 \quad \text{and} \quad S; O \Vdash s_1, c \Uparrow e, s_2$$

□

6.3.1 Persistent Signal Binding

Lemma 6.3.3 (Soundness of the persistent signal binding) We consider the program rule for the persistent signal binding:

$$\frac{\Sigma, z : \forall\alpha. I_z \triangleright I_z \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta \qquad \Sigma \vdash \forall\alpha. \{I_z\}\, c_h\, \{I_z\}\ \texttt{throws}\ \eta \qquad \Sigma\ \textbf{stable}\ \forall\alpha. I_z \triangleright I_z}{\Sigma \vdash \forall\alpha. \{P \wedge I_z\}\ \texttt{bind}\ z\ \texttt{to}\ c_h\ \texttt{in}\ c_B\, \{Q \wedge I_z\}\ \texttt{throws}\ \eta}$$

where $\texttt{throws}\ \eta = \texttt{throws}\ e_1 : Q'_1, \ldots, e_n : Q'_n$

Suppose the judgments $\Sigma, z : \forall\alpha.I_z \triangleright I_z \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta$ and $\Sigma \vdash \forall\alpha.\{I_z\}\, c_h\, \{I_z\}\ \texttt{throws}\ \eta$ are derivable. Also suppose that Σ **stable** $\forall\alpha.I_z \triangleright I_z$ holds. We need to show that the judgment $\Sigma \vdash \forall\alpha.\{P \wedge I_z\}\ \texttt{bind}\ z\ \texttt{to}\ c_h\ \texttt{in}\ c_B\ \{Q \wedge I_z\}\ \texttt{throws}\ \eta$ is valid (Definition 6.3.1).

Proof An evaluation of $\texttt{bind}\ z\ \texttt{to}\ c_h\ \texttt{in}\ c_B$ can be of the following two forms:

$$\frac{S\,[\,z \mapsto c_h\,]; O \Vdash s_1, c_B \Downarrow s_2}{S; O \Vdash s_1, \texttt{bind}\ z\ \texttt{to}\ c_h\ \texttt{in}\ c_B \Downarrow s_2}$$

$$\frac{S\,[\,z \mapsto c_h\,]; O \Vdash s_1, c_B \Uparrow e_1, s_2}{S; O \Vdash s_1, \texttt{bind}\ z\ \texttt{to}\ c_h\ \texttt{in}\ c_B \Uparrow e_1, s_2}$$

Case 1

$$\frac{S\,[\,z \mapsto c_h\,]; O \Vdash s_1, c_B \Downarrow s_2}{S; O \Vdash s_1, \texttt{bind}\ z\ \texttt{to}\ c_h\ \texttt{in}\ c_B \Downarrow s_2}$$

Let Σ **pairstable**, and $S; O \models \Sigma$. Hence we can apply the induction hypothesis to the smaller derivation trees for c_h and c_B.

First, let $s_1 \models I_z$. Then, by the induction hypothesis (for c_h), $s_2 \models I_z$. It remains to check the conditions on signals. We know that z is a signal such that $z \notin \mathrm{dom}(\Sigma)$. Let z' be a signal name such that $z' \neq z$ and $\Sigma(z') = (P_{z'} \triangleright Q_{z'})$. We make a case distinction, based on whether $z' \in \mathrm{dom}(O)$ or not.

Case $z' \in \mathrm{dom}(O)$ Suppose that $s_1 \models P_{z'}$. We need to prove that $s_2 \models P_{z'} \vee Q_{z'}$.

By the induction hypothesis (for c_h), $s_2 \models P_{z'} \vee Q_{z'}$.

Case $z' \notin \mathrm{dom}(O)$ There are two subcases, where $s_1 \models P_{z'}$ or $s_1 \models Q_{z'}$

Case $s_1 \models P_{z'}$ By the induction hypothesis (for c_h), $s_2 \models P_{z'}$.

Case $s_1 \models Q_{z'}$ By the induction hypothesis (for c_h), $s_2 \models Q_{z'}$.

We know that $S; O \models \Sigma$. Thus, the signal handler c_h in $\forall\alpha.\{I_z\}\, c_h\, \{I_z\}$ could be interrupted by any signal from the $\mathrm{dom}(\Sigma)$ during its execution, except by itself

$(z \notin \mathrm{dom}(\Sigma))$. That yields to $[\, z \mapsto c_h \,]$ and $z : \forall\alpha.I_z \triangleright I_z$.

It was assumed that $\Sigma \vdash \forall\alpha.\{I_z\}\, c_h\, \{I_z\}$ is derivable; thus, using Lemma 5.5.1 we infer I_z **stable** Σ. Let the following Hoare logic rule's stability assumption for $(z : \forall\alpha.I_z \triangleright I_z)$ holds: Σ **stable** $\forall\alpha.I_z \triangleright I_z$. Then, using Lemma 6.2.4, we infer $S\,[\, z \mapsto c_h \,]; O \models \Sigma, z : \forall\alpha.I_z \triangleright I_z$ and $\Sigma, z : \forall\alpha.I_z \triangleright I_z$ **pairstable**.

Let $s_1 \models P$. Then, by the induction hypothesis (for c_B), $s_2 \models Q$. That gives us the condition on the Hoare triple required for validity. It remains to check the conditions on signals.

We know that z is a signal name with $\Sigma, z : \forall\alpha.I_z \triangleright I_z(z) = \forall\alpha.(I_z \triangleright I_z)$ such that $z \notin \mathrm{dom}(O)$. Let z' be a signal name with $\Sigma, z : \forall\alpha.I_z \triangleright I_z(z') = (P_{z'} \triangleright Q_{z'})$ such that $z' \in \mathrm{dom}(O)$.

Case $z \notin \mathrm{dom}(O)$ We assumed that $s_1 \models I_z$. We need to prove that $s_2 \models I_z$. By the induction hypothesis (for c_B), $s_2 \models I_z$.

Case $z' \in \mathrm{dom}(O)$ Suppose that $s_1 \models P_{z'}$. By the induction hypothesis (for c_B), $s_2 \models P_{z'} \vee Q_{z'}$.

Case 2

$$\frac{S\,[\, z \mapsto c_h \,]; O \Vdash s_1, c_B \Uparrow e_1, s_2}{S; O \Vdash s_1, \mathtt{bind}\, z\, \mathtt{to}\, c_h\, \mathtt{in}\, c_B \Uparrow e_1, s_2}$$

Let Σ **pairstable**, and $S; O \models \Sigma$. Hence we can apply the induction hypothesis to the smaller derivation trees for c_h and c_B.

First, let $s_1 \models I_z$. Then, by the induction hypothesis (for c_h), $s_2 \models I_z$. It remains to check the conditions on signals. We know that z is a signal such that $z \notin \mathrm{dom}(\Sigma)$. Let z' be a signal name such that $z' \neq z$ and $\Sigma(z') = (P_{z'} \triangleright Q_{z'})$. We make a case distinction, based on whether $z' \in \mathrm{dom}(O)$ or not.

Case $z' \in \mathrm{dom}(O)$ Suppose that $s_1 \models P_{z'}$. We need to prove that $s_2 \models P_{z'} \vee Q_{z'}$. By the induction hypothesis (for c_h), $s_2 \models P_{z'} \vee Q_{z'}$.

Case $z' \notin \mathrm{dom}(O)$ There are two subcases, where $s_1 \models P_{z'}$ or $s_1 \models Q_{z'}$

Case $s_1 \models P_{z'}$ By the induction hypothesis (for c_h), $s_2 \models P_{z'}$.

Case $s_1 \models Q_{z'}$ By the induction hypothesis (for c_h), $s_2 \models Q_{z'}$.

We know that $S; O \models \Sigma$. Thus, the signal handler c_h in $\forall\alpha.\{I_z\}\, c_h\, \{I_z\}$ could be interrupted by any signal from the $\mathrm{dom}(\Sigma)$ during its execution, except by itself ($z \notin \mathrm{dom}(\Sigma)$). That yields to $[\, z \mapsto c_h \,]$ and $z : \forall\alpha.I_z \triangleright I_z$.

It was assumed that $\Sigma \vdash \forall\alpha.\{I_z\}\, c_h\, \{I_z\}$ is derivable; thus, using Lemma 5.5.1 we infer I_z **stable** Σ. Let the following Hoare logic rule's stability assumption for $(z : \forall\alpha.I_z \triangleright I_z)$ holds: Σ **stable** $\forall\alpha.I_z \triangleright I_z$. Then, using Lemma 6.2.4, we infer $S\,[\, z \mapsto c_h \,]; O \models \Sigma, z : \forall\alpha.I_z \triangleright I_z$ and $\Sigma, z : \forall\alpha.I_z \triangleright I_z$ **pairstable**.

Let $s_1 \models P$. Then, by the induction hypothesis (for c_B), $s_2 \models Q_1'$. That gives us the condition on the Hoare triple required for validity. It remains to check the conditions on signals.

We know that z is a signal name with $\Sigma, z : \forall\alpha.I_z \triangleright I_z(z) = \forall\alpha.(I_z \triangleright I_z)$ such that $z \notin \mathrm{dom}(O)$. Let z' be a signal name with $\Sigma, z : \forall\alpha.I_z \triangleright I_z(z') = (P_{z'} \triangleright Q_{z'})$ such that $z' \in \mathrm{dom}(O)$.

Case $z \notin \mathrm{dom}(O)$ We assumed that $s_1 \models I_z$. We need to prove that $s_2 \models I_z$. By the induction hypothesis (for c_B), $s_2 \models I_z$.

Case $z' \in \mathrm{dom}(O)$ Suppose that $s_1 \models P_{z'}$. By the induction hypothesis (for c_B), $s_2 \models P_{z'} \vee Q_{z'}$.

Finally The judgment $\Sigma \vdash \forall\alpha.\{P \wedge I_z\}\, \texttt{bind}\, z\, \texttt{to}\, c_h\, \texttt{in}\, c_B\, \{Q \wedge I_z\}\, \texttt{throws}\, \eta$ is valid.

□

6.3.2 One-shot Signal Binding

Lemma 6.3.4 (Soundness of the one-shot sign bind. and exns) We consider the program rule for the one-shot signal binding:

$$\frac{\Sigma, z : \forall\alpha.P_z \triangleright Q_z \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta \qquad \Sigma \vdash \forall\alpha.\{P_z\}\, c_h\, \{Q_z\}\ \texttt{throws}\ \eta \qquad \Sigma\ \textbf{stable}\ \forall\alpha.P_z \triangleright Q_z}{\Sigma \vdash \forall\alpha.\{P \wedge P_z\}\, \texttt{bind/1}\, z\, \texttt{to}\, c_h\, \texttt{in}\, c_B\, \{Q \wedge (P_z \vee Q_z)\}\ \texttt{throws}\ \eta}$$

where $\texttt{throws}\ \eta = \texttt{throws}\ e_1 : Q'_1, \ldots, e_n : Q'_n$

Suppose the judgments $\Sigma, z : \forall\alpha.P_z \triangleright Q_z \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta$ and $\Sigma \vdash \forall\alpha.\{P_z\}\, c_h\, \{Q_z\}\ \texttt{throws}\ \eta$ are derivable. Also suppose that $\Sigma\ \textbf{stable}\ \forall\alpha.P_z \triangleright Q_z$ holds. We need to show that the judgment $\Sigma \vdash \forall\alpha.\{P \wedge P_z\}\, \texttt{bind/1}\, z\, \texttt{to}\, c_h\, \texttt{in}\, c_B\, \{Q \wedge (P_z \vee Q_z)\}\ \texttt{throws}\ \eta$ is valid (Definition 6.3.1).

Proof An evaluation of $\texttt{bind/1}\, z\, \texttt{to}\, c_h\, \texttt{in}\, c_B$ can be of the following two forms:

$$\frac{S; O\,[\, z \mapsto c_h\,] \Vdash s_1, c_B \Downarrow s_2}{S; O \Vdash s_1, \texttt{bind/1}\, z\, \texttt{to}\, c_h\, \texttt{in}\, c_B \Downarrow s_2}$$

$$\frac{S; O\,[\, z \mapsto c_h\,] \Vdash s_1, c_B \Uparrow e_1, s_2}{S; O \Vdash s_1, \texttt{bind/1}\, z\, \texttt{to}\, c_h\, \texttt{in}\, c_B \Uparrow e_1, s_2}$$

Case 1

$$\frac{S; O\,[\, z \mapsto c_h\,] \Vdash s_1, c_B \Downarrow s_2}{S; O \Vdash s_1, \texttt{bind/1}\, z\, \texttt{to}\, c_h\, \texttt{in}\, c_B \Downarrow s_2}$$

Let Σ **pairstable**, and $S; O \models \Sigma$. Hence we can apply the induction hypothesis to the smaller derivation trees for c_h and c_B.

First, let $s_1 \models P_z$. Then, by the induction hypothesis (for c_h), $s_2 \models Q_z$. It remains to check the conditions on signals. We know that z is a signal such that $z \notin \mathrm{dom}(\Sigma)$. Let z' be a signal name such that $z' \neq z$ and $\Sigma(z') = (P_{z'} \triangleright Q_{z'})$. We make a case

distinction, based on whether $z' \in \mathrm{dom}(O)$ or not.

Case $z' \in \mathrm{dom}(O)$ Suppose that $s_1 \models P_{z'}$. We need to prove that $s_2 \models P_{z'} \vee Q_{z'}$. By the induction hypothesis (for c_h), $s_2 \models P_{z'} \vee Q_{z'}$.

Case $z' \notin \mathrm{dom}(O)$ There are two subcases, where $s_1 \models P_{z'}$ or $s_1 \models Q_{z'}$

Case $s_1 \models P_{z'}$ By the induction hypothesis (for c_h), $s_2 \models P_{z'}$.

Case $s_1 \models Q_{z'}$ By the induction hypothesis (for c_h), $s_2 \models Q_{z'}$.

We know that $S; O \models \Sigma$. Thus, the signal handler c_h in $\{P_z\}\, c_h\, \{Q_z\}$ could be interrupted by any signal from the $\mathrm{dom}(\Sigma)$ during its execution, except by itself. That yields to $[\, z \mapsto c_h \,]$ and $z : P_z \triangleright Q_z$.

It was assumed that $\Sigma \vdash \forall\alpha.\{P_z\}\, c_h\, \{Q_z\}$ is derivable; thus, using Lemma 5.5.1 we infer P_z **stable** Σ and Q_z **stable** Σ. The Hoare logic rule assumes the following stability assumption for $(z : \forall\alpha.P_z \triangleright Q_z)$: Σ **stable** $\forall\alpha.P_z \triangleright Q_z$. Then, using Lemma 6.2.3, we infer $S; O\,[\, z \mapsto c_h \,] \models \Sigma, z : \forall\alpha.P_z \triangleright Q_z$ and $\Sigma, z : \forall\alpha.P_z \triangleright Q_z$ **pairstable**.

Let $s_1 \models P$. Then, by the induction hypothesis (for c_B), $s_2 \models Q$. That gives us the condition on the Hoare triple required for validity. It remains to check the conditions on signals.

We know that z is a signal name with $\Sigma, z : \forall\alpha.P_z \triangleright Q_z(z) = \forall\alpha.(P_z \triangleright Q_z)$ such that $z \in \mathrm{dom}(O)$. Let z' be a signal name with $\Sigma, z : \forall\alpha.P_z \triangleright Q_z(z') = (P_{z'} \triangleright Q_{z'})$ such that $z' \notin \mathrm{dom}(O)$.

Case $z \in \mathrm{dom}(O)$ We assumed $s_1 \models P_z$. We need to prove that $s_2 \models P_z \vee Q_z$. By the induction hypothesis (for c_B), $s_2 \models P_z \vee Q_z$.

Case $z' \notin \mathrm{dom}(O)$ There are two subcases, where $s_1 \models P_{z'}$ or $s_1 \models Q_{z'}$.

Case $s_1 \models P_{z'}$ By the induction hypothesis (for c_B), $s_2 \models P_{z'}$.

Case $s_1 \models Q_{z'}$ By the induction hypothesis (for c_B), $s_2 \models Q_{z'}$.

Case 2

$$\frac{S; O\,[\,z \mapsto c_h\,] \Vdash s_1, c_B \Uparrow e_1, s_2}{S; O \Vdash s_1, \texttt{bind/1}\; z\; \texttt{to}\; c_h\; \texttt{in}\; c_B \Uparrow e_1, s_2}$$

Let Σ **pairstable**, and $S; O \models \Sigma$. Hence we can apply the induction hypothesis to the smaller derivation trees for c_h and c_B.

First, let $s_1 \models P_z$. Then, by the induction hypothesis (for c_h), $s_2 \models Q_z$. It remains to check the conditions on signals. We know that z is a signal such that $z \notin \mathrm{dom}(\Sigma)$. Let z' be a signal name such that $z' \neq z$ and $\Sigma(z') = (P_{z'} \triangleright Q_{z'})$. We make a case distinction, based on whether $z' \in \mathrm{dom}(O)$ or not.

Case $z' \in \mathrm{dom}(O)$ Suppose that $s_1 \models P_{z'}$. We need to prove that $s_2 \models P_{z'} \vee Q_{z'}$.

By the induction hypothesis (for c_h), $s_2 \models P_{z'} \vee Q_{z'}$.

Case $z' \notin \mathrm{dom}(O)$ There are two subcases, where $s_1 \models P_{z'}$ or $s_1 \models Q_{z'}$

Case $s_1 \models P_{z'}$ By the induction hypothesis (for c_h), $s_2 \models P_{z'}$.

Case $s_1 \models Q_{z'}$ By the induction hypothesis (for c_h), $s_2 \models Q_{z'}$.

We know that $S; O \models \Sigma$. Thus, the signal handler c_h in $\{P_z\}\, c_h\, \{Q_z\}$ could be interrupted by any signal from the $\mathrm{dom}(\Sigma)$ during its execution, except by itself. That yields to $[\,z \mapsto c_h\,]$ and $z : P_z \triangleright Q_z$.

It was assumed that $\Sigma \vdash \forall\alpha.\{P_z\}\, c_h\, \{Q_z\}$ is derivable; thus, using Lemma 5.5.1 we infer P_z **stable** Σ and Q_z **stable** Σ. The Hoare logic rule assumes the following stability assumption for $(z : \forall\alpha.P_z \triangleright Q_z)$: Σ **stable** $\forall\alpha.P_z \triangleright Q_z$. Then, using Lemma 6.2.3, we infer $S; O\,[\,z \mapsto c_h\,] \models \Sigma, z : \forall\alpha.P_z \triangleright Q_z$ and $\Sigma, z : \forall\alpha.P_z \triangleright Q_z$ **pairstable**.

Let $s_1 \models P$. Then, by the induction hypothesis (for c_B), $s_2 \models Q_1'$. That gives us the condition on the Hoare triple required for validity. It remains to check the conditions on signals.

We know that z is a signal name with $\Sigma, z : \forall\alpha.P_z \triangleright Q_z(z) = \forall\alpha.(P_z \triangleright Q_z)$ such that $z \in \mathrm{dom}(O)$. Let z' be a signal name with $\Sigma, z : \forall\alpha.P_z \triangleright Q_z(z') = (P_{z'} \triangleright Q_{z'})$

such that $z' \notin \mathrm{dom}(O)$.

Case $z \in \mathrm{dom}(O)$ We assumed that $s_1 \models P_z$. We need to prove that $s_2 \models P_z \vee Q_z$.

By the induction hypothesis (for c_B), $s_2 \models P_z \vee Q_z$.

Case $z' \notin \mathrm{dom}(O)$ There are two subcases, where $s_1 \models P_{z'}$ or $s_1 \models Q_{z'}$.

Case $s_1 \models P_{z'}$ By the induction hypothesis (for c_B), $s_2 \models P_{z'}$.

Case $s_1 \models Q_{z'}$ By the induction hypothesis (for c_B), $s_2 \models Q_{z'}$.

Finally $\Sigma \vdash \forall\alpha.\{P \wedge P_z\}\,\texttt{bind/1}\,z\,\texttt{to}\,c_h\,\texttt{in}\,c_B\,\{Q \wedge (P_z \vee Q_z)\}\,\texttt{throws}\,\eta$ is valid.

□

6.3.3 Signal Blocking

Lemma 6.3.5 (Soundness of the signal blocking rule) Given the program logic rule for the one-shot signal blocking:

$$\frac{\Sigma \vdash \{P\}\,c\,\{Q\}\,\texttt{throws}\,\eta \qquad P\ \textbf{stable}\ \forall\alpha.P_z \triangleright Q_z \qquad Q\ \textbf{stable}\ \forall\alpha.P_z \triangleright Q_z \qquad \eta\ \textbf{stable}\ \forall\alpha.P_z \triangleright Q_z}{\Sigma, z : \forall\alpha.P_z \triangleright Q_z \vdash \{P\}\,\texttt{block}\,z\,\texttt{in}\,c\,\{Q\}\,\texttt{throws}\,\eta}$$

where $\texttt{throws}\,\eta = \texttt{throws}\,e_1 : Q'_1, \ldots, e_n : Q'_n$.

Suppose the judgment $\Sigma \vdash \{P\}\,c\,\{Q\}\texttt{throws}\eta$ is derivable. and that P **stable** $\forall\alpha.P_z \triangleright Q_z$, Q **stable** $\forall\alpha.P_z \triangleright Q_z$ and η **stable** $\forall\alpha.P_z \triangleright Q_z$ hold. We need to show that the judgment $\Sigma, z : \forall\alpha.P_z \triangleright Q_z \vdash \{P\}\,\texttt{block}\,z\,\texttt{in}\,c\,\{Q\}\,\texttt{throws}\,\eta$ is valid (Definition 6.3.1).

Proof An evaluation of the $\texttt{block}\,z\,\texttt{in}\,c$ can be of the following two forms:

$$\frac{S - z; O - z \Vdash s_1, c \Downarrow s_2}{S; O \Vdash s_1, \texttt{block}\,z\,\texttt{in}\,c \Downarrow s_2}$$

$$\frac{S - z; O - z \Vdash s_1, c \Uparrow e_1, s_2}{S; O \Vdash s_1, \texttt{block}\,z\,\texttt{in}\,c \Uparrow e_1, s_2}$$

Let $\Sigma, z : \forall\alpha.P_z \triangleright Q_z$ **pairstable**, P **stable** $\forall\alpha.P_z \triangleright Q_z$, Q **stable** $\forall\alpha.P_z \triangleright Q_z$ and η **stable** $\forall\alpha.P_z \triangleright Q_z$. Let $S; O \models \Sigma, z : \forall\alpha.P_z \triangleright Q_z$ where $O(z) = c_h$ or $S(z) = c_h$, but not simultaneously. Then, using Lemma 6.2.2, we infer $S - z; O - z \models \Sigma$. Hence we can apply the induction hypothesis to the smaller derivation tree for

$$S - z; O - z \Vdash s_1, c \Downarrow s_2 \qquad \text{or} \qquad S - z; O - z \Vdash s_1, c \Uparrow e_1, s_2$$

Case $\Downarrow$ Let $s_1 \models P$. Then, by the induction hypothesis (for c), $s_2 \models Q$.

Case $\Uparrow e_1$ Let $s_1 \models P$. Then, by the induction hypothesis (for c), $s_2 \models Q'_1$.

That gives us the condition on the Hoare triple required for validity. It remains to check the conditions on signals. The next part of the proof is similar for both cases.

Let z' be a signal name such that $z' \neq z$ and $\Sigma(z') = (P_{z'} \triangleright Q_{z'})$. We make a case distinction, based on whether $z' \in \mathrm{dom}(O)$ or not.

Case $z' \in \mathrm{dom}(O)$ Suppose that $s_1 \models P_{z'}$. We need to prove that $s_2 \models P_{z'} \vee Q_{z'}$. By the induction hypothesis (for c), $s_2 \models P_{z'} \vee Q_{z'}$.

Case $z' \notin \mathrm{dom}(O)$ There are two subcases, where $s_1 \models P_{z'}$ or $s_1 \models Q_{z'}$

Case $s_1 \models P_{z'}$ By the induction hypothesis (for c), $s_2 \models P_{z'}$.

Case $s_1 \models Q_{z'}$ By the induction hypothesis (for c), $s_2 \models Q_{z'}$.

Finally Thus, the judgment $\Sigma, z : \forall\alpha.P_z \triangleright Q_z \vdash \{P\}\,\mathtt{block}\,z\,\mathtt{in}\,c\,\{Q\}\,\mathtt{throws}\,\eta$ is valid. □

6.3.4 Sequential Composition

Lemma 6.3.6 (Soundness of the seq comp with sign and exns) We consider the program rule for the sequential composition:

$$\frac{\Sigma \vdash \{P_1\}\,c_1\,\{P_2\}\,\mathtt{throws}\,\eta \qquad \Sigma \vdash \{P_2\}\,c_2\,\{P_3\}\,\mathtt{throws}\,\eta}{\Sigma \vdash \{P_1\}\,c_1\,;\,c_2\,\{P_3\}\,\mathtt{throws}\,\eta}$$

where $\texttt{throws}\ \eta = \texttt{throws}\ e_1 : Q_1', \ldots, e_n : Q_n'$.

Suppose the judgments $\Sigma \vdash \{P_1\}\, c_1\, \{P_2\}\ \texttt{throws}\ \eta$, $\Sigma \vdash \{P_2\}\, c_2\, \{P_3\}\ \texttt{throws}\ \eta$ are derivable. We need to show that the judgment $\Sigma \vdash \{P_1\}\, c_1 \,;\, c_2\, \{P_3\}\ \texttt{throws}\ \eta$ is valid (Definition 6.3.1).

Proof An evaluation of $c_1 \,;\, c_2$ can be of the following three forms (due to exception convention):

$$\frac{S; O_1 \Vdash s_1, c_1 \Downarrow s_2 \quad S; O_2 \Vdash s_2, c_2 \Downarrow s_3}{S; O_1 * O_2 \Vdash s_1, (c_1 \,;\, c_2) \Downarrow s_3}$$

$$\frac{S; O_1 \Vdash s_1, c_1 \Downarrow s_2 \qquad S; O_2 \Vdash s_2, c_2 \Uparrow e_j, s_3}{S; O_1 * O_2 \Vdash s_1, (c_1; c_2) \Uparrow e_j, s_3}$$

$$\frac{S; O \Vdash s_1, c_1 \Uparrow e_j, s_2}{S; O \Vdash s_1, (c_1; c_2) \Uparrow e_j, s_2}$$

Case 1

$$\frac{S; O_1 \Vdash s_1, c_1 \Downarrow s_2 \quad S; O_2 \Vdash s_2, c_2 \Downarrow s_3}{S; O_1 * O_2 \Vdash s_1, (c_1 \,;\, c_2) \Downarrow s_3}$$

Let Σ **pairstable**, and $S; O_1 * O_2 \models \Sigma$. Then, using Lemma 6.2.1, we infer $S; O_1 \models \Sigma$ and $S; O_2 \models \Sigma$. Hence we can apply the induction hypothesis to the smaller derivation trees for

$$S; O_1 \Vdash s_1, c_1 \Downarrow s_2 \qquad \text{and} \qquad S; O_2 \Vdash s_2, c_2 \Downarrow s_3$$

First, let $s_1 \models P_1$. Then by the induction hypothesis, $s_2 \models P_2$ and, again by the induction hypothesis, $s_3 \models P_3$. That gives us the condition on the Hoare triple required for validity. It remains to check the conditions on signals.

Let z be a signal name with $\Sigma(z) = (P_z \triangleright Q_z)$. We make a case distinction, based on whether $z \in \mathrm{dom}(O_1 * O_2)$ or not.

Case $z \in \mathrm{dom}(O_1 * O_2)$ Suppose that $s_1 \models P_z$. We need to prove that $s_3 \models P_z \vee Q_z$. As O_1 and O_2 have disjoint domains, there are two subcases, where z is

in either $\mathrm{dom}(O_1)$ or in $\mathrm{dom}(O_2)$

Case $z \in \mathrm{dom}(O_1)$ **and** $z \notin \mathrm{dom}(O_2)$ By the induction hypothesis (for c_1), $s_2 \models P_z \vee Q_z$.

Case $s_2 \models P_z$ By the induction hypothesis (for c_2), $s_3 \models P_z$.

Case $s_2 \models Q_z$ By the induction hypothesis (for c_2), $s_3 \models Q_z$.

Case $z \notin \mathrm{dom}(O_1)$ **and** $z \in \mathrm{dom}(O_2)$ By the induction hypothesis (for c_1), $s_2 \models P_z$. Then, by the induction hypothesis (for c_2), $s_3 \models P_z \vee Q_z$.

Case $z \notin \mathrm{dom}(O_1 * O_2)$ in this case, $z \notin \mathrm{dom}(O_1)$ and $z \notin \mathrm{dom}(O_2)$.

Case $s_1 \models P_z$ By the induction hypothesis (for c_1), $s_2 \models P_z$. Then, by the induction hypothesis (for c_2), $s_3 \models P_z$.

Case $s_1 \models Q_z$ By the induction hypothesis (for c_1), $s_2 \models Q_z$.

Then, by the induction hypothesis (for c_2), $s_3 \models Q_z$.

Case 2

$$\frac{S; O_1 \Vdash s_1, c_1 \Downarrow s_2 \qquad S; O_2 \Vdash s_2, c_2 \Uparrow e_j, s_3}{S; O_1 * O_2 \Vdash s_1, (c_1; c_2) \Uparrow e_j, s_3}$$

Let Σ **pairstable**, and $S; O_1 * O_2 \models \Sigma$. Then, using Lemma 6.2.1, we infer $S; O_1 \models \Sigma$ and $S; O_2 \models \Sigma$. Hence we can apply the induction hypothesis to the smaller derivation trees for

$$S; O_1 \Vdash s_1, c_1 \Downarrow s_2 \qquad \text{and} \qquad S; O_2 \Vdash s_2, c_2 \Uparrow e_j, s_3$$

First, let $s_1 \models P_1$. Then by the induction hypothesis, $s_2 \models P_2$ and, again by the induction hypothesis, $s_3 \models Q'_j$. That gives us the condition on the Hoare triple required for validity. It remains to check the conditions on signals.

Let z be a signal name with $\Sigma(z) = (P_z \triangleright Q_z)$. We make a case distinction, based on whether $z \in \mathrm{dom}(O_1 * O_2)$ or not.

Case $z \in \mathrm{dom}(O_1 * O_2)$ Suppose that $s_1 \models P_z$. We need to prove that $s_3 \models P_z \vee$

Q_z. As O_1 and O_2 have disjoint domains, there are two subcases, where z is in either $\mathrm{dom}(O_1)$ or in $\mathrm{dom}(O_2)$

Case $z \in \mathrm{dom}(O_1)$ **and** $z \notin \mathrm{dom}(O_2)$ By the induction hypothesis (for c_1), $s_2 \models P_z \vee Q_z$.

Case $s_2 \models P_z$ By the induction hypothesis (for c_2), $s_3 \models P_z$.

Case $s_2 \models Q_z$ By the induction hypothesis (for c_2), $s_3 \models Q_z$.

Case $z \notin \mathrm{dom}(O_1)$ **and** $z \in \mathrm{dom}(O_2)$ By the induction hypothesis (for c_1), $s_2 \models P_z$. Then, by the induction hypothesis (for c_2), $s_3 \models P_z \vee Q_z$.

Case $z \notin \mathrm{dom}(O_1 * O_2)$ in this case, $z \notin \mathrm{dom}(O_1)$ and $z \notin \mathrm{dom}(O_2)$.

Case $s_1 \models P_z$ By the induction hypothesis (for c_1), $s_2 \models P_z$. Then, by the induction hypothesis (for c_2), $s_3 \models P_z$.

Case $s_1 \models Q_z$ By the induction hypothesis (for c_1), $s_2 \models Q_z$.

Then, by the induction hypothesis (for c_2), $s_3 \models Q_z$.

Case 3

$$\frac{S; O \Vdash s_1, c_1 \Uparrow e_j, s_2}{S; O \Vdash s_1, (c_1; c_2) \Uparrow e_j, s_2}$$

Let Σ **pairstable**, and $S; O \models \Sigma$. Hence we can apply the induction hypothesis to the smaller derivation tree for

$$S; O \Vdash s_1, c_1 \Uparrow e_j, s_2$$

First, let $s_1 \models P_1$. Then by the induction hypothesis, $s_2 \models Q'_j$. That gives us the condition on the Hoare triple required for validity. It remains to check the conditions on signals.

Let z be a signal name with $\Sigma(z) = (P_z \triangleright Q_z)$. We make a case distinction, based on whether $z \in \mathrm{dom}(O)$ or not.

Case $z \in \mathrm{dom}(O)$ Suppose that $s_1 \models P_z$. By the induction hypothesis (for c_1), $s_2 \models P_z \vee Q_z$.

Case $z \notin \mathrm{dom}(O)$ We make a case distinction, based on whether $s_1 \models P_z$ or $s_1 \models Q_z$.

Case $s_1 \models P_z$ By the induction hypothesis (for c_1), $s_2 \models P_z$.

Case $s_1 \models Q_z$ By the induction hypothesis (for c_1), $s_2 \models Q_z$.

Finally Thus, the judgment $\Sigma \vdash \{P_1\}\, c_1 \,;\, c_2\, \{P_3\}\, \texttt{throws}\, \eta$ is valid for every case.

□

6.3.5 Atomic Commands

Lemma 6.3.7 (Soundness of the atomic rule) We consider the program rule for the atomic command:

$$\frac{\{P\}\, a\, \{Q\} \qquad (P \triangleright Q)\ \textbf{stable}\ \Sigma \qquad \eta\ \textbf{stable}\ \Sigma \qquad \forall z.\Sigma(z) = z : \forall \alpha. P_z \triangleright Q_z \qquad \exists E.Q \implies P_z\,[\,\alpha \mapsto E\,]}{\Sigma \vdash \{P\}\, a\, \{Q\}\, \texttt{throws}\, \eta}$$

Suppose the judgment $\{P\}\, a\, \{Q\}$ is derivable. Suppose that $(P \triangleright Q)$ **stable** Σ and η **stable** Σ. Assume that $\forall z.\Sigma(z) = z : \forall \alpha. P_z \triangleright Q_z \quad \exists E.Q \implies P_z\,[\,\alpha \mapsto E\,]$ is satisfied. We need to show that the judgment $\Sigma \vdash \{P\}\, a\, \{Q\}\, \texttt{throws}\, \eta$ is valid (Definition 6.3.1).

Proof We have an atomic command a. Assume the specification $\{P\}\, a\, \{Q\}$ as given. Assume that $S; O \Vdash s_1, a \Downarrow s_2$. Assume the Hoare logic rule's stability assumptions: $(P \triangleright Q)$ **stable** Σ and η **stable** Σ holds. Assume that $\forall z.\Sigma(z) = z : \forall \alpha. P_z \triangleright Q_z\ \exists E.Q \implies P_z\,[\,\alpha \mapsto E\,]$ is satisfied. Finally, assume the soundness of atomic a as given. □

6.3.6 Repetitive `while` Command

Lemma 6.3.8 (Soundness of the `while` rule) We consider the program rule for the `while`:

$$\frac{\Sigma \vdash \{I \wedge E\}\, c\, \{I\}\ \texttt{throws}\ \eta \qquad \neg E\ \textbf{stable}\ \Sigma}{\Sigma \vdash \{I\}\ \texttt{while}\ E\ \texttt{do}\ c\, \{I \wedge \neg E\}\ \texttt{throws}\ \eta}$$

Suppose the judgment $\Sigma \vdash \{I \wedge E\}\, c\, \{I\}\ \texttt{throws}\ \eta$ is derivable. Also suppose that $\neg E$ **stable** Σ holds. We need to show that the judgment $\Sigma \vdash \{I\}\ \texttt{while}\ E\ \texttt{do}\ c\, \{I \wedge \neg E\}\ \texttt{throws}\ \eta$ is valid (Definition 6.3.1).

Proof An evaluation of $\texttt{while}\ E\ \texttt{do}\ c$ can be of the following three forms (due to exception convention):

$$\frac{s_1 \models \neg E}{S; O \Vdash s_1, \texttt{while}\ E\ \texttt{do}\ c \Downarrow s_1}$$

$$\frac{s_1 \models E \qquad S; O \Vdash s_1, c \Uparrow e_1, s_2}{S; O \Vdash s_1, \texttt{while}\ E\ \texttt{do}\ c \Uparrow e_1, s_2}$$

$$\frac{s_1 \models E \qquad S; O_1 \Vdash s_1, c \Downarrow s_2 \qquad S; O_2 \Vdash s_2, \texttt{while}\ E\ \texttt{do}\ c \Downarrow s_3}{S; O_1 * O_2 \Vdash s_1, \texttt{while}\ E\ \texttt{do}\ c \Downarrow s_3}$$

The proof is by induction over derivation of `while` logic and semantic rules.

Case while-false $s_1 \models \neg E$.

Let Σ **pairstable**, $S; O \models \Sigma$, $\neg E$ **stable** Σ and $s_1 \models I$. As loop never runs, the final state equals to the initial state. Thus, the final state $s_1 \models I$. It was assumed that E doesn't hold. Therefore, $s_1 \models I \wedge \neg E$.

It remains to check the conditions on signals. Let z be a signal name with $\Sigma(z) = (P_z \triangleright Q_z)$. We make a case distinction, based on whether $s_1 \models P_z$ or $s_1 \models Q_z$.

Case $s_1 \models P_z$ then trivially $s_1 \models P_z$.

Case $s_1 \models Q_z$ then trivially $s_1 \models Q_z$.

Case while-true-exception $s_1 \models E$ and $\Uparrow e_1$.

Let Σ **pairstable**, $S; O \models \Sigma$, $\neg E$ **stable** Σ and $s_1 \models I$. Hence we can apply the induction hypothesis to the smaller derivation trees for

$$S; O \Vdash s_1, c \Uparrow e_1, s_2$$

Then by the induction hypothesis (for c), $s_2 \models I$. That gives us the condition on the Hoare triple required for validity. It remains to check the conditions on signals.

Let z be a signal name with $\Sigma(z) = (P_z \triangleright Q_z)$. We make a case distinction, based on whether $z \in \mathrm{dom}(O)$ or not.

Case $z \in \mathrm{dom}(O)$ Suppose that $s_1 \models P_z$. By the induction hypothesis (for c), $s_2 \models P_z \vee Q_z$.

Case $z \notin \mathrm{dom}(O)$ We make a case distinction, based on whether $s_1 \models P_z$ or $s_1 \models Q_z$.

Case $s_1 \models P_z$ By the induction hypothesis (for c), $s_2 \models P_z$.

Case $s_1 \models Q_z$ By the induction hypothesis (for c), $s_2 \models Q_z$.

Case while-true $s_1 \models E$.

Let Σ **pairstable**, $S; O_1 * O_2 \models \Sigma$, and $\neg E$ **stable** Σ. Then, using Lemma 6.2.1, we infer $S; O_1 \models \Sigma$ and $S; O_2 \models \Sigma$.

A constructions of the derivation tree is done using WhileTrue operational semantic rule. We can observer that there are two subtrees of the big-step derivation. One subtree is for the body of the `while`, and the other for another iteration of the `while` loop again. Hence we can apply the induction hypothesis to the smaller derivation trees for

$$S; O_1 \Vdash s_1, c \Downarrow s_2 \qquad \text{and} \qquad S; O_2 \Vdash s_2, \texttt{while}\, E\, \texttt{do}\, c \Downarrow s_3$$

First, let $s_1 \models I$. Then by the induction hypothesis (for c), $s_2 \models I$.

When it comes to the `while` above the inference, the size of the program logic tree for inferring the judgement about it is the same; however, the size of the big-step semantics tree is strictly smaller than the tree we are analyzing.

Again by the induction hypothesis (for $\texttt{while}\, E\, \texttt{do}\, c$), $s_3 \models I \wedge \neg E$. That gives us the condition on the Hoare triple required for validity. It remains to check the conditions on signals.

Let z be a signal name with $\Sigma(z) = (P_z \triangleright Q_z)$. We make a case distinction, based on whether $z \in \mathrm{dom}(O_1 * O_2)$ or not.

Case $z \in \mathrm{dom}(O_1 * O_2)$ Suppose that $s_1 \models P_z$. We need to prove that $s_3 \models P_z \vee Q_z$. As O_1 and O_2 have disjoint domains, there are two subcases, where z is in either $\mathrm{dom}(O_1)$ or in $\mathrm{dom}(O_2)$

Case $z \in \mathrm{dom}(O_1)$ **and** $z \notin \mathrm{dom}(O_2)$ By the induction hypothesis (for c), $s_2 \models P_z \vee Q_z$.

Case $s_2 \models P_z$ By the induction hypothesis (for $\texttt{while}\, E\, \texttt{do}\, c$), $s_3 \models P_z$.

Case $s_2 \models Q_z$ By the induction hypothesis (for $\texttt{while}\, E\, \texttt{do}\, c$), $s_3 \models Q_z$.

Case $z \notin \mathrm{dom}(O_1)$ **and** $z \in \mathrm{dom}(O_2)$ By the induction hypothesis (for c), $s_2 \models P_z$. Then, by the induction hypothesis (for $\texttt{while}\, E\, \texttt{do}\, c$), $s_3 \models P_z \vee Q_z$.

Case $z \notin \mathrm{dom}(O_1 * O_2)$ in this case, $z \notin \mathrm{dom}(O_1)$ and $z \notin \mathrm{dom}(O_2)$.

Case $s_1 \models P_z$ By the induction hypothesis (for c), $s_2 \models P_z$. Then, by the induction hypothesis (for $\texttt{while}\, E\, \texttt{do}\, c$), $s_3 \models P_z$.

Case $s_1 \models Q_z$ By the induction hypothesis (for c), $s_2 \models Q_z$. Then, by the induction hypothesis (for $\texttt{while}\, E\, \texttt{do}\, c$), $s_3 \models Q_z$.

Finally Thus, the judgment $\Sigma \vdash \{I\}\, \texttt{while}\, E\, \texttt{do}\, c\, \{I \wedge \neg E\}\, \texttt{throws}\, \eta$ is valid.

□

6.3.7 Rule of Consequence

Lemma 6.3.9 (Soundness of the rule of consequence) We consider the program rule:

$$\frac{\Sigma \vdash \{P\}\, c\, \{Q\}\ \texttt{throws}\ \eta \quad P' \Rightarrow P \quad Q \Rightarrow Q' \quad \eta \Rightarrow \eta' \qquad P' \textbf{ stable } \Sigma \qquad Q' \textbf{ stable } \Sigma \qquad \eta' \textbf{ stable } \Sigma}{\Sigma \vdash \{P'\}\, c\, \{Q'\}\ \texttt{throws}\ \eta'}$$

where $\eta = e_1 : Q_1, \ldots, e_n : Q_n$ and $\eta' = e_1 : Q'_1, \ldots, e_n : Q'_n$

Suppose the judgment $\Sigma \vdash \{P\}\, c\, \{Q\}\texttt{throws}\eta$ is derivable. Also suppose that $P' \Rightarrow P$, $Q \Rightarrow Q'$, $\eta \Rightarrow \eta'$, P' **stable** Σ, Q' **stable** Σ and η' **stable** Σ hold. We need to show that the judgment $\Sigma \vdash \{P'\}\, c\, \{Q'\}\ \texttt{throws}\ \eta'$ is valid (Definition 6.3.1).

Proof An evaluation of c can be of the following two forms:

$$S; O \Vdash s_1, c \Downarrow s_2 \qquad \text{and} \qquad S; O \Vdash s_1, c \Uparrow e_j, s_2$$

Case $\Downarrow$ Let Σ **pairstable**, $S; O \models \Sigma$, $P' \Rightarrow P$, $Q \Rightarrow Q'$, $\eta \Rightarrow \eta'$, P' **stable** Σ, Q' **stable** Σ and Q'_j **stable** Σ.

First, let $s_1 \models P'$. Then by implications $P' \Rightarrow P$, we get $s_1 \models P$. Hence we can apply the induction hypothesis to the derivation tree for c above the line.

Then by the induction hypothesis, $s_2 \models Q$. Then by the implication $Q \Rightarrow Q'$, $s_2 \models Q'$.

Case $\Uparrow e_1$ Let Σ **pairstable**, $S; O \models \Sigma$, $P' \Rightarrow P$, $Q \Rightarrow Q'$, $\eta \Rightarrow \eta'$, P' **stable** Σ, Q' **stable** Σ and Q'_j **stable** Σ.

First, let $s_1 \models P'$. Then by implications $P' \Rightarrow P$, we get $s_1 \models P$. Hence we can apply the induction hypothesis to the derivation tree for c above the line.

Then by the induction hypothesis, $s_2 \models Q_j$. Then by the implication $\eta \Rightarrow \eta'$, $s_2 \models Q'_j$.

That gives us the condition on the Hoare triple required for validity. It remains to check the conditions on signals. Next part of the proof is analogous for both cases.

Let z be a signal name with $\Sigma(z) = (P_z \triangleright Q_z)$. We make a case distinction, based on whether $z \in \mathrm{dom}(O)$ or not.

Case $z \in \mathrm{dom}(O)$ Suppose that $s_1 \models P_z$. We need to prove that $s_2 \models P_z \vee Q_z$. By the induction hypothesis (for c), $s_2 \models P_z \vee Q_z$.

Case $z \notin \mathrm{dom}(O)$ There are two subcases, where $s_1 \models P_z$ or $s_1 \models Q_z$

Case $s_1 \models P_z$ By the induction hypothesis (for c), $s_2 \models P_z$.

Case $s_1 \models Q_z$ By the induction hypothesis (for c), $s_2 \models Q_z$.

Finally The judgment $\Sigma \vdash \{P'\}\, c\, \{Q'\}\ \texttt{throws}\ \eta'$ is valid for all cases. □

6.3.8 Rule of Conjunction

Lemma 6.3.10 (Soundness of the rule of conjunction) We consider the program rule:

$$\frac{\Sigma \vdash \{P_1\}\, c\, \{Q_1\}\ \texttt{throws}\ \eta_1 \qquad \Sigma \vdash \{P_2\}\, c\, \{Q_2\}\ \texttt{throws}\ \eta_2}{\Sigma \vdash \{P_1 \wedge P_2\}\, c\, \{Q_1 \wedge Q_2\}\ \texttt{throws}\ \eta_1 \wedge \eta_2}$$

where $\eta_1 = e_1 : Q'_1, \ldots, e_n : Q'_n$ and $\eta_2 = e_1 : Q"_1, \ldots, e_n : Q"_n$.

Suppose the judgments $\Sigma \vdash \{P_1\}\, c\, \{Q_1\}\ \texttt{throws}\ \eta_1$ and $\Sigma \vdash \{P_2\}\, c\, \{Q_2\}\ \texttt{throws}\ \eta_2$ are derivable. We need to show that the judgment $\Sigma \vdash \{P_1 \wedge P_2\}\, c\, \{Q_1 \wedge Q_2\}\ \texttt{throws}\ \eta_1 \wedge \eta_2$ is valid (Definition 6.3.1).

Proof An evaluation of c can be of the following two forms:

$$S; O \Vdash s_1, c \Downarrow s_2 \quad \text{and} \quad S; O \Vdash s_1, c \Uparrow e_1, s_2$$

Case $\Downarrow$ Let Σ **pairstable**, and $S; O \models \Sigma$. Hence we can apply the induction hypothesis to the derivation trees for c above the line.

First, let $s_1 \models P_1 \wedge P_2$. That is equivalent to $s_1 \models P_1$ and $s_1 \models P_2$. Let $s_1 \models P_1$, then by the induction hypothesis (for the first c above the line), $s_2 \models Q_1$. Let $s_1 \models P_2$, then by the induction hypothesis (for the second c above the line), $s_2 \models Q_2$. That is equivalent to $s_2 \models Q_1 \wedge Q_2$.

Case $\Uparrow e_1$ Let Σ **pairstable**, and $S; O \models \Sigma$. Hence we can apply the induction hypothesis to the derivation trees for c above the line.

First, let $s_1 \models P_1 \wedge P_2$. That is equivalent to $s_1 \models P_1$ and $s_1 \models P_2$. Let $s_1 \models P_1$, then by the induction hypothesis (for the first c above the line), $s_2 \models Q'_1$. Let $s_1 \models P_2$, then by the induction hypothesis (for the second c above the line), $s_2 \models Q"_1$. That is equivalent to $s_2 \models Q'_1 \wedge Q"_1$.

That gives us the condition on the Hoare triple required for validity. It remains to check the conditions on signals. Next part of the proof is analogous for both cases.

Let z be a signal name with $\Sigma(z) = (P_z \triangleright Q_z)$. We make a case distinction, based on whether $z \in \mathrm{dom}(O)$ or not.

Case $z \in \mathrm{dom}(O)$ Suppose that $s_1 \models P_z$. We need to prove that $s_2 \models P_z \vee Q_z$. By the induction hypothesis (for c), $s_2 \models P_z \vee Q_z$.

Case $z \notin \mathrm{dom}(O)$ There are two subcases, where $s_1 \models P_z$ or $s_1 \models Q_z$

Case $s_1 \models P_z$ By the induction hypothesis (for c), $s_2 \models P_z$.

Case $s_1 \models Q_z$ By the induction hypothesis (for c), $s_2 \models Q_z$.

Finally Thus, the judgment $\Sigma \vdash \{P_1 \wedge P_2\}\, c\, \{Q_1 \wedge Q_2\}\, \texttt{throws}\, \eta_1 \wedge \eta_2$ is valid for every case. $\square$

6.3.9 Conditional `if-else` Command

Lemma 6.3.11 (Soundness of the `if-else` rule) We consider the program rule:

$$\frac{\Sigma \vdash \{E \wedge P\}\, c_1\, \{Q\}\, \texttt{throws}\, \eta \qquad \Sigma \vdash \{\neg E \wedge P\}\, c_2\, \{Q\}\, \texttt{throws}\, \eta}{\Sigma \vdash \{P\}\, \texttt{if}\, E\, \texttt{then}\, c_1\, \texttt{else}\, c_2\, \{Q\}\, \texttt{throws}\, \eta}$$

where $\eta = e_1 : Q'_1, \ldots, e_n : Q'_n$.

Suppose the judgments $\Sigma \vdash \{E \wedge P\}\, c_1\, \{Q\} \texttt{throws} \eta$ and $\Sigma \vdash \{\neg E \wedge P\}\, c_2\, \{Q\} \texttt{throws} \eta$ are derivable. We need to show that the judgment $\Sigma \vdash \{P\}\, \texttt{if}\, E\, \texttt{then}\, c_1\, \texttt{else}\, c_2\, \{Q\}\, \texttt{throws}\, \eta$ is valid (Definition 6.3.1).

Proof An evaluation of $\texttt{if}\, E\, \texttt{then}\, c_1\, \texttt{else}\, c_2$ can be of the following four forms (due to exception convention):

$$\frac{s_1 \models E \qquad S; O \Vdash s_1, c_1 \Downarrow s_2}{S; O \Vdash s_1, \texttt{if}\, E\, \texttt{then}\, c_1\, \texttt{else}\, c_2 \Downarrow s_2}$$

$$\frac{s_1 \models \neg E \qquad S; O \Vdash s_1, c_2 \Downarrow s_3}{S; O \Vdash s_1, \texttt{if}\, E\, \texttt{then}\, c_1\, \texttt{else}\, c_2 \Downarrow s_3}$$

$$\frac{s_1 \models E \qquad S; O \Vdash s_1, c_1 \Uparrow e_1, s_2}{S; O \Vdash s_1, \texttt{if}\, E\, \texttt{then}\, c_1\, \texttt{else}\, c_2 \Uparrow e_1, s_2}$$

$$\frac{s_1 \models \neg E \qquad S; O \Vdash s_1, c_2 \Uparrow e_1, s_3}{S; O \Vdash s_1, \texttt{if}\, E\, \texttt{then}\, c_1\, \texttt{else}\, c_2 \Uparrow e_1, s_3}$$

Case $s_1 \models E$ Let Σ **pairstable**, and $S; O \models \Sigma$.

Case $\Downarrow$ Let $s_1 \models P_1 \wedge E$. Then by the induction hypothesis (for c_1), $s_2 \models Q$.

Case $\Uparrow e_1$ Let $s_1 \models P_1 \wedge E$. Then by the induction hypothesis (for c_1), $s_2 \models Q'_1$.

That gives us the condition on the Hoare triple required for validity. It remains to check the conditions on signals. Next part of the proof is analogous for both

subcases.

Let z be a signal name with $\Sigma(z) = (P_z \triangleright Q_z)$. We make a case distinction, based on whether $z \in \mathrm{dom}(O)$ or not.

Case $z \in \mathrm{dom}(O)$ Suppose that $s_1 \models P_z$. We need to prove that $s_2 \models P_z \vee Q_z$. By the induction hypothesis (for c_1), $s_2 \models P_z \vee Q_z$.

Case $z \notin \mathrm{dom}(O)$ There are two subcases, where $s_1 \models P_z$ or $s_1 \models Q_z$

Case $s_1 \models P_z$ By the induction hypothesis (for c_1), $s_2 \models P_z$.

Case $s_1 \models Q_z$ By the induction hypothesis (for c_1), $s_2 \models Q_z$.

Case $s_1 \models \neg E$ Let Σ **pairstable**, and $S; O \models \Sigma$.

Case $\Downarrow$ Let $s_1 \models P_1 \wedge \neg E$. Then by the induction hypothesis (for c_2), $s_2 \models Q$.

Case $\Uparrow e_1$ Let $s_1 \models P_1 \wedge \neg E$. Then by the induction hypothesis (for c_2), $s_2 \models Q_1'$.

That gives us the condition on the Hoare triple required for validity. It remains to check the conditions on signals. Next part of the proof is analogous for both subcases.

Let z be a signal name with $\Sigma(z) = (P_z \triangleright Q_z)$. We make a case distinction, based on whether $z \in \mathrm{dom}(O)$ or not.

Case $z \in \mathrm{dom}(O)$ Suppose that $s_1 \models P_z$. We need to prove that $s_2 \models P_z \vee Q_z$. By the induction hypothesis (for c_2), $s_2 \models P_z \vee Q_z$.

Case $z \notin \mathrm{dom}(O)$ There are two subcases, where $s_1 \models P_z$ or $s_1 \models Q_z$

Case $s_1 \models P_z$ By the induction hypothesis (for c_2), $s_2 \models P_z$.

Case $s_1 \models Q_z$ By the induction hypothesis (for c_2), $s_2 \models Q_z$.

Finally Thus, the judgment $\Sigma \vdash \{P\}\ \texttt{if}\ E\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \{Q\}\ \texttt{throws}\ \eta$ is valid.

□

6.3.10 Exception `throw` Command

Lemma 6.3.12 (Soundness of the `throw` rule) We consider the program rule for the `throw`:

$$\frac{\eta \textbf{ stable } \Sigma \qquad Q \textbf{ stable } \Sigma}{\Sigma \vdash \{\eta(e_j)\}\,\texttt{throw}\,e_j\,\{Q\}\,\texttt{throws}\,\eta}$$

where $\texttt{throws}\,\eta = \texttt{throws}\,e_1 : Q'_1, \ldots, e_n : Q'_n$. And $\eta(e_j) = Q'_j$ in a precondition position, means that Q'_j holds immediately before the exception is thrown. Suppose that η **stable** Σ and Q **stable** Σ hold.

We need to show that the judgment $\Sigma \vdash \{\eta(e_j)\}\,\texttt{throw}\,e_j\,\{Q\}\,\texttt{throws}\,\eta$ is valid (Definition 6.3.1).

Proof An evaluation of $\texttt{throw}\,e$ can only be inferred in the following way:

$$\frac{}{S; O \Vdash s, \texttt{throw}\,e_j \Uparrow e_j, s}$$

Let Σ **pairstable**, $S; O \models \Sigma$, η **stable** Σ and Q **stable** Σ. Let $s \models Q'_j$ such that $\eta(e_j) = Q'_j$.

The command `throw` does not change the initial state s, but raises an exception e_j. As there is no change of the state s, we infer $s \models Q'_j$ afterwards.

It remains to check the conditions on signals. Let z be a signal name with $\Sigma(z) = (P_z \triangleright Q_z)$. We make a case distinction, based on whether $s \models P_z$ or $s \models Q_z$.

Case $s \models P_z$ then trivially $s \models P_z$.

Case $s \models Q_z$ then trivially $s \models Q_z$.

Thus, the judgment $\Sigma \vdash \{\eta(e_j)\}\,\texttt{throw}\,e_j\,\{Q\}\,\texttt{throws}\,\eta$ is valid. □

6.3.11 Exception Handling

Lemma 6.3.13 (Soundness of the exception handling) We consider the program rule for the exception handling:

$$\frac{\Sigma \vdash \{P\}\, c_B\, \{Q_b\}\, \texttt{throws}\, \eta, e_k : Q'_k \qquad \Sigma \vdash \{Q'_k\}\, c_h\, \{Q_h\}\, \texttt{throws}\, \eta}{\Sigma \vdash \{P\}\, \texttt{try}\, c_B\, \texttt{handle}\, e_k\, \texttt{by}\, c_h\, \{Q_b \vee Q_h\}\, \texttt{throws}\, \eta}$$

where $\texttt{throws}\ \eta = \texttt{throws}\ e_1 : Q'_1, \ldots, e_n : Q'_n$.

Suppose the judgments
$\Sigma \vdash \{P\}\, c_B\, \{Q_b\}\, \texttt{throws}\, \eta, e_k : Q'_k$ and $\Sigma \vdash \{Q'_k\}\, c_h\, \{Q_h\}\, \texttt{throws}\, \eta$ are derivable. We need to show that the judgment $\Sigma \vdash \{P\}\, \texttt{try}\, c_B\, \texttt{handle}\, e_k\, \texttt{by}\, c_h\, \{Q_b \vee Q_h\}\, \texttt{throws}\, \eta$ is valid (Definition 6.3.1).

Proof An evaluation of $\texttt{try}\, c_B\, \texttt{handle}\, e_k\, \texttt{by}\, c_h$ can be of the following four forms (due to exception convention):

$$\frac{S; O \Vdash s_1, c_B \Downarrow s_2}{S; O \Vdash s_1, \texttt{try}\, c_B\, \texttt{handle}\, e_k\, \texttt{by}\, c_h \Downarrow s_2}$$

$$\frac{S; O_1 \Vdash s_1, c_B \Uparrow e_k, s_2 \quad S; O_2 \Vdash s_2, c_h \Downarrow s_3}{S; O_1 * O_2 \Vdash s_1, \texttt{try}\, c_B\, \texttt{handle}\, e_k\, \texttt{by}\, c_h \Downarrow s_3}$$

$$\frac{S; O_1 \Vdash s_1, c_B \Uparrow e_k, s_2 \quad S; O_2 \Vdash s_2, c_h \Uparrow e_2, s_3}{S; O_1 * O_2 \Vdash s_1, \texttt{try}\, c_B\, \texttt{handle}\, e_k\, \texttt{by}\, c_h \Uparrow e_2, s_3}$$

$$\frac{S; O \Vdash s_1, c_B \Uparrow e_2, s_2 \quad e_2 \neq e_k}{S; O \Vdash s_1, \texttt{try}\, c_B\, \texttt{handle}\, e_k\, \texttt{by}\, c_h \Uparrow e_2, s_2}$$

Case 1

$$\frac{S; O \Vdash s_1, c_B \Downarrow s_2}{S; O \Vdash s_1, \texttt{try}\, c_B\, \texttt{handle}\, e_k\, \texttt{by}\, c_h \Downarrow s_2}$$

Let Σ **pairstable**, and $S; O \models \Sigma$. Hence we can apply the induction hypothesis to the smaller derivation tree for

$$S; O \Vdash s_1, c_B \Downarrow s_2$$

First, let $s_1 \models P$. Then by the induction hypothesis, $s_2 \models Q_b$. That gives us the condition on the Hoare triple required for validity. It remains to check the conditions on signals.

Let z be a signal name with $\Sigma(z) = (P_z \triangleright Q_z)$. We make a case distinction, based on whether $z \in \mathrm{dom}(O)$ or not.

Case $z \in \mathrm{dom}(O)$ Suppose that $s_1 \models P_z$. By the induction hypothesis (for c_B), $s_2 \models P_z \vee Q_z$.

Case $z \notin \mathrm{dom}(O)$ We make a case distinction, based on whether $s_1 \models P_z$ or $s_1 \models Q_z$.

Case $s_1 \models P_z$ By the induction hypothesis (for c_B), $s_2 \models P_z$.

Case $s_1 \models Q_z$ By the induction hypothesis (for c_B), $s_2 \models Q_z$.

Case 2

$$\frac{S; O_1 \Vdash s_1, c_B \Uparrow e_k, s_2 \quad S; O_2 \Vdash s_2, c_h \Downarrow s_3}{S; O_1 * O_2 \Vdash s_1, \texttt{try}\ c_B\ \texttt{handle}\ e_k\ \texttt{by}\ c_h \Downarrow s_3}$$

Let Σ **pairstable**, and $S; O_1 * O_2 \models \Sigma$. Then, using Lemma 6.2.1, we infer $S; O_1 \models \Sigma$ and $S; O_2 \models \Sigma$. Hence we can apply the induction hypothesis to the smaller derivation trees for

$$S; O_1 \Vdash s_1, c_B \Uparrow e_k, s_2 \qquad \text{and} \qquad S; O_2 \Vdash s_2, c_h \Downarrow s_3$$

First, let $s_1 \models P$. Then by the induction hypothesis (for c_B), $s_2 \models Q'_k$ and, again by the induction hypothesis (for c_h), $s_3 \models Q_h$. That gives us the condition on the Hoare triple required for validity. It remains to check the conditions on signals.

Let z be a signal name with $\Sigma(z) = (P_z \triangleright Q_z)$. We make a case distinction, based on whether $z \in \mathrm{dom}(O_1 * O_2)$ or not.

Case $z \in \mathrm{dom}(O_1 * O_2)$ Suppose that $s_1 \models P_z$. We need to prove that $s_3 \models P_z \vee Q_z$. As O_1 and O_2 have disjoint domains, there are two subcases, where z is in either $\mathrm{dom}(O_1)$ or in $\mathrm{dom}(O_2)$

Case $z \in \mathrm{dom}(O_1)$ **and** $z \notin \mathrm{dom}(O_2)$ By the induction hypothesis (for c_B), $s_2 \models P_z \vee Q_z$.

Case $s_2 \models P_z$ By the induction hypothesis (for c_h), $s_3 \models P_z$.

Case $s_2 \models Q_z$ By the induction hypothesis (for c_h), $s_3 \models Q_z$.

Case $z \notin \mathrm{dom}(O_1)$ **and** $z \in \mathrm{dom}(O_2)$ By the induction hypothesis (for c_B), $s_2 \models P_z$. Then, by the induction hypothesis (for c_h), $s_3 \models P_z \vee Q_z$.

Case $z \notin \mathrm{dom}(O_1 * O_2)$ in this case, $z \notin \mathrm{dom}(O_1)$ and $z \notin \mathrm{dom}(O_2)$.

Case $s_1 \models P_z$ By the induction hypothesis (for c_B), $s_2 \models P_z$.

Then, by the induction hypothesis (for c_h), $s_3 \models P_z$.

Case $s_1 \models Q_z$ By the induction hypothesis (for c_B), $s_2 \models Q_z$. Then, by the induction hypothesis (for c_h), $s_3 \models Q_z$.

Case 3

$$\frac{S; O_1 \Vdash s_1, c_B \Uparrow e_k, s_2 \quad S; O_2 \Vdash s_2, c_h \Uparrow e_2, s_3}{S; O_1 * O_2 \Vdash s_1, \texttt{try}\ c_B\ \texttt{handle}\ e_k\ \texttt{by}\ c_h \Uparrow e_2, s_3}$$

Let Σ **pairstable**, and $S; O_1 * O_2 \models \Sigma$. Then, using Lemma 6.2.1, we infer $S; O_1 \models \Sigma$ and $S; O_2 \models \Sigma$. Hence we can apply the induction hypothesis to the smaller derivation trees for

$$S; O_1 \Vdash s_1, c_B \Uparrow e_k, s_2 \qquad \text{and} \qquad S; O_2 \Vdash s_2, c_h \Uparrow e_2, s_3$$

First, let $s_1 \models P$. Then by the induction hypothesis (for c_B), $s_2 \models Q'_k$ and, again by the induction hypothesis (for c_h), $s_3 \models Q'_2$. That gives us the condition on the Hoare triple required for validity. It remains to check the conditions on signals.

Let z be a signal name with $\Sigma(z) = (P_z \triangleright Q_z)$. We make a case distinction, based on whether $z \in \mathrm{dom}(O_1 * O_2)$ or not.

Case $z \in \mathrm{dom}(O_1 * O_2)$ Suppose that $s_1 \models P_z$. We need to prove that $s_3 \models P_z \vee Q_z$. As O_1 and O_2 have disjoint domains, there are two subcases, where z is in either $\mathrm{dom}(O_1)$ or in $\mathrm{dom}(O_2)$

Case $z \in \mathrm{dom}(O_1)$ **and** $z \notin \mathrm{dom}(O_2)$ By the induction hypothesis (for c_B), $s_2 \models P_z \vee Q_z$.

Case $s_2 \models P_z$ By the induction hypothesis (for c_h), $s_3 \models P_z$.

Case $s_2 \models Q_z$ By the induction hypothesis (for c_h), $s_3 \models Q_z$.

Case $z \notin \mathrm{dom}(O_1)$ **and** $z \in \mathrm{dom}(O_2)$ By the induction hypothesis (for c_B), $s_2 \models P_z$. Then, by the induction hypothesis (for c_h), $s_3 \models P_z \vee Q_z$.

Case $z \notin \mathrm{dom}(O_1 * O_2)$ in this case, $z \notin \mathrm{dom}(O_1)$ and $z \notin \mathrm{dom}(O_2)$.

Case $s_1 \models P_z$ By the induction hypothesis (for c_B), $s_2 \models P_z$.

Then, by the induction hypothesis (for c_h), $s_3 \models P_z$.

Case $s_1 \models Q_z$ By the induction hypothesis (for c_B), $s_2 \models Q_z$. Then, by the induction hypothesis (for c_h), $s_3 \models Q_z$.

Case 4

$$\frac{S; O \Vdash s_1, c_B \Uparrow e_2, s_2 \quad e_2 \neq e_k}{S; O \Vdash s_1, \texttt{try}\ c_B\ \texttt{handle}\ e_k\ \texttt{by}\ c_h \Uparrow e_2, s_2}$$

Let Σ **pairstable**, and $S; O \models \Sigma$. Hence we can apply the induction hypothesis to the smaller derivation tree for

$$S; O \Vdash s_1, c_B \Uparrow e_2, s_2$$

First, let $s_1 \models P$. Then by the induction hypothesis, $s_2 \models Q_2'$. That gives us the condition on the Hoare triple required for validity. It remains to check the conditions on signals.

Let z be a signal name with $\Sigma(z) = (P_z \triangleright Q_z)$. We make a case distinction, based on whether $z \in \mathrm{dom}(O)$ or not.

Case $z \in \mathrm{dom}(O)$ Suppose that $s_1 \models P_z$. By the induction hypothesis (for c_B), $s_2 \models P_z \vee Q_z$.

Case $z \notin \mathrm{dom}(O)$ We make a case distinction, based on whether $s_1 \models P_z$ or $s_1 \models Q_z$.

Case $s_1 \models P_z$ By the induction hypothesis (for c_B), $s_2 \models P_z$.

Case $s_1 \models Q_z$ By the induction hypothesis (for c_B), $s_2 \models Q_z$.

Finally Thus, the judgment $\Sigma \vdash \{P\}\,\texttt{try}\ c_B\ \texttt{handle}\ e_k\ \texttt{by}\ c_h\,\{Q\}\,\texttt{throws}\,\eta$ is valid for every case. □

CHAPTER 7

NESTED BINDINGS

In this chapter we explain how our operational semantics and logic handle the situation when two signals with the same name are bound in the signal context.

Both operational semantics and logic can support nested bindings of the signals with the same name, but in a slightly different manner. In operational semantics, it is straightforward. We define $[\, z \mapsto c_h \,]$ in $S\,[\, z \mapsto c_h \,]$ as an update of the signal binding S with two alternatives: if initially $z \notin \mathrm{dom}(S)$, then z is added to the S; if $z \in \mathrm{dom}(S)$, then it is overridden in a sense that z points to the new handler c_h.

However, we can restrict the signal binding only to install signals that haven't been already bound. This restriction is enforced in our logic, whenever we write $\Sigma, z : I_z \triangleright I_z$ that is the case that $z \notin \mathrm{dom}(\Sigma)$. If $z \in \mathrm{dom}(\Sigma)$, then $\Sigma, z : I_z \triangleright I_z$ is undefined. If we assume that we add a signal z into the binding with z and both signals are pointing to the different handlers, then the binding becomes inconsistent. Restricting them to point to the same handler would not be enough, as a problem of finding which z belongs to the outer scope and which to the inner one still remains.

One may say, that our logic is too restrictive when dealing with the signal binding. That is not true, as we can easily achieve a signal overriding in the logic by combination of blocking and binding commands , thanks to the notion of block in our language. Compared to the operational semantics, where we can override by a literal updating, it may look a bit cumbersome, but the logic remains clear and consistent with the restrictions on Σ and z.

$$\frac{\dfrac{[\,z \mapsto c_{h2}\,];\emptyset \Vdash s_1, c_1 \Downarrow s_2}{[\,z \mapsto c_{h1}\,];\emptyset \Vdash s_1, \mathtt{bind}\, z\, \mathtt{to}\, c_{h2}\, \mathtt{in}\, c_1 \Downarrow s_2} \qquad [\,z \mapsto c_{h1}\,];\emptyset \Vdash s_2, c_2 \Downarrow s_3}{[\,z \mapsto c_{h1}\,];\emptyset \Vdash s_1, (\mathtt{bind}\, z\, \mathtt{to}\, c_{h2}\, \mathtt{in}\, c_1); c_2 \Downarrow s_3}$$

Figure 7.1: Nested persistent signal binding

$$\frac{[\,z \mapsto c_{h2}\,](z) = c_{h2} \qquad S;\emptyset \Vdash s_1, c_{h2} \Downarrow s_2 \qquad S;\emptyset \Vdash s_2, c_1 \Downarrow s_3}{S;\,[\,z \mapsto c_{h2}\,] \Vdash s_1, c_1 \Downarrow s_3}$$

$$\frac{S;\,[\,z \mapsto c_{h2}\,] \Vdash s_1, c_1 \Downarrow s_3 \qquad S;\emptyset \Vdash s_3, c_2 \Downarrow s_4}{S;\,[\,z \mapsto c_{h2}\,] \Vdash s_1, c_1; c_2 \Downarrow s_4}$$

$$\frac{\dfrac{S;\,[\,z \mapsto c_{h2}\,] \Vdash s_1, c_1; c_2 \Downarrow s_4}{S;\,[\,z \mapsto c_{h1}\,] \Vdash s_1, \mathtt{bind/1}\, z\, \mathtt{to}\, c_{h2}\, \mathtt{in}\, c_1; c_2 \Downarrow s_4} \qquad S;\,[\,z \mapsto c_{h1}\,] \Vdash s_4, c_3 \Downarrow s_5}{S;\,[\,z \mapsto c_{h1}\,] \Vdash s_1, (\mathtt{bind/1}\, z\, \mathtt{to}\, c_{h2}\, \mathtt{in}\, c_1; c_2); c_3 \Downarrow s_5}$$

Figure 7.2: Nested one-shot signal binding

7.1 Operational Semantics Example

It is not a problem to override the persistent signal and the corresponding handler in the big-step operational semantics. Assume that the signal binding S already contains z, such that $S(z) = c_{h1}$. If deeper in a tree, a command for the signal binding of z to another handler is called, we get $S\,[\,z \mapsto c_{h2}\,]$. In both places we have $z \in \mathrm{dom}(S)$, but z is bound to the different handlers. That perfectly complies with the idea of block structures. Moreover, when a control flow returns from a brunch with the updated handler for z, the previous binding for z is restored as shown in the Figure 7.1.

With the one-shot signals we observe the following situation in Figure 7.2. Assume that the one-shot binding is split between two branches. Deeper in the first branch the one-shot signal with an existing name (let's call it z) is installed. If signal arrives after it, a newly bound handler will be executed and the signal name will be removed from the binding. When the control flow leaves a block where the last signal was bound, the previous binding is restored. This actually means that the one-shot signal binding is reinstalled.

$$\frac{S\,[\,z \mapsto c_g\,];\emptyset \Vdash s_1, c_1 \Downarrow s_2 \quad S\,[\,z \mapsto c_g\,](z) = c_g \quad S;\emptyset \Vdash s_2, c_g \Downarrow s_3}{\dfrac{S;\emptyset \Vdash s_1, \mathtt{bind}\, z\, \mathtt{to}\, c_g\, \mathtt{in}\, c_1 \Downarrow s_3}{S\,[\,z \mapsto c_h\,];\emptyset \Vdash s_1, \mathtt{block}\, z\, \mathtt{in}\, (\mathtt{bind}\, z\, \mathtt{to}\, c_g\, \mathtt{in}\, c_1) \Downarrow s_3}}$$

$$\frac{S\,[\,z \mapsto c_h\,];\emptyset \Vdash s_1, b \Downarrow s_3 \quad S\,[\,z \mapsto c_h\,](z) = c_h \quad S;\emptyset \Vdash s_3, c_h \Downarrow s_4}{S;\emptyset \Vdash s_1, \mathtt{bind}\, z\, \mathtt{to}\, c_h\, \mathtt{in}\, (\mathtt{block}\, z\, \mathtt{in}\, (\mathtt{bind}\, z\, \mathtt{to}\, c_g\, \mathtt{in}\, c_1)) \Downarrow s_4}$$

Figure 7.3: Persistent signal binding scope

$$\frac{\emptyset;\,[\,z \mapsto c_g\,] \Vdash s_1, c_1 \Downarrow s_2 \qquad [\,z \mapsto c_g\,](z) = c_g \qquad \emptyset;\emptyset \Vdash s_2, c_g \Downarrow s_3}{\dfrac{\emptyset;\emptyset \Vdash s_1, \mathtt{bind/1}\, z\, \mathtt{to}\, c_g\, \mathtt{in}\, c_1 \Downarrow s_3}{\emptyset;\,[\,z \mapsto c_h\,] \Vdash s_1, \mathtt{block}\, z\, \mathtt{in}\, (\mathtt{bind/1}\, z\, \mathtt{to}\, c_g\, \mathtt{in}\, c_1) \Downarrow s_3}}$$

$$\frac{\emptyset;\,[\,z \mapsto c_h\,] \Vdash s_1, b \Downarrow s_3 \qquad [\,z \mapsto c_h\,](z) = c_h \qquad \emptyset;\emptyset \Vdash s_3, c_h \Downarrow s_4}{\emptyset;\emptyset \Vdash s_1, \mathtt{bind/1}\, z\, \mathtt{to}\, c_h\, \mathtt{in}\, (\mathtt{block}\, z\, \mathtt{in}\, (\mathtt{bind/1}\, z\, \mathtt{to}\, c_g\, \mathtt{in}\, c_1)) \Downarrow s_4}$$

where $b = \mathtt{block}\, z\, \mathtt{in}\, (\mathtt{bind/1}\, z\, \mathtt{to}\, c_g\, \mathtt{in}\, c_1)$

Figure 7.4: One-shot signal binding scope

At first glance it might seem that the idea of one-shot signals is violated in this example, because a handler with the same name could run twice in this situation. However, from the perspective of the block structuring (scoping), it is a proper behaviour. When the binding command had been invoked, a new block was formed. Thus, the signal z was handled according to the binding of its current block. Opposite to the binding command, which creates a scope where the corresponding signal exists (bound to a handler and an element of the signal binding), the blocking command defines a scope where corresponding signal is blocked (literally excluded from the signal binding).

We may apply the uniqueness restriction for the signal binding in operational semantics. For the example in the Figure 7.3, we have to assume that initially $z \notin \mathrm{dom}(S)$ and $b = \mathtt{block}\, z\, \mathtt{in}\, (\mathtt{bind}\, z\, \mathtt{to}\, c_g\, \mathtt{in}\, c_1)$. Example with the one-shot signals is given in Figure 7.4.

$$\frac{\Sigma, z : I_z \triangleright I_z \vdash \{P_1\}\, c_1\, \{Q\} \qquad \Sigma \vdash \{I_z\}\, c_h\, \{I_z\} \qquad I_1 \textbf{ stable } I_z \triangleright I_z}{\Sigma \vdash \{P_1 \wedge I_z\}\, \texttt{bind}\, z\, \texttt{to}\, c_h\, \texttt{in}\, c_1\, \{Q \wedge I_z\}}$$

$$\frac{\Sigma \vdash \{P_1 \wedge I_z\}\, \texttt{bind}\, z\, \texttt{to}\, c_h\, \texttt{in}\, c_1\, \{Q \wedge I_z\} \qquad \Sigma \vdash \{Q \wedge I_z\}\, c_2\, \{P_3\}}{\Sigma \vdash \{P_1 \wedge I_z\}\, (\texttt{bind}\, z\, \texttt{to}\, c_h\, \texttt{in}\, c_1)\, ;\, c_2\, \{P_3\}}$$

where $\Sigma = z_1 : I_1 \triangleright I_1$

Figure 7.5: Nested signal binding

$$\frac{\Sigma, z : I_g \triangleright I_g \vdash \{P\}\, c_1\, \{Q\} \qquad \Sigma \vdash \{I_g\}\, c_g\, \{I_g\} \qquad \Sigma \textbf{ stable } I_g \triangleright I_g}{\Sigma \vdash \{P \wedge I_g\}\, \texttt{bind}\, z\, \texttt{to}\, c_g\, \texttt{in}\, c_1\, \{Q \wedge I_g\} \qquad P \wedge I_g \textbf{ stable } I_h \triangleright I_h \qquad Q \wedge I_g \textbf{ stable } I_h \triangleright I_h}{\Sigma, z : I_h \triangleright I_h \vdash \{P \wedge I_g\}\, \texttt{block}\, z\, \texttt{in}\, (\texttt{bind}\, z\, \texttt{to}\, c_g\, \texttt{in}\, c_1)\, \{Q \wedge I_g\}}$$

$$\frac{\Sigma, z : I_h \triangleright I_h \vdash \{P \wedge I_g\}\, b\, \{Q \wedge I_g\} \qquad \Sigma \vdash \{I_h\}\, c_h\, \{I_h\} \quad \Sigma \textbf{ stable } I_h \triangleright I_h}{\Sigma \vdash \{P \wedge I_g \wedge I_h\}\, \texttt{bind}\, z\, \texttt{to}\, c_h\, \texttt{in}\, b\, \{Q \wedge I_g \wedge I_h\}}$$

where $b = (\texttt{block}\, z\, \texttt{in}\, (\texttt{bind}\, z\, \texttt{to}\, c_g\, \texttt{in}\, c_1))$

Figure 7.6: Signal binding and blocking result in overriding

7.2 Logic Example

Whenever we apply a binding rule for the signal z, it is always restricted to the case that $z \notin \text{dom}(\Sigma)$. Therefore, in the example in Figure 7.5, $z \neq z_1$. In other words $z \notin \text{dom}(z_1 : I_1 \triangleright I_1)$.

In the example in Figure 7.6, an overriding is achieved by the combination of signals binding and blocking. Assume that initially $z \notin \text{dom}(\Sigma)$ and that all required stability assumptions hold.

CHAPTER 8

INTRODUCTION TO REENTRANCY

In this chapter we summarise the ideas behind reentrancy from various domains and provide a glossary of the reentrancy related terms. One of the main contributions of this chapter is addressing relations between reentrancy and other notions such as interrupted signal handlers, thread safety, stability, locks and exception handlers.

Reentrancy is an important notion, but a definition for it, and in some sense attitude, is highly domain dependent. For example, in Object Oriented Programming (OOP) domain a lot of work has been done to develop various reentrancy detection systems, and in some approaches authors go even further and propose the reentrancy elimination techniques. In approaches towards verification, the reentrant calls could be classified into consistent and inconsistent calls, where an absence of the latter ones is desired. In the domain of the hardware interrupts, reentrancy has also negative connotation, therefore there are proposed techniques to prevent reentrancy of interrupts. At the same time, in multithreading Operating Systems (OS) domain, reentrancy is tightly connected with the notion of thread safety. Moreover, even in the single-threaded environment a reentrant code (usually functions or procedures) is more preferable to nonreentrant one, as even in the single-threaded environments a piece of code may not be atomic.

Reentrancy might have different meaning in every domain, and could be related to various concepts such as interrupts, thread-safety and atomicity. One of the goals in our work is to summarise ideas of reentrancy from various domains, and try to propose some basic and general definition. Thus, we had to check the literature in this domain first.

8.1 Reentrancy in OOP

In OOP, the reentrancy of a call on the object could be explained in the following way. Assume an object has a few methods. When one of the methods called, a code inside that method non-directly or directly calls another (or the same) method of the same object. The second method call in these circumstances is called a recursive call. This definition does not say anything about the number of threads, but still, it is quite important to know how it is interrupted or where the second method comes from, as the interrupting method call could come from the same thread, or from another concurrent one.

8.1.1 Short Literature Observation

The reentrancy could be defined via graphs [68]. There is a notion of object race, which is not an equivalence to a data race, but it is its prerequisite. A part of the race detection analysis is actually a reentrancy analysis (another part is a lock analysis). The relation between object and object calls could be represented via graphs. Therefore, when we talk about reentrancy or nonreentrancy in this approach, technically we mean reentrancy/nonreentrancy of edges of the object graph. Nonreentrancy is important for the object race analysis, as it indicates that the execution of events in two threads does not yield to the object race. Idea it that if an edge is not reentrant, then two threads must not execute events on that edge.

Reentrancy in OOP could be defined without graphs, but still points-to graphs are used for the reentrancy analysis [29]. One may say that the author performs reasoning in terms of the modules, as concepts of combination and inclusion of the modules is one of the main ideas. Therefore, for better understand of this work one should first understand the notion of modules.

When the same class or block of code is included to more than one program, then there is possibility of a reentrant call. The notion of a stack is used in analysis and definitions. An object is called *active* in a particular state, if the stack in that state

contains a frame for this object. To be precise, a frame, where that object is a receiver of the method call. In other words, an object is called active in a state if there is an active method call on it. According to [29], an object is consistent, if its invariant holds, and inconsistent otherwise. A method call is reentrant on an object, if the call stack already contains a method invocation on that object. Author calls the method invocations reentrant on an object, whenever any method invocation called while another invocation is still registered in the stack. So, it shouldn't be the same method, to call the method invocation reentrant. Thus, reentrancy is defined with respect to the notion of the object, but not the function/procedure. Identifying the reentrancy of calls is important for the program validation and verification.

8.1.2 Example Scenarios

- Assume that the method of an object has access to the class global variable. One can make an invariant on it. Code inside the method non-directly, avoiding plain recursion, calls the same method on the same object (on itself). Thus, the second method call is classified as a recursive call.

- An object has a few methods. When one of the methods called, a code inside that method non-directly calls another method of the same object. Thus, the second method call is also classified as a recursive call.

8.2 OOP and Multithreading

In OOP with multithreading, the focus moves from the interaction of objects and method calls inside of the single thread, to interaction between two threads via method calls on the same object simultaneously. Reentrancy could be addressed as a property of the program executed in the multithreading environment. Moreover, some authors propose an approach, where nonreentrant Java programs are converted into reentrant via replacing the *global state* with the *thread-local state* and performing each execution in a new thread

[104]. Actually, authors deal with the reentrancy of external calls to API methods of a program. That is not the same as recursive method calls internal to the program. One may conclude that authors do not focus on a low-level (interrupts or self modifying code), but focus on the user-level program running. It should be noted that the definition of the low-level and up-level are highly domain dependent. A notion of reentrancy also has a tight relation to the multicores. If the program is reentrant, as defined by [104], then it can be run in parallel on multicore without concurrency control.

8.3 Multithreading as Part of the OS

In this section we focus on a definition of the reentrant procedure and Operating Systems with multiple users support. One of the definitions of reentrancy via a definition of the reentrant procedures in operating systems is as follows. "A reentrant procedure is one in which a single copy of the program code can be shared by multiple users during the same period of time." [90]. There are few obligations to achieve procedure reentrancy. The local data for every user should be stored separately and the program code shouldn't modify itself. The idea of the reentrant procedure is that it can be safely interrupted by another program which calls the same procedure. Safety in this context means that both procedures interrupted and interrupting will produce correct and reliable result as if they were executed sequentially. According to the author [90], a reentrant code allows an efficient use of memory, as only a single copy of the code is loaded and kept in the main memory, while many applications can call that code or procedure.

In multithreaded Operating Systems, a concurrent thread scheduler manages threads in the kernel [37]. It is called concurrent, because there may be multiple threads in the system running the scheduler code at the same time. A thread scheduling itself might be implemented as a separate thread that runs concurrently with other threads in the kernel. One shouldn't forget about the context switch between the threads. One thread might be preempted while executing a routine, then another thread will call the same routine

or start using the same resource (e.g.: queue). When the interrupts are introduced, the non-reentrancy of the interrupts is ensured via the special flag or register. The interrupt service routines (ISR) could be executed in the separate threads. The context of the previously running thread should be stored before the ISR and restored afterwards.

A reentrancy of threads is related to the thread safety, as when one thread is interrupted, via an explicit interrupt or a thread manager, and another thread starts running, it may result in a situation when the block of code runs inside of itself (interrupted *copy*). If the thread is interrupted actually by the interrupt, then ISR should run. In some implementations it runs in a separate thread, but still a non-reentrancy of the ISR is enforced, and no other threads' context is loaded until ISR returns.

8.3.1 Reentrant Kernels

According to [11], "All Unix kernels are reentrant". Therefore, several processes may be executing in Kernel Mode simultaneously. Nonreentrancy of the kernel means that a process can only be suspended while it is in User mode. To achieve a reentrant kernel only the reentrant functions should be used, or alternatively the locking mechanism should be used to ensure that only one process executes a nonreentrant function at a time.

8.4 Event-Driven Programming

Literally, an event is a common name for both hardware interrupts and signals, which are also known as software interrupts. In the domain of event-driven programs, exists an event enabling/disabling mechanism that might be considered as an alternative to the blocking mechanism designed in our work.

According to the authors who work in the domain of event driven programming, the "events have been viewed by many researchers as alternative to threads" [38]. That is rather challenging statement, and studying reentrancy in the domain of the event driven programming could be one of the directions for future work.

8.5 Reentrant Locks

The idea of the reentrant locks is that they can be acquired multiple times by the same thread without blocking [104]. The reentrant locks may lead to the data races and deadlocks. Nevertheless, the correct usage of them facilitates the concurrent programming. For example, keeping track of locks that are used by any particular thread facilitates the verification that resources, which are associated with the locks, remain available even after multiple lock acquisition by the same thread [39]

8.6 Objective C

According to the Apple Concurrency Programming Guide, a **reentrant code** is the code that can be started on a new thread safely while it is already running on another thread" [49]. An OS X Glossary [48] covers the notion of a reentrant functions in the following way. "Said of code that can process multiple interleaved requests for service nearly simultaneously. For example, a **reentrant function** can begin responding to one call, be interrupted by other calls, and complete them all with the same results as if the function had received and executed each call serially."

The Objective C inherits signal handling mechanism from the BSD. However, an alternative mechanism on the higher level is proposed. The Grand Central Dispatch (GCD) mechanism is a proposed alternative to the threads. GCD also provides a handling mechanism for events (signals) on a higher level. By our classification, only the one-shot signals, which are implemented via queue, were adopted, what simplifies and in some cases eliminates the reentrancy problems.

8.7 Glossary of the Reentrancy Related Terms

This section combines various definitions of reentrancy and reentrant functions, method, procedures, routines, programs, locks and etc.

reentrant code is the code that can be started on a new thread safely while it is already running on another thread [49].

reentrant (code) can process multiple interleaved requests for service nearly simultaneously. For example, a reentrant function can begin responding to one call, be interrupted by other calls, and complete them all with the same results as if the function had received and executed each call serially [48].

reentrant functions One way to provide reentrancy is to write functions so that they modify only local variables and do not alter global data structures. Such functions are called reentrant functions [11].

reentrant (function) A function is reentrant when it is possible for it to be called at the same time by more than one thread. This implies that any global state be protected by mutexes. Note that this term is not used uniformly and is sometimes used to mean either recursive or signal-safe. These three issues are orthogonal. [61, p. 367]

reentrant function A function is said to be reentrant if it can safely be simultaneously executed by multiple threads of execution in the same process. In this context, 'safe' means that the function achieves its expected result, regardless of the state of execution of any other thread of execution [56, p. 423].

The SUSv3 definition of a reentrant function is one "whose effect, when called by two or more threads, is guaranteed to be as if the threads each executed the function one after the other in an undefined order, even if the actual execution is interleaved."

reentrant function A reentrant function is one that can be used by more than one task concurrently without fear of data corruption. A reentrant function can be interrupted at any time and resumed at a later time without loss of data. Reentrant functions either use local variables or protect their data when global variables are used [54].

reentrant function A reentrant function can also be called simultaneously from multiple threads, but only if each invocation uses its own data [80].

reentrant function Does not hold static data over successive calls. Does not return a pointer to static data; all data is provided by the caller of the function. Uses local data or ensures protection of global data by making a local copy of it. Must not call any non-reentrant functions [54].

non-reentrant function A non-reentrant function is one that cannot be shared by more than one task unless mutual exclusion to the function is ensured either by using a semaphore or by disabling interrupts during critical sections of code [54].

non-reentrant function a function that have static variables or that modify global variables or resources without any sort of locking mechanisms [27].

reentrant kernel But a reentrant kernel is not limited only to such reentrant functions (although that is how some real-time kernels are implemented). Instead, the kernel can include nonreentrant functions and use locking mechanisms to ensure that only one process can execute a nonreentrant function at a time.

All Unix kernels are reentrant [11].

reentrant program Let an execution of a program P be any external invocation of P, e.g., running P's main method or invoking a public API method from P. A program P is reentrant iff for any two executions e_i and e_j of P such that e_i and e_j have no mutable shared inputs (...), the results of e_i and e_j are unaffected by how the executions are ordered, including parallel interleavings [104].

reentrant lock The reentrant locks can be acquired multiple times by the same thread without blocking [104].

The reentrant locks may lead to the data races and deadlocks. However, the correct usage of them facilitates the concurrent programming [39].

reentrant program A program is reentrant if distinct executions of the program on distinct inputs cannot affect each other, whether run sequentially or concurrently [104]

reentrant procedure A useful concept, particularly in a system that supports multiple users at the same time, is that of the reentrant procedure. A reentrant procedure is the one in which a single copy of the program code can be shared by multiple users during the same period of time. Reentrancy has two key aspects: the program code cannot modify itself and the local data for each user must be stored separately. A reentrant procedure can be interrupted and called by an interrupting program and still execute correctly upon return to the procedure. In a shared system, reentrancy allows more efficient use of main memory: one copy of the program code is kept in main memory, but more than one application can call the procedure [90].

reentrancy Reentrancy refers to a function's capability to work correctly, even when it's interrupted by another running thread that calls the same function. That is, a function is reentrant if multiple instances of the same function can run in the same address space concurrently without creating the potential for inconsistent states [27].

reentrant event Blocking, Preemption, Nesting, and Reentrancy. Contrarily to the threads, event handlers cannot block so they run to completion except when preempted by another event handler. Events have an asymmetric preemption relation with the non-event code: event handlers can preempt non-event code but not the contrary. Events are nested when they preempt each other. Nesting events are used to allow time-sensitive events to be handled with low latency. An event is said to be reentrant when it directly or indirectly preempts itself [38].

reentrant method call A method call is re-entrant on an object o, if the call stack already contains a method invocation of the method on object o. A re-entrant call furthermore is inconsistent, if the object o is not consistent at the re-entrant call site. An object o is active in a particular execution state, if the state contains a stack

frame where o is the receiver object of the method call. An object is consistent, if its invariant holds [29].

async-signal safe function a function is async-signal safe if it can be safely called from within a signal handler [87].

A function may be async-signal safe in one implementation, and not async-signal safe in others. Thus, the async-signal safety is highly implementation dependent.

asynchronous-safe function (asynchronous-safe sometimes referred to as async-safe, or signal-safe) An asynchronous-safe function is a function that can safely and correctly run even if it is interrupted by an asynchronous event, such as a signal handler or interrupting thread. An asynchronous-safe function is by definition reentrant, but has the additional property of correctly dealing with signal interruptions. Generally speaking, all signal handlers need to be asynchronous-safe. [27]

thread-safe function A thread-safe function can be called simultaneously from multiple threads, even when the invocations use shared data, because all references to the shared data are serialized [80].

thread safe function A function is said to be thread-safe if it can safely be invoked by multiple threads at the same time; put conversely, if a function is not thread-safe, then we can't call it from one thread while it is being executed in another thread [56, p. 655].

. . . a function is not thread-safe: it employs global or static variables that are shared by all threads [56, p. 656].

reentrant service A service that is safe to call from multiple threads in parallel. If a service is reentrant, there is no burden placed on calling routines to serialize their access or take other explicit precautions. See also thread-serial service, and thread-synchronous service. IBM Glossary: `http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp?topic=/com.ibm.aix.glossary/doc/glossary/glossary02.htm`

8.8 Towards New Definition and Glossary

One may divide the concept of reentrancy in two parts: literal re-entering of the function or code to itself and re-accessing of the resource already held by the function or code. Actually, there is no problem if a function is interrupted by itself or in other words re-enters itself, unless both instances share the same global variable. Thus, an explanation of reentrancy in OOP domain looks more precise and detailed. If reentrancy happens, it does not mean that everything breaks without further investigation. Therefore, a classification of reentrancy into *good* (safe) and *bad* (unsafe) sounds reasonable.

We try to provide a glossary for the terms that are used by programmers and rationalize them in terms of theory and language we have designed. First of all, inconsistency is not a property of the state s, but a relation between a state and a resource invariant. A state is consistent if it satisfies the invariant, $s \models I$. Second, reentrancy is a special case of indirect recursion that is nondeterministic via signal handlers. And third, async-safety and reentrancy are related but different; async-safety could be achieved by enforcing reentrancy or blocking of the signals.

8.9 Comparison of the Reentrancy in OOP and Procedural Paradigms

Object in OOP is a mutable state with a set of methods that can modify the state (Figure 8.1). When the issue of reentrancy is raised, usually, the reentrancy of method calls is meant. The reentrancy of method call occurs, when from an active method call on object another method of that object is called. That could be a direct or non direct call. One may ask, why is it called reentrant even if not the same method is actually called? Such thought are influenced by the notion of reentrancy from the procedural languages, where for reentrancy to happen, a function should re-enter itself (Figure 8.2). It should be noted that reentrancy is not a property of a function or code, it is a relation

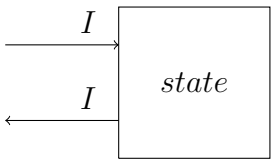

Figure 8.1: Method calls and returns

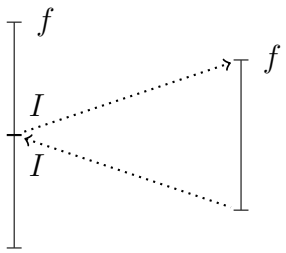

Figure 8.2: Function reenters function

and resource sharing influences the reentrancy.

Thus, in procedural language, when we say a reentrant function it means that concurrent execution of that function will not corrupt the resource for any instance of that function [56, 11, 27]. That could be expressed using invariants. A reentrant function does not invalidate an invariant initialised for some shared resource. It could be noted that invariant could be violated inside of the function, but should be restored before return. Same idea could be used to understand the reentrancy of method calls in OOP. An invariant should be initialised before the method call, and should be expected to remain valid on return from the method call. We should think of an object as of a resource, and not focusing on the inner structure of it. Then it becomes clear, why for the reentrant method call it is not limited for calling itself only. When a method call runs on an object, and then directly or indirectly calls another method of the same object, a second method call re-enters the object.

Despite this huge similarity, there are some differences as well. In procedural languages, a function could be reentrant or not reentrant. That is already enough to understand that if a reentrant function is interrupted via signal handler by another call of the same function, everything remains in a consistent state. For sure, for the sake of this example, it is assumed that the signal handler does not modify any global state. We can

assume that the signal handler consists of one reentrant function only. In OOP, a fact of the reentrant method call on an object does not infer that the object remains in a consistent state or not. Therefore, the reentrant method calls are divided into consistent and inconsistent reentrant calls [29]. Shortly, if a reentrant method call invalidates an invariant, it is called inconsistent.

8.9.1 Invariants

Instantiation of invariant for method call on object looks easier comparing to the function or procedure. Programmer can see the inner structure of the object (its fields - its state), if the source code is available. For the procedures, it is even not obvious, if the reentrancy exists and at what level does it happen; thus, instantiation of a proper invariant may need much more efforts. Instance of an object encapsulates a state inside, and reentrancy (re-enter of functions or re-access by functions) is defined with respect to the object's state. In procedural languages, reentrancy of a function code itself does not represent much. More importantly, if deeper in a chain of function calls, some global variable (resource) is used.

8.9.2 Reentrant Call From an Inconsistent State

A consistency of the state is a relation between an invariant and the state. If the invariant holds in a particular state, then it is called a consistent state, and vice versa. In terms of OOP, when the reentrancy of method calls happens in the inconsistent state, then the reentrancy is inconsistent (bad).

One may ask the following question: "How is the notion of consistent reentrancy related to the interrupter and interrupted method or function?" Save-restore is a known technique to achieve reentrancy in some cases. This technique is straightforward and details are as follows: when interrupted function resumes its execution, all linked resources should remain in the same state, or invariant that was instantiated before interruption should hold. Use of invariants, for example some variable should remain positive, gives

us more freedom and flexibility.

Before we call a method (function) we instantiate an invariant, that holds just before the call. That invariant could be violated in-between of call and return, but it should be valid on return. Assume that the method is interrupted when invariant does not hold (inconsistent state), that is the case of inconsistent reentrancy. Assume that the interrupter restores all linked resources on return, thus for the interrupted method that was just a pause. However, interrupter started its execution in the inconsistent state. So it did not corrupted the interrupted method call, but it used inconsistent data as input for its own execution; thus, whatever result was produced, it is not reliable.

TOCTTOU (time of check to time of use) is a similar class of problems that should be described. It is a next example for the problematic situation described above. In TOCTTOU, the interrupter in some cases may be considered as unaffected participant. Where the interrupted procedure is no longer reliable or consistent after it resumes.

An artificial example of badly encoded database transactions is given below. Assume we have a function (method) that encapsulates a balance check and if there is enough money withdraws them from the account. First instance of a function starts its execution, performs a balance check, and after ensuring that there is enough money wants to call money withdraw sub-procedure. However, it is interrupted and another instance of this function checks balance and withdraws money. Then the first instance of a function resumes its execution and performs another withdraw, potentially making a negative balance.

Noteworthy, that the save-restore technique doesn't achieve reentrancy for both callers in some circumstances. Thus, this technique is just a mitigation, as it potentially can achieve a reentrancy for one participant only.

8.10 Reentrant and Interrupted Handlers

From the implementation point of view, the signal handler is just a function. Being a part of the OS architecture, imposes some restrictions and limitations. Creating a reentrant signal handler might be hard. It should be noted, that reentrancy here has a meaning that another instance of the same handler (function) re-enters itself. Therefore, in most Unix based system a signal that caused a signal handler to run is implicitly (automatically) blocked inside of the handler. Thus, reentrancy of a signal handler with respect to itself is avoided.

That may create a false feeling of a safety: as reentrancy of the same handler is forbidden, the signal handler need not to be reentrant [11]. The signal handler interruption by other signals is not forbidden. Technically (according to the current notation in the literature), such interruption is not a case of reentrancy of two distinct handlers. However, directly or indirectly (via another function call) two distinct signal handlers may share the same resource. For example, `errno` global variable. Thus, two distinct signal handlers actually re-enter the same state.

Keeping this renewed notion of reentrancy in mind, we may conclude that for every non-blocked signal its handler should be reentrant with respect to all other non-blocked signals' handlers. Means that the handlers should not share directly or indirectly any resources with each other. In practice, it might be very hard to achieve; in other words just unfeasible. Therefore, one may try to adopt invariants to address reentrancy. And the definition of the reentrancy could be modified a bit (inspired by reentrancy in OOP). We may break a tight connection between reentrancy implies safety (current definition in procedural languages). Then, we will divide reentrancy in safe and unsafe. e.g.: two functions re-enter a state, thus, it is a case of reentrancy. Is it safe or unsafe depends on what they do with that state: modify or just read?

Here come the invariants. Let's consider a simple scenario. When the handler (function) is interrupted, an invariant with respect to the shared variable should be instanti-

ated. Another signal handler runs, and on its return that invariant is validated. If the invariant holds, that is the case of a safe reentrancy, otherwise unsafe.

8.11 Reentrancy and Thread-Safety

Discussion in this section is inspired by "Writing Reentrant and Thread-Safe Code" by IBM http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp?topic=/com.ibm.aix.genprogc/doc/

Reentrance and thread safety are both related to the way that functions handle resources. Reentrance and thread safety are separate concepts: a function can be either reentrant, thread-safe, both, or neither [54].

A thread-safe function protects shared resources from concurrent access by locks. Thread safety concerns only the implementation of a function and does not affect its external interface. There are several methods to make a function thread safe. First of all, one may associate a lock with a function or group of functions, which operate with the same resources. Thus, when the function is called, a lock will be acquired, and on function return the lock will be released. That is very straightforward approach that has its limitations such as only one thread at a time may use the function. Therefore, in most critical situation (if that function is a big chunk of the program) all threads will operate almost sequentially, what definitely contradicts the idea of multithreaded and parallel program execution. More advance method is to associate the lock not with the whole function, but only with its critical sections that operate with some shared variable. Thus, the lock will be obtained and released only during work in the critical sections. Consequently, most of the time parallel threads would not wait for each other, and wait only if more than one thread are in critical section and requested the lock. "Thread-specific data" or "Thread-local storage" are another methods that makes a function thread-safe without changing its interface. The rough idea behind these techniques is to use only memory that is local to the thread. Surprisingly, that includes even static and global memory, but which is made local (to be more precise, part of it made local)

for the particular thread. For more detail on thread-specific data please see "PThreads Primer"[61] by Lewis et. al.

There are functions that by definition are nonreentrant, because they operate with (or return the) pointers to statically allocated storage, or just use static storage to keep some information between multiple function calls. Finally, the most efficient way of making a function thread-safe is to make it reentrant, as reentrant functions do not require locking mechanism to achieve thread safety [56].

According to the documentation of a Qt project [80], a **thread-safe function** can be called from multiple threads simultaneously, even when the shared data is used by the invocations, because all references to the shared data are serialized, where a **reentrant function** can be called from multiple threads simultaneously, only if no data is shared among invocations.

Comparing Qt to the IBM documentation, thread safety and reentrancy is not tightly related, as extra conditions should be explicitly satisfied. A function could be thread-safe but not reentrant. A reentrant function could be thread safe with some extra conditions such as each invocation uses its own data. Where according to the IBM documentation, for a function to be reentrant implies that it is thread-safe as well.

8.12 Relation Between Stability and Reentrancy

In this section we try to answer the following question. Do stability assumptions implicitly guarantee reentrancy?

Σ **pairstable** means that any signal from the $\mathrm{dom}(\Sigma)$ does not invalidate invariants, preconditions or postconditions of other signals from the $\mathrm{dom}(\Sigma)$. Thus, any signal handler may be interrupted by another one in a safe way, for all z such that $z \in \mathrm{dom}(\Sigma)$. Therefore, we may conclude that all signal handlers ($\forall z$. s.t. $z \in \mathrm{dom}(\Sigma)$), are reentrant with respect to each other and invariants they hold. Please note, that the Definition of Σ **pairstable** does not cover the case when the signal handler for z is interrupted

by itself. However, two identical signal handlers could be bound to two different signal names; thus, when one of them is interrupted by another one and Σ **pairstable**, then that handler is indirectly reentrant to itself. Therefore, when we assume Σ **pairstable** in our logic, we restrict all signals in dom(Σ) to be reentrant with respect to each other.

We are talking about signal handlers' reentrancy, and our invariant is instantiated at the beginning of the handler and validated on its return. So, it is a bit more abstract level than functions, procedures or their inner structure. However, technically, signal handlers remain being plain functions.

Two signal handlers may call the same library function that is known to be non-reentrant. They may use a locking mechanism to call that function, but the risk of a deadlock becomes very high if main program also calls that non-reentrant function. Blocking all of the signals, while calling non-reentrant function, is an alternative, but it may impair the performance. Saving the affected resources before and restoring them after the non-reentrant function call, could be a solution in some cases (e.g. reporting error with *errno*), but definitely it is not a silver bullet.

Achieving reentrancy of a function with respect to itself is not easy, and it becomes even harder if that function is a signal handler, as locks are dangerous.

8.13 Locks, Reentrancy and Signal Handlers

It should be clarified, what kind of locking mechanism we keep in mind while addressing signal handling. Locks in a multithread environment and locks in a single thread have subtle difference. The latter kind is extremely dangerous for usage inside of the signal handlers.

Signals arrive nondeterministically, so the signal handlers might be called at any time of the program run, no matter whether a thread holds a lock or not. Assume a single threaded program with enabled signals handling mechanism. Both the program and the signal handler operate with the same resource lock. If a program (thread) acquires a

lock and then a signal arrives, which triggers the signal handler, that will result in a deadlock. The signal handler cannot proceed, as it is waiting for a lock that should be released by the program, but the program cannot release the lock and proceed either, as it is interrupted by the signal handler and wait until it returns.

Thus, calling locks from a signal handler is not recommended. However, the following workaround could be used. A signal handler may just spawn a new thread, which contains all the required logic including lock acquisition, and immediately return. So the main thread of the program would not be blocked for a long time and could continue as soon as the signal handler returns. As a result, we get two threads that will compete for the resource in parallel, but the danger of the deadlock is eliminated. If the first thread acquires the lock, the second thread will block on attempt to get the same lock, but when the first thread returns the lock, the second thread continues.

Another example comes from the Multithreaded Programming Guide by Oracle `http://docs.oracle.com/cd/E19253-01/816-5137/gen-26/index.html`, which describes a similarity between thread safety and asynchronous-signal safety. The problems of asynchronous-signal safety arises when the operation of a signal handler interfere with the executing operation that is being interrupted. Assume that a program called a function `printf()` that has not completed and returned yet. Then, due to a received signal, a signal handler is called that also calls the function `printf()`. As a result, with high probability an output of two simultaneously calls of the function `printf()` would be intertwined. As we learned from the previous example of this section, such problem cannot be mitigated with locks (an example of synchronisation primitives), as syntonisation between the signal handler and the corresponding interrupted thread with locks will quickly result into a deadlock. Thus, to avoid interference between the thread and the signal handler, one should not use synchronisation primitives, but, for example, block signals with signal mask or call only asynchronous-signal safe functions form the signal handlers.

To avoid interference between the handler and the operation, ensure that the situation

never arises. Perhaps you can mask off signals at critical moments, or invoke only Async-Signal-Safe operations from inside signal handlers.

8.14 Signal and Exception Handlers

In the current real-life implementations, exceptions have higher priority over signals. However, in some cases exceptions could be implemented via signals. That shifts a focus of the discussion to another question: priority within signals. In our language, we consider exceptions as a separate construct that resembles signals but nevertheless different. The priority of exceptions over signals in real-life implementations is influenced by the fact that exceptions usually indicate that some error happened (e.g.: memory corruption) and further calculations with high probability are no longer reliable or just impossible due to hardware fault. Our semantics and logic can support both models (priority of the signals or priority of the exceptions) after minor modifications, but we stick to the convention accepted in the real-life implementations. Also, both exception and signal handlers are represented via functions or commands in our language

8.15 Summary and Discussion

To avoid clashes, during executions of the non-reentrant code, all interrupts (signals) should be blocked. However, non-interruptible code is dangerous in a way that if it loops, it can't be interrupted. Therefore, the non-interruptible signal handlers should not contain process sleep, wait or freeze kind of commands.

For a function to be reentrant, it shouldn't call nonreentrant functions. For a kernel to be reentrant, all processes should deal with non-reentrant functions via lock, or use only reentrant functions. For a handler to be reentrant, it shouldn't contain nonreentrant code or calls to the nonreentrant functions. So, can we use locks in the handlers to deal with the nonreentrant code, analogous to the approach used for kernels? At least in the Linux, we cannot reliably call mutex locks and unlocks from the signal handlers. Assume

that the signal arrives while a thread holds the lock. If the signal handler tries to acquire the same lock, it fails, as the lock is held by the thread. Thus, they result in a deadlock, as the signal handler will wait for a lock, while the thread will wait until the handler returns to proceed and to free the lock.

A few scenarios of the signal handling should be considered. In first scenario, all signals are blocked in the signal handler, thus they are uninterruptible. In the second, only the same signal is blocked, thus it cannot interrupt itself, but it is interruptible by other signal handlers. And in the third scenario, no signals are blocked. According to the literature (e.g.:[11]), the signal handlers in the first and the second scenarios need not to be reentrant, where in the last one they should be reentrant. Some authors [27] mention the danger of the second scenario though. Also, the GNU libc documentation [34] warns that if not all signals blocked then they shouldn't call nonreentrant functions or modify global data. What if in the second scenario, two handlers call the same nonreentrant code? Thus, when one handler interrupts another one (assume both handlers are different, but call the same nonreentrant function), the same nonreentrant code may re-enter itself.

CHAPTER 9

REENTRANCY LINEAR TYPE SYSTEM

In this chapter we extend our language with local variables and functions to study reentrancy. Then we define a Reentrancy Linear Type System and extended logic which is used to address programs with reentrant and nonreentrant functions. First of all, the extended logic with RLTS could be used to verify whether a function is reentrant or not. And the key contribution is that it could be used to verify a program whether it is safe to use nonreentrant functions in it or not.

There are two strategies one can adopt while designing a language. One may design a semantics that will prevent unsafe situations; thus, following this approach a safe language could be designed. Alternatively, a semantics may describe an unsafe language, where a bad coding practice will result in a critical situations such that memory corruptions, faults or errors. This is rather philosophic question, to decide at what level the safety of a language should be achieved. Should the language try to "defend" itself, or should the programmer follow the acceptable code practices, so as an output one will get a program that wouldn't fail during the runtime.

We pick the second approach, so our operational semantics, despite being idealized up to some limit, reflects the real-life implementations. Therefore, we aim to contribute a logic (type system), using which one may statically analyse the code and predict that the program will run as expected or finish in an `error state`. However, we also designed several pieces of the operational semantics that try to prevent faulty situations. We used it for better understanding of the interaction between signals and functions; and as a first

step towards studying reentrancy. One can use this semantics to design a safe language with functions, exceptions and signals.

A question of reentrancy is not limited to the persistent signals only, as the one-shot signals interfere with a code of the main program. Thus, the reentrancy of a function may occur when the signal handler or the main program code is interrupted by a signal handler. It could be the same or completely different signal, as various signal handlers may call the same nonreentrant functions. We call a signal handler reentrant if it calls reentrant functions only. We call a signal handler nonreentrant, if it calls at least one nonreentrant function.

The sharing that is problematic concerns the resource that the non-reentrant functions access, not the functions themselves. For example, `malloc` and `free` are different functions accessing the same resource, the free list. Non-async-safe functions are a linearly used resource in one-shot signal handlers. The specification of library functions does not necessarily tell us what the shared resource is that a non-safe handler accesses. The specification tells us whether the function is async safe. Even if signal handlers are not interruptible, a non-safe function could have been called by the main program, interrupted by a signal handler, and then called by the handler again. Async-safety and reentrancy are related but different. Async-safety could be achieved if all used functions are reentrant or all signals are blocked.

It should be noted that even async-signal safe functions deal with `errno` in non-reentrant way [56]. Thus, one can conclude that async-signal safe functions are not completely safe if error handling mechanism is considered as part of that functions.

9.1 Language with Functions and Local Variables

A reentrant-safe self-interruption of signal handlers imposes strong restrictions on what can be done inside of a handler. Commands modify a global state; thus, for a code to make something useful in such restrictive conditions, the local state should be introduced.

Therefore, we introduce functions in our language, as they have local variables and the same time have access to the global variables.

Local variables declaration is of the form

$$\texttt{local}\ y_1,\ \ldots\ , y_n\ \texttt{in}\ c$$

Here $y_1, \ldots, y_n$ are the local variables and c is the command, in scope of which local variables are defined. Each function declaration is of the form

$$\texttt{fun}\ f\ (\ ) = c_f\ \texttt{in}\ c_B$$

Here f is the name of a function, c_f is the body of the function, and c_B is the command, in scope of which the function f is defined. Finally, the function bindings are stored in a function context F that is of the form

$$[\, f_1 \mapsto c_1\,], \ldots, [\, f_n \mapsto c_n\,]$$

and the general rule of the function binding and the function context extension is of the following form

$$\frac{F\,[\, f_j \mapsto c_j\,] \Vdash s_1, c_B \Downarrow s_2}{F \Vdash s_1, \texttt{fun}\ f_1\ (\ ) = c_1\ \texttt{in}\ \ \ldots\ f_n\ (\ ) = c_n\ \texttt{in}\ c_B \Downarrow s_2}$$

where j has the next range $1 \leq j \leq n$.

We define a big-step semantics for a language with function calls and local variables in Definition 9.1.1. The big-step rules for operational semantics are given in Figure 9.2, Figure 9.3 and Figure 9.4.

Definition 9.1.1 The syntax of the language is given in Figure 9.1.

Commands

c	::=	$\texttt{while}\, E_B\, \texttt{do}\, c$	(while construct)
	|	$\texttt{if}\, E_B\, \texttt{then}\, c_1\, \texttt{else}\, c_2$	(if else construct)
	|	a	(Atomic command)
	|	$x := E$	(Assignment)
	|	$x{+}{+}$	(Increment)
	|	$x{-}{-}$	(Decrement)
	|	$c_1; c_2$	(Sequential composition)
	|	$\texttt{local}\, y_1, \ldots, y_n\, \texttt{in}\, c$	(command with local variables)
	|	$\texttt{fun}\, f\, (\,) = c_f\, \texttt{in}\, c_B$	(Function declaration)
	|	$f(\,);$	(Function call)
	|	$\texttt{fun}\, f_1\, (\,) = c_f\, \texttt{in}\, \ldots$	
		$\quad f_n\, (\,) = c_n\, \texttt{in}\, c_B$	(Program run)
	|	$\texttt{throw}\, e$	(Exception throwing)
	|	$\texttt{try}\, c_B\, \texttt{handle}\, e\, \texttt{by}\, c_h$	(Exception handling)
	|	$\texttt{block}\, z\, \texttt{in}\, c$	(Blocking of the signals)
	|	$\texttt{bind}\, z\, \texttt{to}\, f_z\, \texttt{in}\, c_B$	(Binding of the persistent sign handler)
	|	$\texttt{bind/1}\, z\, \texttt{to}\, f_z\, \texttt{in}\, c_B$	(Binding of the one-shot sign handler)

Expressions

E	::=	x	(Variables)
	|	E_I	(Integer expressions)
	|	E_B	(Boolean expressions)
E_I	::=	n	(Integers)
	|	$E_I + E_I \mid E_I - E_I \mid \ldots$	(Basic arithmetic operations)
E_B	::=	$\texttt{true} \mid \texttt{false}$	(Booleans)
	|	$E_I \leq E_I \mid E_I > E_I \mid \ldots$	(Basic arithmetic operations)

Figure 9.1: The syntax of the language

$$\frac{\Vdash s_1, E_B \Downarrow \texttt{true} \quad S; O_1; F \Vdash s_1, c \Downarrow s_2 \quad S; O_2; F \Vdash s_2, \texttt{while}\, E_B\, \texttt{do}\, c \Downarrow s_3}{S; O_1 * O_2; F \Vdash s_1, \texttt{while}\, E_B\, \texttt{do}\, c \Downarrow s_3}$$

$$\frac{\Vdash s_1, E_B \Downarrow \texttt{true} \qquad S; O; F \Vdash s_1, c \Uparrow e, s_2}{S; O; F \Vdash s_1, \texttt{while}\, E_B\, \texttt{do}\, c \Uparrow e, s_2}$$

$$\frac{\Vdash s_1, E_B \Downarrow \texttt{false}}{S; O; F \Vdash s_1, \texttt{while}\, E_B\, \texttt{do}\, c \Downarrow s_1}$$

$$\frac{S; O; F \Vdash s_1\,[\,p_j \mapsto 0\,], c\,[\,y_j \mapsto p_j\,] \Downarrow s_2}{S; O; F \Vdash s_1, \texttt{local}\ y_1, y_2, \ldots\ \texttt{in}\ c \Downarrow (s_2 \upharpoonright \mathrm{dom}(s_1))}$$

$$\frac{S; O; F \Vdash s_1\,[\,p_j \mapsto 0\,], c\,[\,y_j \mapsto p_j\,] \Uparrow e, s_2}{S; O; F \Vdash s_1, \texttt{local}\ y_1, y_2, \ldots\ \texttt{in}\ c \Uparrow e, (s_2 \upharpoonright \mathrm{dom}(s_1))}$$

$$\frac{S; O; F\,[\,f \mapsto c_f\,] \Vdash s_1, c_B \Downarrow s_2}{S; O; F \Vdash s_1, \texttt{fun}\ f\ (\ ) = c_f\ \texttt{in}\ c_B \Downarrow s_2}$$

$$\frac{S; O; F\,[\,f \mapsto c_f\,] \Vdash s_1, c_B \Uparrow e, s_2}{S; O; F \Vdash s_1, \texttt{fun}\ f\ (\ ) = c_f\ \texttt{in}\ c_B \Uparrow e, s_2}$$

$$\frac{F(f) = c_f \qquad S; O; F \Vdash s_1, c_f \Downarrow s_2}{S; O; F \Vdash s_1, f(\ ); \Downarrow s_2} \quad \frac{F(f) = c_f \qquad S; O; F \Vdash s_1, c_f \Uparrow e, s_2}{S; O; F \Vdash s_1, f(\ ); \Uparrow e, s_2}$$

$$\frac{f_z \in \mathrm{dom}(F) \qquad S; O\,[\,z \mapsto f_z\,]; F \Vdash s_1, c_B \Downarrow s_2}{S; O; F \Vdash s_1, \texttt{bind/1}\ z\ \texttt{to}\ f_z\ \texttt{in}\ c_B \Downarrow s_2}$$

$$\frac{f_z \in \mathrm{dom}(F) \qquad S; O\,[\,z \mapsto f_z\,]; F \Vdash s_1, c_B \Uparrow e, s_2}{S; O; F \Vdash s_1, \texttt{bind/1}\ z\ \texttt{to}\ f_z\ \texttt{in}\ c_B \Uparrow e, s_2}$$

$$\frac{f_z \in \mathrm{dom}(F) \qquad S\,[\,z \mapsto f_z\,]; O; F \Vdash s_1, c_B \Downarrow s_2}{S; O; F \Vdash s_1, \texttt{bind}\ z\ \texttt{to}\ f_z\ \texttt{in}\ c_B \Downarrow s_2}$$

$$\frac{f_z \in \mathrm{dom}(F) \qquad S\,[\,z \mapsto f_z\,]; O; F \Vdash s_1, c_B \Uparrow e, s_2}{S; O; F \Vdash s_1, \texttt{bind}\ z\ \texttt{to}\ f_z\ \texttt{in}\ c_B \Uparrow e, s_2}$$

$$\frac{S - z; O - z; F \Vdash s_1, c \Downarrow s_2}{S; O; F \Vdash s_1, \texttt{block}\ z\ \texttt{in}\ c \Downarrow s_2} \quad \frac{S - z; O - z; F \Vdash s_1, c \Uparrow e, s_2}{S; O; F \Vdash s_1, \texttt{block}\ z\ \texttt{in}\ c \Uparrow e, s_2}$$

Figure 9.2: Big-step rules for operational semantics - Part 1

$$\frac{S; O_1; F \Vdash s_1, c_B \Downarrow s_2 \qquad S(z) = f_z \qquad S - z; O_2; F \Vdash s_2, f_z(\,); \Downarrow s_3}{S; O_1 * O_2; F \Vdash s_1, c_B \Downarrow s_3}$$

$$\frac{S(z) = f_z \qquad S - z; O_1; F \Vdash s_1, f_z(\,); \Downarrow s_2 \qquad S; O_2; F \Vdash s_2, c_B \Downarrow s_3}{S; O_1 * O_2; F \Vdash s_1, c_B \Downarrow s_3}$$

$$\frac{S; O_1 - z; F \Vdash s_1, c_B \Downarrow s_2 \quad O_1 * O_2(z) = f_z \quad S; O_2 - z; F \Vdash s_2, f_z(\,); \Downarrow s_3}{S; O_1 * O_2; F \Vdash s_1, c_B \Downarrow s_3}$$

$$\frac{O_1 * O_2(z) = f_z \quad S; O_1 - z; F \Vdash s_1, f_z(\,); \Downarrow s_2 \quad S; O_2 - z; F \Vdash s_2, c_B \Downarrow s_3}{S; O_1 * O_2; F \Vdash s_1, c_B \Downarrow s_3}$$

$$\frac{S; O_1; F \Vdash s_1, c_B \Downarrow s_2 \qquad S(z) = f_z \qquad S - z; O_2; F \Vdash s_2, f_z(\,); \Uparrow e, s_3}{S; O_1 * O_2; F \Vdash s_1, c_B \Uparrow e, s_3}$$

$$\frac{S(z) = f_z \qquad S - z; O_1; F \Vdash s_1, f_z(\,); \Downarrow s_2 \qquad S; O_2; F \Vdash s_2, c_B \Uparrow e, s_3}{S; O_1 * O_2; F \Vdash s_1, c_B \Uparrow e, s_3}$$

$$\frac{S; O_1 - z; F \Vdash s_1, c_B \Downarrow s_2 \quad O_1 * O_2(z) = f_z \quad S; O_2 - z; F \Vdash s_2, f_z(\,); \Uparrow e, s_3}{S; O_1 * O_2; F \Vdash s_1, c_B \Uparrow e, s_3}$$

$$\frac{O_1 * O_2(z) = f_z \quad S; O_1 - z; F \Vdash s_1, f_z(\,); \Downarrow s_2 \quad S; O_2 - z; F \Vdash s_2, c_B \Uparrow e, s_3}{S; O_1 * O_2; F \Vdash s_1, c_B \Uparrow e, s_3}$$

$$\frac{S(z) = f_z \qquad S - z; O; F \Vdash s_1, f_z(\,); \Uparrow e, s_2}{S; O; F \Vdash s_1, c_B \Uparrow e, s_2}$$

$$\frac{O(z) = f_z \qquad S; O - z; F \Vdash s_1, f_z(\,); \Uparrow e, s_2}{S; O; F \Vdash s_1, c_B \Uparrow e, s_2}$$

Figure 9.3: Big-step rules for operational semantics - Part 2

$$\frac{s_1, a \Downarrow s_2}{S; O; F \Vdash s_1, a \Downarrow s_2} \qquad \frac{\Vdash s_1, E \Downarrow v}{S; O; F \Vdash s_1, x := E \Downarrow s_1 \,[\, x \mapsto v \,]}$$

$$\frac{S; O_1; F \Vdash s_1, c_1 \Downarrow s_2 \qquad S; O_2; F \Vdash s_2, c_2 \Uparrow e, s_3}{S; O_1 * O_2; F \Vdash s_1, c_1 \,;\, c_2 \Uparrow e, s_3}$$

$$\frac{S; O_1; F \Vdash s_1, c_1 \Downarrow s_2 \qquad S; O_2; F \Vdash s_2, c_2 \Downarrow s_3}{S; O_1 * O_2; F \Vdash s_1, c_1 \,;\, c_2 \Downarrow s_3}$$

$$\frac{S; O_1; F \Vdash s_1, c_1 \Uparrow e, s_2}{S; O_1 * O_2; F \Vdash s_1, c_1 \,;\, c_2 \Uparrow e, s_2} \qquad \frac{}{S; O; F \Vdash s_1, \texttt{throw}\, e \Uparrow e, s_1}$$

$$\frac{S; O; F \Vdash s_1, c_B \Uparrow e, s_2 \qquad S; O; F \Vdash s_2, c_h \Downarrow s_3}{S; O; F \Vdash s_1, \texttt{try}\ c_B\ \texttt{handle}\ e\ \texttt{by}\ c_h \Downarrow s_3}$$

$$\frac{S; O; F \Vdash s_1, c_B \Downarrow s_2}{S; O; F \Vdash s_1, \texttt{try}\ c_B\ \texttt{handle}\ e\ \texttt{by}\ c_h \Downarrow s_2}$$

$$\frac{S; O; F \Vdash s_1, c_B \Uparrow e, s_2 \qquad S; O; F \Vdash s_2, c_h \Uparrow e_2, s_3}{S; O; F \Vdash s_1, \texttt{try}\ c_B\ \texttt{handle}\ e\ \texttt{by}\ c_h \Uparrow e_2, s_3}$$

$$\frac{S; O; F \Vdash s_1, c_B \Uparrow e_2, s_2 \qquad e_2 \neq e}{S; O; F \Vdash s_1, \texttt{try}\ c_B\ \texttt{handle}\ e\ \texttt{by}\ c_h \Uparrow e_2, s_2}$$

$$\frac{\Vdash s_1, E_B \Downarrow \texttt{true} \qquad S; O; F \Vdash s_1, c_1 \Downarrow s_2}{S; O; F \Vdash s_1, \texttt{if}\ E_B\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \Downarrow s_2}$$

$$\frac{\Vdash s_1, E_B \Downarrow \texttt{true} \qquad S; O; F \Vdash s_1, c_1 \Uparrow e, s_2}{S; O; F \Vdash s_1, \texttt{if}\ E_B\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \Uparrow e, s_2}$$

$$\frac{\Vdash s_1, E_B \Downarrow \texttt{false} \qquad S; O; F \Vdash s_1, c_2 \Downarrow s_2}{S; O; F \Vdash s_1, \texttt{if}\ E_B\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \Downarrow s_2}$$

$$\frac{\Vdash s_1, E_B \Downarrow \texttt{false} \qquad S; O; F \Vdash s_1, c_2 \Uparrow e, s_2}{S; O; F \Vdash s_1, \texttt{if}\ E_B\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \Uparrow e, s_2}$$

Figure 9.4: Big-step rules for operational semantics - Part 3

9.1.1 Big-Step Rules in Detail

The present set of the big-step rules builds on the language presented in Chapter 3. To keep the rules short, the exception convention is assumed implicitly.

Local variables

The local variables are replaced by a fresh location (p) that is initialised to 0 or `false` depending on the required type (integer or boolean).

$$\frac{S; O; F \Vdash s_1\,[\,p_j \mapsto 0\,], c\,[\,y_j \mapsto p_j\,] \Downarrow s_2}{S; O; F \Vdash s_1, \texttt{local}\ y_1, y_2, \ldots\ \texttt{in}\ c \Downarrow (s_2 \upharpoonright \mathrm{dom}(s_1))}$$

$$\frac{S; O; F \Vdash s_1\,[\,p_j \mapsto 0\,], c\,[\,y_j \mapsto p_j\,] \Uparrow e, s_2}{S; O; F \Vdash s_1, \texttt{local}\ y_1, y_2, \ldots\ \texttt{in}\ c \Uparrow e, (s_2 \upharpoonright \mathrm{dom}(s_1))}$$

State limitation ($s_2 \upharpoonright \mathrm{dom}(s_1)$) results into location cleanup. Thus, all local variables that have been introduced during the state change to s_2 are discarded, and only local variables that had been introduced before state change from s_1 to s_2 remains.

Function declaration

$$\frac{S; O; F\,[\,f \mapsto c_f\,] \Vdash s_1, c_B \Downarrow s_2}{S; O; F \Vdash s_1, \texttt{fun}\ f\ (\ ) = c_f\ \texttt{in}\ c_B \Downarrow s_2}$$

$$\frac{S; O; F\,[\,f \mapsto c_f\,] \Vdash s_1, c_B \Uparrow e, s_2}{S; O; F \Vdash s_1, \texttt{fun}\ f\ (\ ) = c_f\ \texttt{in}\ c_B \Uparrow e, s_2}$$

We keep two separate rules for the local variables and the function calls. Therefore, the rule for the function call doesn't include state limitation (location clean up).

Function call of non-recursive functions

For the non-recursive functions, the running function is excluded from the function context F.

$$\frac{F(f) = c_f \qquad S; O; F - f \Vdash s_1, c_f \Downarrow s_2}{S; O; F \Vdash s_1, f(\,); \Downarrow s_2}$$

$$\frac{F(f) = c_f \qquad S; O; F - f \Vdash s_1, c_f \Uparrow e, s_2}{S; O; F \Vdash s_1, f(\,); \Uparrow e, s_2}$$

Function call of recursive functions

For the recursive functions, the running function is not excluded from the function context F in contrast to the non-recursive functions.

$$\frac{F(f) = c_f \qquad S; O; F \Vdash s_1, c_f \Downarrow s_2}{S; O; F \Vdash s_1, f(\,); \Downarrow s_2}$$

$$\frac{F(f) = c_f \qquad S; O; F \Vdash s_1, c_f \Uparrow e, s_2}{S; O; F \Vdash s_1, f(\,); \Uparrow e, s_2}$$

Signal binding and blocking

To bind a function f_z as a signal handler for the signal z, a function context F should contain the function f_z. Therefore, we have $f_z \in \mathrm{dom}(F)$ condition in binding rules. To block a signal z, it is removed from the signal context S and O. See Figure 9.5.

Signal handling

In Figure 9.6, we consider three cases such as signal handlers are uninterruptible, partially interruptible (cannot interrupt itself), and fully interruptible. One may observe, that there is no difference in the last two groups of the rules for the ones-shot signal handling. One-shot signals could be handled only once; thus, the one-shot signal handler cannot interrupt itself, as it contradicts an idea of the one-shot signals in general. Examples of the signal handling combinations with functions are given in Figure 9.7.

In Figure 9.8, we observe that the non-recursive function prevents the signal handlers from calling itself. Despite the fact that the signal bindings are not restricted, this limi-

$$\frac{f_z \in \mathrm{dom}(F) \qquad S; O\,[\,z \mapsto f_z\,]; F \Vdash s_1, c_B \Downarrow s_2}{S; O; F \Vdash s_1, \texttt{bind/1}\ z\ \texttt{to}\ f_z\ \texttt{in}\ c_B \Downarrow s_2}$$

$$\frac{f_z \in \mathrm{dom}(F) \qquad S; O\,[\,z \mapsto f_z\,]; F \Vdash s_1, c_B \Uparrow e, s_2}{S; O; F \Vdash s_1, \texttt{bind/1}\ z\ \texttt{to}\ f_z\ \texttt{in}\ c_B \Uparrow e, s_2}$$

$$\frac{f_z \in \mathrm{dom}(F) \qquad S\,[\,z \mapsto f_z\,]; O; F \Vdash s_1, c_B \Downarrow s_2}{S; O; F \Vdash s_1, \texttt{bind}\ z\ \texttt{to}\ f_z\ \texttt{in}\ c_B \Downarrow s_2}$$

$$\frac{f_z \in \mathrm{dom}(F) \qquad S\,[\,z \mapsto f_z\,]; O; F \Vdash s_1, c_B \Uparrow e, s_2}{S; O; F \Vdash s_1, \texttt{bind}\ z\ \texttt{to}\ f_z\ \texttt{in}\ c_B \Uparrow e, s_2}$$

$$\frac{S - z; O - z; F \Vdash s_1, c \Downarrow s_2}{S; O; F \Vdash s_1, \texttt{block}\ z\ \texttt{in}\ c \Downarrow s_2}$$

$$\frac{S - z; O - z; F \Vdash s_1, c \Uparrow e, s_2}{S; O; F \Vdash s_1, \texttt{block}\ z\ \texttt{in}\ c \Uparrow e, s_2}$$

Figure 9.5: Signal binding and blocking

tation arise from the nature of non-recursive functions. Therefore, to address reentrancy our language has to support recursive functions. The corresponding examples are given in Figure 9.9.

According to the example in Figure 9.10, one may say that there is no explicit need to support recursive functions, as by the nature of one-shot signals, a signal handler cannot be executed twice. However, the next situation should be considered as well. The same function could be bound to a few different signals. For example, see Figure 9.11. Therefore, recursive functions are needed even for the one-shot signal handling.

Assignment

It is analogous to the rule defined in Chapter 3, but extended with a function context F.

$$\frac{\Vdash s_1, E \Downarrow v}{S; O; F \Vdash s_1, x := E \Downarrow s_1\,[\,x \mapsto v\,]}$$

Signal handlers are uninterruptible:

$$\frac{S; O; F \Vdash s_1, c_B \Downarrow s_2 \qquad S(z) = f_z \qquad \emptyset; \emptyset; F \Vdash s_2, f_z(\,); \Downarrow s_3}{S; O; F \Vdash s_1, c_B \Downarrow s_3}$$

$$\frac{S(z) = f_z \qquad \emptyset; \emptyset; F \Vdash s_1, f_z(\,); \Downarrow s_2 \qquad S; O; F \Vdash s_2, c_B \Downarrow s_3}{S; O; F \Vdash s_1, c_B \Downarrow s_3}$$

$$\frac{S; O - z; F \Vdash s_1, c_B \Downarrow s_2 \qquad O(z) = f_z \qquad \emptyset; \emptyset; F \Vdash s_2, f_z(\,); \Downarrow s_3}{S; O; F \Vdash s_1, c_B \Downarrow s_3}$$

$$\frac{O(z) = f_z \qquad \emptyset; \emptyset; F \Vdash s_1, f_z(\,); \Downarrow s_2 \qquad S; O - z; F \Vdash s_2, c_B \Downarrow s_3}{S; O; F \Vdash s_1, c_B \Downarrow s_3}$$

Signal handlers are interruptible, except by itself:

$$\frac{S; O_1; F \Vdash s_1, c_B \Downarrow s_2 \qquad S(z) = f_z \qquad S - z; O_2; F \Vdash s_2, f_z(\,); \Downarrow s_3}{S; O_1 * O_2; F \Vdash s_1, c_B \Downarrow s_3}$$

$$\frac{S(z) = f_z \qquad S - z; O_1; F \Vdash s_1, f_z(\,); \Downarrow s_2 \qquad S; O_2; F \Vdash s_2, c_B \Downarrow s_3}{S; O_1 * O_2; F \Vdash s_1, c_B \Downarrow s_3}$$

$$\frac{S; O_1 - z; F \Vdash s_1, c_B \Downarrow s_2 \quad O_1 * O_2(z) = f_z \quad S; O_2 - z; F \Vdash s_2, f_z(\,); \Downarrow s_3}{S; O_1 * O_2; F \Vdash s_1, c_B \Downarrow s_3}$$

$$\frac{O_1 * O_2(z) = f_z \quad S; O_1 - z; F \Vdash s_1, f_z(\,); \Downarrow s_2 \quad S; O_2 - z; F \Vdash s_2, c_B \Downarrow s_3}{S; O_1 * O_2; F \Vdash s_1, c_B \Downarrow s_3}$$

Signal handlers are fully interruptible:

$$\frac{S; O_1; F \Vdash s_1, c_B \Downarrow s_2 \qquad S(z) = f_z \qquad S; O_2; F \Vdash s_2, f_z(\,); \Downarrow s_3}{S; O_1 * O_2; F \Vdash s_1, c_B \Downarrow s_3}$$

$$\frac{S(z) = f_z \qquad S; O_1; F \Vdash s_1, f_z(\,); \Downarrow s_2 \qquad S; O_2; F \Vdash s_2, c_B \Downarrow s_3}{S; O_1 * O_2; F \Vdash s_1, c_B \Downarrow s_3}$$

$$\frac{S; O_1 - z; F \Vdash s_1, c_B \Downarrow s_2 \quad O_1 * O_2(z) = f_z \quad S; O_2 - z; F \Vdash s_2, f_z(\,); \Downarrow s_3}{S; O_1 * O_2; F \Vdash s_1, c_B \Downarrow s_3}$$

$$\frac{O_1 * O_2(z) = f_z \quad S; O_1 - z; F \Vdash s_1, f_z(\,); \Downarrow s_2 \quad S; O_2 - z; F \Vdash s_2, c_B \Downarrow s_3}{S; O_1 * O_2; F \Vdash s_1, c_B \Downarrow s_3}$$

Figure 9.6: Signal interruption

$$\frac{F(f_z) = c_z \qquad \emptyset;\emptyset;F{-}f_z \Vdash s_2, c_z \Downarrow s_3}{\emptyset;\emptyset;F \Vdash s_2, f_z(\,);\Downarrow s_3}$$

$$\frac{S;O;F \Vdash s_1, c_B \Downarrow s_2 \qquad S(z) = f_z \qquad \emptyset;\emptyset;F \Vdash s_2, f_z(\,);\Downarrow s_3}{S;O;F \Vdash s_1, c_B \Downarrow s_3}$$

$$\frac{F(f_z) = c_z \qquad S - z;O_2;F{-}f_z \Vdash s_2, c_z \Downarrow s_3}{S - z;O_2;F \Vdash s_2, f_z(\,);\Downarrow s_3}$$

$$\frac{S;O_1;F \Vdash s_1, c_B \Downarrow s_2 \qquad S(z) = f_z \qquad S - z;O_2;F \Vdash s_2, f_z(\,);\Downarrow s_3}{S;O_1 * O_2;F \Vdash s_1, c_B \Downarrow s_3}$$

$$\frac{F(f_z) = c_z \qquad S;O_2;F{-}f_z \Vdash s_2, c_z \Downarrow s_3}{S;O_2;F \Vdash s_2, f_z(\,);\Downarrow s_3}$$

$$\frac{S;O_1;F \Vdash s_1, c_B \Downarrow s_2 \qquad S(z) = f_z \qquad S;O_2;F \Vdash s_2, f_z(\,);\Downarrow s_3}{S;O_1 * O_2;F \Vdash s_1, c_B \Downarrow s_3}$$

$$\frac{F(f_z) = c_z \qquad \emptyset;\emptyset;F{-}f_z \Vdash s_2, c_z \Downarrow s_3}{\emptyset;\emptyset;F \Vdash s_2, f_z(\,);\Downarrow s_3}$$

$$\frac{S;O;F \Vdash s_1, c_B \Downarrow s_2 \qquad O(z) = f_z \qquad \emptyset;\emptyset;F \Vdash s_2, f_z(\,);\Downarrow s_3}{S;O;F \Vdash s_1, c_B \Downarrow s_3}$$

$$\frac{F(f_z) = c_z \qquad S;O_2 - z;F{-}f_z \Vdash s_2, c_z \Downarrow s_3}{S;O_2 - z;F \Vdash s_2, f_z(\,);\Downarrow s_3}$$

$$\frac{S;O_1 - z;F \Vdash s_1, c_B \Downarrow s_2 \qquad O_1 * O_2(z) = f_z \qquad S;O_2 - z;F \Vdash s_2, f_z(\,);\Downarrow s_3}{S;O_1 * O_2;F \Vdash s_1, c_B \Downarrow s_3}$$

$$\frac{F(f_z) = c_z \qquad S;O_2 - z;F{-}f_z \Vdash s_2, c_z \Downarrow s_3}{S;O_2 - z;F \Vdash s_2, f_z(\,);\Downarrow s_3}$$

$$\frac{S;O_1 - z;F \Vdash s_1, c_B \Downarrow s_2 \qquad O_1 * O_2(z) = f_z \qquad S;O_2 - z;F \Vdash s_2, f_z(\,);\Downarrow s_3}{S;O_1 * O_2;F \Vdash s_1, c_B \Downarrow s_3}$$

Figure 9.7: Persistent and One-shot Signal handling

$$\frac{S; O_2'; F{-}f_z \Vdash s_2, c_z \Downarrow s_3 \qquad S(z) = f_z \qquad \dfrac{(F{-}f_z)(f_z) = \emptyset \qquad \lightning}{S; O_2''; F{-}f_z \Vdash s_3, f_z(\,); \Downarrow s_4}}{S; O_2' * O_2''; F{-}f_z \Vdash s_2, c_z \Downarrow s_4}$$

$$\frac{F(f_z) = c_z \qquad S; O_2' * O_2''; F{-}f_z \Vdash s_2, c_z \Downarrow s_4}{S; O_2; F \Vdash s_2, f_z(\,); \Downarrow s_4}$$

$$\frac{S; O_1; F \Vdash s_1, c_B \Downarrow s_2 \qquad S(z) = f_z \qquad S; O_2; F \Vdash s_2, f_z(\,); \Downarrow s_4}{S; O_1 * O_2; F \Vdash s_1, c_B \Downarrow s_4}$$

Figure 9.8: Non-rec function prevents the handlers from calling itself

$$\frac{S; O_2'; F \Vdash s_2, c_z \Downarrow s_3 \qquad S(z) = f_z \qquad \dfrac{F(f_z) = c_z \qquad S; O_2''; F \Vdash s_3, c_z \Downarrow s_4}{S; O_2''; F \Vdash s_3, f_z(\,); \Downarrow s_4}}{S; O_2' * O_2''; F \Vdash s_2, c_z \Downarrow s_4}$$

$$\frac{F(f_z) = c_z \qquad S; O_2' * O_2''; F \Vdash s_2, c_z \Downarrow s_4}{S; O_2; F \Vdash s_2, f_z(\,); \Downarrow s_4}$$

$$\frac{S; O_1; F \Vdash s_1, c_B \Downarrow s_2 \qquad S(z) = f_z \qquad S; O_2; F \Vdash s_2, f_z(\,); \Downarrow s_4}{S; O_1 * O_2; F \Vdash s_1, c_B \Downarrow s_4}$$

Figure 9.9: Recursive function

$$\frac{F(f_z) = c_z \qquad S; O_2 - z; F \Vdash s_2, c_z \Downarrow s_3}{S; O_2 - z; F \Vdash s_2, f_z(\,); \Downarrow s_3}$$

$$\frac{S; O_1 - z; F \Vdash s_1, c_B \Downarrow s_2 \quad O_1 * O_2(z) = f_z \quad S; O_2 - z; F \Vdash s_2, f_z(\,); \Downarrow s_3}{S; O_1 * O_2; F \Vdash s_1, c_B \Downarrow s_3}$$

Figure 9.10: One-shot signal handling and non-recursive functions

$$\frac{F(f_z) = c_z \qquad S; O_4 - z'; F \Vdash s_3, c_z \Downarrow s_4}{S; O_4 - z'; F \Vdash s_3, f_z(\,); \Downarrow s_4}$$

$$\frac{S; O_3 - z'; F \Vdash s_2, c_z \Downarrow s_3 \quad O_3 * O_4(z') = f_z \quad S; O_4 - z'; F \Vdash s_3, f_z(\,); \Downarrow s_4}{S; O_3 * O_4; F \Vdash s_2, c_z \Downarrow s_4}$$

$$\frac{F(f_z) = c_z \qquad S; O_2 - z; F \Vdash s_2, c_z \Downarrow s_4}{S; O_2 - z; F \Vdash s_2, f_z(\,); \Downarrow s_4}$$

$$\frac{S; O_1 - z; F \Vdash s_1, c_B \Downarrow s_2 \quad O_1 * O_2(z) = f_z \quad S; O_2 - z; F \Vdash s_2, f_z(\,); \Downarrow s_4}{S; O_1 * O_2; F \Vdash s_1, c_B \Downarrow s_4}$$

where $O_2 - z = O_3 * O_4$

Figure 9.11: One-shot signal handling and recursive functions

Increment and Decrement

The rules are analogous to the rules defined in Chapter 3, but extended with a function context F.

$$\frac{\Vdash s_1, x \Downarrow v}{S; O; F \Vdash s_1, x{+}{+} \Downarrow s_1\,[\,x \mapsto v + 1\,]}$$

$$\frac{\Vdash s_1, x \Downarrow v}{S; O; F \Vdash s_1, x{-}{-} \Downarrow s_1\,[\,x \mapsto v - 1\,]}$$

Sequential composition

These rules are analogous to the rules defined in Chapter 3, but extended with a function context F. Also note that function context is copied between branches similar to persistent signal context.

$$\frac{S; O_1; F \Vdash s_1, c_1 \Downarrow s_2 \qquad S; O_2; F \Vdash s_2, c_2 \Downarrow s_3}{S; O_1 * O_2; F \Vdash s_1, c_1 \,;\, c_2 \Downarrow s_3}$$

$$\frac{S; O_1; F \Vdash s_1, c_1 \Downarrow s_2 \qquad S; O_2; F \Vdash s_2, c_2 \Uparrow e, s_3}{S; O_1 * O_2; F \Vdash s_1, c_1 \,;\, c_2 \Uparrow e, s_3}$$

$$\frac{S; O_1; F \Vdash s_1, c_1 \Uparrow e, s_2}{S; O_1 * O_2; F \Vdash s_1, c_1 \,;\, c_2 \Uparrow e, s_2}$$

Repetitive `while` command

Rules for the `while` command are analogous to the rules defined in Chapter 3, but extended with a function context F.

$$\frac{\Vdash s_1, E_B \Downarrow \texttt{false}}{S; O; F \Vdash s_1, \texttt{while}\, E_B\, \texttt{do}\, c \Downarrow s_1}$$

$$\frac{\Vdash s_1, E_B \Downarrow \texttt{true} \quad S; O_1; F \Vdash s_1, c \Downarrow s_2 \quad S; O_2; F \Vdash s_2, \texttt{while}\, E_B\, \texttt{do}\, c \Downarrow s_3}{S; O_1 * O_2; F \Vdash s_1, \texttt{while}\, E_B\, \texttt{do}\, c \Downarrow s_3}$$

$$\frac{\Vdash s_1, E_B \Downarrow \texttt{true} \qquad S; O; F \Vdash s_1, c \Uparrow e, s_2}{S; O; F \Vdash s_1, \texttt{while}\, E_B\, \texttt{do}\, c \Uparrow e, s_2}$$

Conditional `if-else` structure

Rules for the `if-else` structure are analogous to the rules defined in Chapter 3, but extended with a function context F.

$$\frac{\Vdash s_1, E_B \Downarrow \texttt{true} \qquad S; O; F \Vdash s_1, c_1 \Downarrow s_2}{S; O; F \Vdash s_1, \texttt{if}\ E_B\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \Downarrow s_2}$$

$$\frac{\Vdash s_1, E_B \Downarrow \texttt{false} \qquad S; O; F \Vdash s_1, c_2 \Downarrow s_2}{S; O; F \Vdash s_1, \texttt{if}\ E_B\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \Downarrow s_2}$$

$$\frac{\Vdash s_1, E_B \Downarrow \texttt{true} \qquad S; O; F \Vdash s_1, c_1 \Uparrow e, s_2}{S; O; F \Vdash s_1, \texttt{if}\ E_B\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \Uparrow e, s_2}$$

$$\frac{\Vdash s_1, E_B \Downarrow \texttt{false} \qquad S; O; F \Vdash s_1, c_2 \Uparrow e, s_2}{S; O; F \Vdash s_1, \texttt{if}\ E_B\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \Uparrow e, s_2}$$

Exception handling

Rules for the exception handling are analogous to the rules defined in Chapter 3, but extended with a function context F.

$$\frac{}{S; O; F \Vdash s_1, \texttt{throw}\ e \Uparrow e, s_1}$$

$$\frac{S; O; F \Vdash s_1, c_B \Uparrow e, s_2 \qquad S; O; F \Vdash s_2, c_h \Downarrow s_3}{S; O; F \Vdash s_1, \texttt{try}\ c_B\ \texttt{handle}\ e\ \texttt{by}\ c_h \Downarrow s_3}$$

$$\frac{S; O; F \Vdash s_1, c_B \Downarrow s_2}{S; O; F \Vdash s_1, \texttt{try}\ c_B\ \texttt{handle}\ e\ \texttt{by}\ c_h \Downarrow s_2}$$

$$\frac{S; O; F \Vdash s_1, c_B \Uparrow e, s_2 \qquad S; O; F \Vdash s_2, c_h \Uparrow e_2, s_3}{S; O; F \Vdash s_1, \texttt{try}\ c_B\ \texttt{handle}\ e\ \texttt{by}\ c_h \Uparrow e_2, s_3}$$

$$\frac{S; O; F \Vdash s_1, c_B \Uparrow e_2, s_2 \qquad e_2 \neq e}{S; O; F \Vdash s_1, \texttt{try}\ c_B\ \texttt{handle}\ e\ \texttt{by}\ c_h \Uparrow e_2, s_2}$$

$$\frac{S; O; F\,[\,f \mapsto c_f\,] \Vdash s_1\,[\,p_1 \mapsto 0\,]\,[\,p_2 \mapsto 0\,], c_B\,[\,y_1 \mapsto p_1\,]\,[\,y_2 \mapsto p_2\,] \Downarrow s_1\,[\,p_1 \mapsto 3\,]\,[\,p_2 \mapsto 9\,]}{S; O; F \Vdash s_1\,[\,p_1 \mapsto 0\,]\,[\,p_2 \mapsto 0\,], (\texttt{fun}\ f\ (\,) = c_f\ \texttt{in}\ c_B)\,[\,y_1 \mapsto p_1\,]\,[\,y_2 \mapsto p_2\,] \Downarrow s_1\,[\,p_1 \mapsto 3\,]\,[\,p_2 \mapsto 9\,]}$$

$$S; O; F \Vdash s_1, \texttt{local}\ y_1, y_2\ \texttt{in}\ (\texttt{fun}\ f\ (\,) = c_f\ \texttt{in}\ c_B) \Downarrow ((s_1\,[\,p_1 \mapsto 3\,]\,[\,p_2 \mapsto 9\,]) \restriction \mathrm{dom}(s_1))$$

where $c_f = y_2 := y_1 * y_1$; and $c_B = y_1 := 3; f(\,)$;

$$\frac{;; \Vdash s_1\,[\,p_1 \mapsto 0\,]\,[\,p_2 \mapsto 0\,], 3 \Downarrow 3}{\dfrac{S; O; F\,[\,f \mapsto c_f\,] \Vdash s_1\,[\,p_1 \mapsto 0\,]\,[\,p_2 \mapsto 0\,], y_1 := 3; \,[\,y_1 \mapsto p_1\,]\,[\,y_2 \mapsto p_2\,] \Downarrow s_1\,[\,p_1 \mapsto 3\,]\,[\,p_2 \mapsto 0\,] \qquad \mathcal{A}}{S; O; F\,[\,f \mapsto c_f\,] \Vdash s_1\,[\,p_1 \mapsto 0\,]\,[\,p_2 \mapsto 0\,], (y_1 := 3; f(\,);)\,[\,y_1 \mapsto p_1\,]\,[\,y_2 \mapsto p_2\,] \Downarrow s_1\,[\,p_1 \mapsto 3\,]\,[\,p_2 \mapsto 9\,]}}$$

$\mathcal{A} =$

$$\frac{F\,[\,f \mapsto c_f\,](f) = c_f \qquad S; O; F \Vdash s_1\,[\,p_1 \mapsto 3\,]\,[\,p_2 \mapsto 0\,], c_f\,[\,y_1 \mapsto p_1\,]\,[\,y_2 \mapsto p_2\,] \Downarrow s_1\,[\,p_1 \mapsto 3\,]\,[\,p_2 \mapsto 9\,]}{S; O; F\,[\,f \mapsto c_f\,] \Vdash s_1\,[\,p_1 \mapsto 3\,]\,[\,p_2 \mapsto 0\,], f(\,);\,[\,y_1 \mapsto p_1\,]\,[\,y_2 \mapsto p_2\,] \Downarrow s_1\,[\,p_1 \mapsto 3\,]\,[\,p_2 \mapsto 9\,]}$$

Figure 9.12: Imitation of the argument passing and return

9.1.2 Argument Passing and Global Variables

Global variables could be used to replace argument passing via parameters, and return mechanism of the functions. The global variables themselves could be imitated using local variables. Before any function definition, two local variables could be defined. One could be used to provide arguments, and another to return results (imitation of a return value). Just assign a value to the first variable before the function call, and store a final value to the second variable inside of the function. For example, see Figure 9.12.

9.2 Logic and Reentrancy Linear Type System

We define a program logic and Reentrancy Linear Type system for the language (Section 9.1) with local variables, functions, signal and exception handling. Reentrancy Linear Type System ensures that non-reentrant functions are used at most once or not used at all in the environment with signals. This definition slightly deviates from the standard definitions of the Linear Logic, where `linear` resources are used exactly once and `affine` resources are used at most once. Thus, in the former case, resources should be used at

least once and at most once. In the latter case, there is no obligations to use a resource; thus, it might be used once or not at all.

9.2.1 Reentrancy Judgement

In our logic, a reentrancy judgment is of the next form

$$\Sigma; \Phi; \Psi \overset{R}{\vdash} c$$

Definition 9.2.1 (Reentrancy judgement) Let Σ be a signal context, Φ be a reentrant function context, Ψ be a non-reentrant function context, and c be a command. We say that $\Sigma; \Phi; \Psi \overset{R}{\vdash} c$ holds if for all function calls inside of the command c, that is the case that the functions are $\in \mathrm{dom}(\Phi)$ or all signals are blocked (in other words, Σ is $\emptyset$). That is checked by induction over the command c construction using rules from Figure 9.13 and Figure 9.14.

For any atomic command a, a reentrancy judgement $\Sigma; \Phi; \Psi \overset{R}{\vdash} a$ trivially holds.

Base cases

Let $f \in \mathrm{dom}(\Phi)$, then the reentrancy judgement holds.

$$\frac{f \in \mathrm{dom}(\Phi)}{\Sigma; \Phi; \Psi \overset{R}{\vdash} f(\,)}$$

Let $n \in \mathrm{dom}(\Psi)$ and signal context be empty, then the reentrancy judgement also holds.

$$\frac{n \in \mathrm{dom}(\Psi)}{\emptyset; \Phi; \Psi \overset{R}{\vdash} n(\,)}$$

$$\frac{}{\Sigma;\Phi;\Psi \overset{R}{\vdash} a}\ (\textsc{Atomic}) \qquad \frac{}{\Sigma;\Phi;\Psi \overset{R}{\vdash} \texttt{throw}\, e_j}\ (\textsc{Throw})$$

$$\frac{\Sigma;\Phi;\Psi \overset{R}{\vdash} c_1 \qquad \Sigma;\Phi;\Psi \overset{R}{\vdash} c_2}{\Sigma;\Phi;\Psi \overset{R}{\vdash} c_1\ ;\ c_2}\ (\textsc{Seq})$$

$$\frac{\Sigma;\Phi;\Psi \overset{R}{\vdash} c_B \qquad \Sigma;\Phi;\Psi \overset{R}{\vdash} c_h}{\Sigma;\Phi;\Psi \overset{R}{\vdash} \texttt{try}\ c_B\ \texttt{handle}\ e_k\ \texttt{by}\ c_h}\ (\textsc{ExnHandle})$$

$$\frac{n \in \mathrm{dom}(\Psi)}{\Sigma;\Phi;\Psi \overset{R}{\vdash} n(\ );}\ (\textsc{NFunCall}) \qquad \frac{n \in \mathrm{dom}(\Psi)}{\emptyset;\Phi;\Psi \overset{R}{\vdash} n(\ );}\ (\textsc{NFunCall})$$

$$\frac{f \in \mathrm{dom}(\Phi)}{\Sigma;\Phi;\Psi \overset{R}{\vdash} f(\ );}\ (\textsc{RFunCall})$$

$$\frac{\Sigma;\Phi;\Psi, n : P_n \triangleright Q_n \overset{R}{\vdash} c_B}{\Sigma;\Phi;\Psi \overset{R}{\vdash} \texttt{fuN}\ n\ (\ ) = c_n\ \texttt{in}\ c_B}\ (\textsc{NFunDef})$$

$$\frac{\Sigma;\Phi, f : P_f \triangleright Q_f;\Psi \overset{R}{\vdash} c_B \qquad \Sigma;\Phi, f : P_f \triangleright Q_f;\Psi \overset{R}{\vdash} c_f}{\Sigma;\Phi;\Psi \overset{R}{\vdash} \texttt{fun}\ f\ (\ ) = c_f\ \texttt{in}\ c_B}\ (\textsc{RFunDef})$$

$$\frac{\Sigma;\Phi;\Psi \overset{R}{\vdash} c}{\Sigma, z : P_z \triangleright Q_z;\Phi;\Psi \overset{R}{\vdash} \texttt{block}\ z\ \texttt{in}\ c}\ (\textsc{SigBlock})$$

$$\frac{\Sigma;\Phi;\Psi \overset{R}{\vdash} c_1 \qquad \Sigma;\Phi;\Psi \overset{R}{\vdash} c_2}{\Sigma;\Phi;\Psi \overset{R}{\vdash} \texttt{if}\ E_B\ \texttt{then}\ c_1\ \texttt{else}\ c_2}\ (\textsc{IfElse})$$

$$\frac{\Sigma;\Phi;\Psi \overset{R}{\vdash} c}{\Sigma;\Phi;\Psi \overset{R}{\vdash} \texttt{while}\ E_B\ \texttt{do}\ c}\ (\textsc{While}) \qquad \frac{\Sigma;\Phi;\Psi \overset{R}{\vdash} c}{\Sigma;\Phi;\Psi \overset{R}{\vdash} \texttt{local}\ y_1, y_2, \ldots\ \texttt{in}\ c}\ (\textsc{Var})$$

Figure 9.13: RLTS logic rules - Part 1

$$\frac{\Sigma, z : P_z \triangleright Q_z; \Phi, f : P_z \triangleright Q_z; \Psi_1 \overset{R}{\vdash} c_B \quad \Sigma; \Phi, f : P_z \triangleright Q_z; \Psi_2 \overset{R}{\vdash} f(\,);}{\Sigma; \Phi, f : P_z \triangleright Q_z; \Psi_1, \Psi_2 \overset{R}{\vdash} \texttt{bind/1}\ z\ \texttt{to}\ f\ \texttt{in}\ c_B}$$

$$\frac{\Sigma, z : I_z \triangleright I_z; \Phi, f : I_z \triangleright I_z; \Psi_1 \overset{R}{\vdash} c_B \qquad \Sigma; \Phi, f : I_z \triangleright I_z; \Psi_2 \overset{R}{\vdash} f(\,);}{\Sigma; \Phi, f : I_z \triangleright I_z; \Psi_1, \Psi_2 \overset{R}{\vdash} \texttt{bind}\ z\ \texttt{to}\ f\ \texttt{in}\ c_B}$$

$$\frac{\Sigma, z : P_z \triangleright Q_z; \Phi; \Psi_1 \overset{R}{\vdash} c_B \qquad \Sigma; \Phi; \Psi_2, n : P_z \triangleright Q_z \overset{R}{\vdash} n(\,);}{\Sigma; \Phi; \Psi_1, \Psi_2, n : P_z \triangleright Q_z \overset{R}{\vdash} \texttt{bind/1}\ z\ \texttt{to}\ n\ \texttt{in}\ c_B}$$

$$\frac{\Sigma, z : I_z \triangleright I_z; \Phi; \Psi_1 \overset{R}{\vdash} c_B \qquad \Sigma; \Phi; \Psi_2, n : I_z \triangleright I_z \overset{R}{\vdash} n(\,);}{\Sigma; \Phi; \Psi_1, \Psi_2, n : I_z \triangleright I_z \overset{R}{\vdash} \texttt{bind}\ z\ \texttt{to}\ n\ \texttt{in}\ c_B}$$

Figure 9.14: RLTS logic rules - Part 2

Advanced example

Any non-reentrant function call with unrestricted signal context is potentially unsafe. Can we say the same about the non-reentrant function definition? A body c_n of the non-reentrant function n, may contain function calls of any non-reentrant function from Ψ. And there is no need to check it for reentrancy, as it is by definition non-reentrant. That is why we have a separate rule `fuN` for the non-reentrant function definition. However, we should check the command c_f. If it does not call any non-reentrant functions, then the whole construct is reentrant safe. On the other hand, if it makes a call to any non-reentrant function (with unrestricted signal context), then the reentrancy judgement could not be derived; thus, the whole construct is not reentrancy safe.

$$\frac{\Sigma; \Phi, f : P_f \triangleright Q_f; \Psi, n : P_n \triangleright Q_n \overset{R}{\vdash} c_f}{\Sigma; \Phi, f : P_f \triangleright Q_f; \Psi \overset{R}{\vdash} (\texttt{fuN}\ n\ (\ ) = c_n\ \texttt{in}\ c_f)}$$

$$\frac{\Sigma; \Phi, f : P_f \triangleright Q_f; \Psi, n : P_n \triangleright Q_n \vdash \{P_f\}\, c_f\, \{Q_f\}\ \texttt{throws}\ \eta \qquad \Sigma; \Phi, f : P_f \triangleright Q_f; \Psi, n : P_n \triangleright Q_n \vdash \{P_n\}\, c_n\, \{Q_n\}\ \texttt{throws}\ \eta}{\Sigma; \Phi, f : P_f \triangleright Q_f; \Psi \vdash \{P_f\}\, \texttt{fuN}\ n\ (\ ) = c_n\ \texttt{in}\ c_f\, \{Q_f\}\ \texttt{throws}\ \eta}$$

$$\frac{\Sigma; \Phi, f : P_f \triangleright Q_f; \Psi \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta \qquad \Sigma; \Phi, f : P_f \triangleright Q_f; \Psi \vdash \{P_f\}\, \texttt{fuN}\ n\ (\ ) = c_n\ \texttt{in}\ c_f\, \{Q_f\}\ \texttt{throws}\ \eta \qquad \Sigma; \Phi, f : P_f \triangleright Q_f; \Psi \overset{R}{\vdash} (\texttt{fuN}\ n\ (\ ) = c_n\ \texttt{in}\ c_f)}{\Sigma; \Phi; \Psi \vdash \{P\}\, \texttt{fun}\ f\ (\ ) = (\texttt{fuN}\ n\ (\ ) = c_n\ \texttt{in}\ c_f)\ \texttt{in}\ c_B\, \{Q\}\ \texttt{throws}\ \eta}$$

If we replace $(\texttt{fuN}\ n\ (\ ) = c_n\ \texttt{in}\ c_f)$ with $\texttt{bind/1}\ z\ \texttt{to}\ n\ \texttt{in}\ c_f$ or $\texttt{bind}\ z\ \texttt{to}\ f\ \texttt{in}\ c_f$, then we have to check all the elements c_f and $n(\ )$; or $f(\ )$;.

Finally, a reentrant function f, which is defined in a scope of c_B, contains a signal binding construct in its function body. Thus, if command c_B will call the function f then during its execution a function n could be invoked if a signal z arrives. We can paraphrase it by saying that execution of c_B could be interrupted by the function n. We know that the function n is non-reentrant, but we have a Ψ splitting mechanism in our linear type system that prevents sharing of a non-reentrant function by the main code and the handler. Please note that the function n could be used before the function call of f, or after it returns, in a sequential way as many times as needed. It is not important what the function n does. It is enough to know that n is non-reentrant, because splitting ensures that it may be used only once; either in the command body or in the signal handler.

9.2.2 Reentrancy Judgement for Non-reentrant Function Call

As we address function reentrancy with respect to itself (and not the case of interrupting of any non-reentrant function by another non-reentrant function), then $\emptyset; \Phi; \Psi \overset{R}{\vdash} f(\ );$ is rather strict.

$$\frac{n \in \mathrm{dom}(\Psi)}{\emptyset; \Phi; \Psi \overset{R}{\vdash} n(\ );} \text{(NFunCall-1)}$$

$$\frac{n \in \mathrm{dom}(\Psi) \qquad \forall z.z \in \mathrm{dom}(\Sigma) \wedge \Sigma(z) = f(\ ); \wedge f \neq n}{\Sigma; \Phi; \Psi \overset{R}{\vdash} n(\ );} \text{(NFunCall-2)}$$

We conjecture that it is enough to show that the function f is not bound to any signal that is stored in the signal context. Thus, for $\Sigma; \Phi; \Psi \overset{R}{\vdash} n(\ );$ we need an extra condition such that $\forall z.z \in \mathrm{dom}(\Sigma)$ that is the case that n is not bound to z: $\forall z.z \in \mathrm{dom}(\Sigma) \wedge \Sigma(z) = f(\ ); \wedge f \neq n$.

We need $\Sigma(z) = f(\ );$ for our rule to work. However, $\Sigma(z)$ was always used to access pre and post conditions of the signal handler; e.g.: $\Sigma(z) = I_z \triangleright I_z$ or even $\Sigma(z) = z : I_z \triangleright I_z$. If we check the signal binding rule, the name of the function is not stored in the signal context, only function's pre- and post- conditions.

Understanding of the relation between signals and functions influences the binding rule for the reentrancy judgement as well.

$$\frac{\Sigma, z : I_z \triangleright I_z; \Phi; \Psi_1 \overset{R}{\vdash} c_B \qquad \Sigma; \Phi; \Psi_2, n : I_z \triangleright I_z \overset{R}{\vdash} n(\ );}{\Sigma; \Phi; \Psi_1, \Psi_2, n : I_z \triangleright I_z \overset{R}{\vdash} \mathtt{bind}\, z\, \mathtt{to}\, n\, \mathtt{in}\, c_B}$$

According to the NFunCall-1 rule, Σ in $\Sigma; \Phi; \Psi_2, n : P_z \triangleright Q_z \overset{R}{\vdash} n(\ );$ should be nonempty to satisfy reentrancy judgement. Adoption of the NFunCall-2 rule, should allow non-empty Σ, if the signals bound in Σ do not call a function f. An explicit relation between signal and function names could require a new definition for the signal

context. However, we should check the influence of RLTS (in particular Ψ splitting) to this problem. We know that the function becomes a signal handler in two steps: a function definition and a signal binding. We also know that Ψ splitting ensures linear (affine) use of the non-reentrant functions; thus, use of the non-reentrant functions in the signal binding construct is safe. In other words, with RLTS (Ψ splitting) it is not possible to have a signal $z \in \mathrm{dom}(\Sigma)$ in $\Sigma; \Phi; \Psi_2, n : P_z \triangleright Q_z \overset{R}{\vdash} n(\,);$, such that z is bound to n.

Thus, NFunCall becomes similar to the RFunCall.

$$\frac{n \in \mathrm{dom}(\Psi)}{\Sigma; \Phi; \Psi \overset{R}{\vdash} n(\,);}\ (\textsc{NFunCall}) \qquad \frac{f \in \mathrm{dom}(\Phi)}{\Sigma; \Phi; \Psi \overset{R}{\vdash} f(\,);}\ (\textsc{RFunCall})$$

Examples

$$\frac{z_1 : I_1 \triangleright I_1, z_2 : I_2 \triangleright I_2; \emptyset; \emptyset \overset{R}{\vdash} c_B \qquad \dfrac{n_2 \in \mathrm{dom}(n_2 : I_2 \triangleright I_2)}{z_1 : I_1 \triangleright I_1; \emptyset; n_2 : I_2 \triangleright I_2 \overset{R}{\vdash} n_2(\,);}}{z_1 : I_1 \triangleright I_1; \emptyset; n_2 : I_2 \triangleright I_2 \overset{R}{\vdash} \mathtt{bind}\, z_2\, \mathtt{to}\, n_2\, \mathtt{in}\, c_B}$$

$$\frac{z_1 : I_1 \triangleright I_1; \emptyset; n_2 : I_2 \triangleright I_2 \overset{R}{\vdash} \mathtt{bind}\, z_2\, \mathtt{to}\, n_2\, \mathtt{in}\, c_B \qquad \dfrac{n_1 \in \mathrm{dom}(n_1 : I_1 \triangleright I_1)}{\emptyset; \emptyset; n_1 : I_1 \triangleright I_1 \overset{R}{\vdash} n_1(\,);}}{\emptyset; \emptyset; n_1 : I_1 \triangleright I_1, n_2 : I_2 \triangleright I_2 \overset{R}{\vdash} \mathtt{bind}\, z_1\, \mathtt{to}\, n_1\, \mathtt{in}\, (\mathtt{bind}\, z_2\, \mathtt{to}\, n_2\, \mathtt{in}\, c_B)}$$

9.2.3 Free Variables

In terms of predicates, quantification of the variables is the only syntactic construct that binds variables in our language (Definition 9.2.2). Thus, in an expression of the form:

$$\forall x.P$$

all occurrences of variable x in predicate P are bound. All other variable occurrences are free.

We define $\mathcal{FV}()$ with respect to the variables (identifiers) to which we can make an assignment. Thus, function, signal and exception names are excluded.

Definition 9.2.2 (Free variables; $\mathcal{FV}(P)$)

$$
\begin{aligned}
\mathcal{FV}(\forall x.P) &= \mathcal{FV}(P) \setminus \{x\} && \text{(quantification)} \\
\mathcal{FV}(\exists x.P) &= \mathcal{FV}(P) \setminus \{x\} && \text{(quantification)} \\
\mathcal{FV}(x) &= \{x\} && \text{(variable)} \\
\mathcal{FV}(n) &= \emptyset && \text{(integer)} \\
\mathcal{FV}(\texttt{true} \mid \texttt{false}) &= \emptyset && \text{(boolean)} \\
\mathcal{FV}(E_1 = E_2) &= \mathcal{FV}(E_1) \cup \mathcal{FV}(E_2) && \text{(equality)} \\
\mathcal{FV}(E_1 \leq E_2) &= \mathcal{FV}(E_1) \cup \mathcal{FV}(E_2) && \text{(less or equal)} \\
\mathcal{FV}(E_1 + E_2) &= \mathcal{FV}(E_1) \cup \mathcal{FV}(E_2) && \text{(addition)}
\end{aligned}
$$

Analogously to the case with predicates, quantification of the variables is one of the syntactic commands that binds variables in commands of our language (Definition 9.2.3). Another syntactic command that binds variables is a local variables construct.

Definition 9.2.3 (Free variables; $\mathcal{FV}(c)$)

$$
\begin{array}{rcll}
\mathcal{FV}(x) & = & \{x\} & \text{(variable)} \\
\mathcal{FV}(n) & = & \emptyset & \text{(integer)} \\
\mathcal{FV}(\texttt{true} \mid \texttt{false}) & = & \emptyset & \text{(boolean)} \\
\mathcal{FV}(x := E) & = & \mathcal{FV}(x) \cup \mathcal{FV}(E) & \text{(assignment)} \\
\mathcal{FV}(c_1; c_2) & = & \mathcal{FV}(c_1) \cup \mathcal{FV}(c_2) & \text{(seq comp)} \\
\mathcal{FV}(\texttt{while}\, E_B\, \texttt{do}\, c) & = & \mathcal{FV}(E_B) \cup \mathcal{FV}(c) & \text{(while)} \\
\mathcal{FV}(\texttt{if}\, E_B\, \texttt{then}\, c_1\, \texttt{else}\, c_2) & = & \mathcal{FV}(E_B) \cup \mathcal{FV}(c_1) & \text{(conditional)} \\
 & & \cup \mathcal{FV}(c_2) & \\
\mathcal{FV}(\texttt{throw}\, e) & = & \emptyset & \text{(exn throw)} \\
\mathcal{FV}(\texttt{try}\, c_B\, \texttt{handle}\, e\, \texttt{by}\, c_h) & = & \mathcal{FV}(c_B) \cup \mathcal{FV}(c_h) & \text{(exn handl)} \\
\mathcal{FV}(\texttt{local}\, y_1, \ldots, y_n\, \texttt{in}\, c) & = & \mathcal{FV}(c) \setminus \{y_1, \ldots, y_n\} & \text{(variables)} \\
\mathcal{FV}(\texttt{fun}\, f\ (\ ) = c_f\ \texttt{in}\, c_B) & = & \mathcal{FV}(c_f) \cup \mathcal{FV}(c_B) & \text{(fun def)} \\
\mathcal{FV}(f(\ );) & = & \emptyset & \text{(function call)} \\
\mathcal{FV}(\texttt{block}\, z\, \texttt{in}\, c) & = & \mathcal{FV}(c) & \text{(signal block)} \\
\mathcal{FV}(\texttt{bind}\, z\, \texttt{to}\, f_z\, \texttt{in}\, c_B) & = & \mathcal{FV}(c_B) & \text{(signal bind)} \\
\mathcal{FV}(\texttt{bind/1}\, z\, \texttt{to}\, f_z\, \texttt{in}\, c_B) & = & \mathcal{FV}(c_B) & \text{(signal bind)} \\
\mathcal{FV}(E_1 = E_2) & = & \mathcal{FV}(E_1) \cup \mathcal{FV}(E_2) & \text{(equality)} \\
\mathcal{FV}(E_1 \leq E_2) & = & \mathcal{FV}(E_1) \cup \mathcal{FV}(E_2) & \text{(less or equal)} \\
\mathcal{FV}(E_1 + E_2) & = & \mathcal{FV}(E_1) \cup \mathcal{FV}(E_2) & \text{(addition)}
\end{array}
$$

Definition 9.2.4 (Fresh name/variable) A fresh variable x in a particular scope is a variable such that is neither bound nor free in a given scope.

Whenever we define a new function (e.g.: $\texttt{fun}\ f\ (\ ) = c_f\ \texttt{in}\ c_B$) or install a new signal via signal binding (e.g.: $\texttt{bind}\, z\, \texttt{to}\, f_z\, \texttt{in}\, c_B$), it is implicitly assumed that f and z are fresh names (Definition 9.2.4).

All the constructs in our language are block structured. Therefore, if the function name f is bound, then there is no limitations in defining a new function with the same name f, as it will be defined in an inner scope. Thus, our language supports function and signal *overloading*. When the control flow leaves the inner scope with redefined function f, the corresponding function context will be automatically restored, as we adopt big-step operational semantics.

Before we can analyse the program using our logic, it is required to know which exceptions could be raised by the program. Thus, a signal context η should contain information about the exception that can be thrown by the (inside of the) program, so we can apply our logic rules. We cannot use functions for the signal binding, unless they are presented in the function context (means that they have been defined earlier). However, we do not limit `throw` to appear only inside of the `try-handle` block. Thus, one may write a program where a signal handler, that throws an exception, is defined in an outer scope, and the exception is caught in some (or not in any) inner scope. Thus, if signal arrives in a scope where a particular exception is not caught, an exception propagation will start. If there is no `handle` construct anywhere in the outer scope, the program will terminate with a raised exception. Thus, we cannot statically decide if an exception e from `throw` e is bound to the `try-handle` or free, as signals that may use `throw` e arrive nondeterministically.

Function and signal names are added only via corresponding rules; thus, whenever one meet function or signal name, then it is definitely bound via the function or the signal context. Therefore, Definition 9.2.4 for the function and signal names could be simplified to the next statement"is a variable such that is not bound", as there is no free function or signal name could appear.

9.2.4 Function Context Ψ Splitting

Definition 9.2.5 Given two non-reentrant function contexts Ψ_1 and Ψ_2, we define a partial operation "," as follows:

- If $\mathrm{dom}(\Psi_1) \cap \mathrm{dom}(\Psi_2) = \emptyset$, we write Ψ_1, Ψ_2 for $\Psi_1 \cup \Psi_2$.
- If $\mathrm{dom}(\Psi_1) \cap \mathrm{dom}(\Psi_2) \neq \emptyset$, then Ψ_1, Ψ_2 is undefined.

In the real-life implementations, exists a list of asynchronous safe functions that is safe to use inside of the signal handlers [56, 87, 91, 88, 9, 51]. In our approach, we keep track of functions that is unsafe to use inside of the signal handlers and call them non-reentrant.

As the reentrancy happens nondeterministically via a signal handler, then the "safety" of that reentrancy should be checked. If the functions that interrupt each other are not from the list of non-reentrant functions, then the reentrancy is safe. If the non-reentrant function is interrupted by any non-reentrant function, then such reentrancy is not safe. Reentrant safe functions do not call non-reentrant functions, or block all signals (to prevent potentially unsafe reentrancy) while the non-reentrant functions are called. Therefore, we introduce a non-reentrant function context splitting (Definition 9.2.5) as part of the reentrancy linear type system. It should be noted that Ψ splitting is orthogonal to Ω splitting. It is safe to call non-reentrant functions sequentially; thus, there is no Ψ splitting in most of the logic rules (e.g.: sequential composition, if-else structure, `while` structure, exception handling and etc). A logic rule for the signal binding (and handling at the same time) requires Ψ splitting as the signals arrive nondeterministically.

Let's consider the following examples in Figure 9.15 and Figure 9.16. In the former example, a nonreentrant function f should not be called from the body b and the handler h. In the latter example, it is safe to call a nonreentrant function in both b and h.

9.2.5 Logic Rules

Logic rules, which are summarised in Figure 9.17, Figure 9.18 and Figure 9.19, are discussed in detail throughout this section.

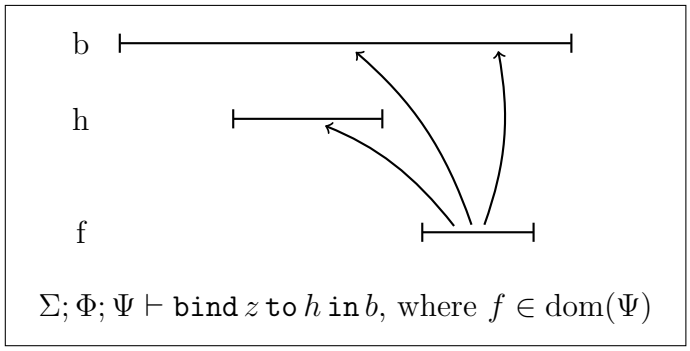

Figure 9.15: Ψ splitting in signal binding

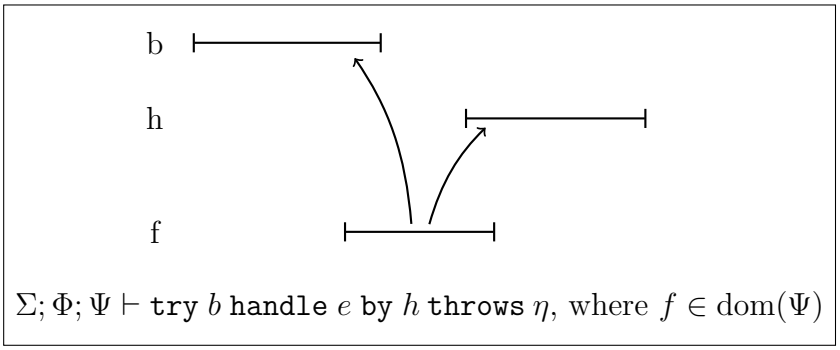

Figure 9.16: No Ψ splitting in exception handling

Function definition

$$\frac{\Sigma; \Phi; \Psi, n : P_n \triangleright Q_n \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta \qquad \Sigma; \Phi; \Psi, n : P_n \triangleright Q_n \vdash \{P_n\}\, c_n\, \{Q_n\}\ \texttt{throws}\ \eta}{\Sigma; \Phi; \Psi \vdash \{P\}\, \texttt{fuN}\ n\ (\,) = c_n\ \texttt{in}\ c_B\, \{Q\}\ \texttt{throws}\ \eta}$$

$$\frac{\Sigma; \Phi, f : P_f \triangleright Q_f; \Psi \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta \qquad \Sigma; \Phi, f : P_f \triangleright Q_f; \Psi \vdash \{P_f\}\, c_f\, \{Q_f\}\ \texttt{throws}\ \eta \qquad \Sigma; \Phi, f : P_f \triangleright Q_f; \Psi \overset{R}{\vdash} c_f}{\Sigma; \Phi; \Psi \vdash \{P\}\, \texttt{fun}\ f\ (\,) = c_f\ \texttt{in}\ c_B\, \{Q\}\ \texttt{throws}\ \eta}$$

Without a reentrancy judgement, we would need two separate rules to cover both cases when the function could be called reentrant (empty signal context or non-reentrant func-

$$\frac{\Sigma;\Phi;\Psi, n : P_n \triangleright Q_n \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta \qquad \Sigma;\Phi;\Psi, n : P_n \triangleright Q_n \vdash \{P_n\}\, c_n\, \{Q_n\}\ \texttt{throws}\ \eta}{\Sigma;\Phi;\Psi \vdash \{P\}\, \texttt{fuN}\ n\ (\,) = c_n\ \texttt{in}\ c_B\, \{Q\}\ \texttt{throws}\ \eta}\ (\textsc{NFDef})$$

$$\frac{\Sigma;\Phi, f : P_f \triangleright Q_f;\Psi \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta \qquad \Sigma;\Phi, f : P_f \triangleright Q_f;\Psi \vdash \{P_f\}\, c_f\, \{Q_f\}\ \texttt{throws}\ \eta \qquad \Sigma;\Phi, f : P_f \triangleright Q_f;\Psi \overset{R}{\vdash} c_f}{\Sigma;\Phi;\Psi \vdash \{P\}\, \texttt{fun}\ f\ (\,) = c_f\ \texttt{in}\ c_B\, \{Q\}\ \texttt{throws}\ \eta}\ (\textsc{RFDef})$$

$$\frac{\Phi(f) = P\ \triangleright\ Q}{\Sigma;\Phi;\Psi \vdash \{P\}\, f(\,);\, \{Q\}\ \texttt{throws}\ \eta}\ (\textsc{RFunCall})$$

$$\frac{\Psi(f) = P\ \triangleright\ Q}{\Sigma;\Phi;\Psi \vdash \{P\}\, f(\,);\, \{Q\}\ \texttt{throws}\ \eta}\ (\textsc{NFunCall})$$

$$\frac{\Sigma, z : P_z \triangleright Q_z;\Phi, f : P_z \triangleright Q_z;\Psi_1 \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta \qquad \Sigma;\Phi, f : P_z \triangleright Q_z;\Psi_2 \vdash \{P_z\}\, f(\,);\, \{Q_z\}\ \texttt{throws}\ \eta \qquad \Sigma\ \textbf{stable}\ P_z \triangleright Q_z}{\Sigma;\Phi, f : P_z \triangleright Q_z;\Psi_1, \Psi_2 \vdash \{P \wedge P_z\}\, \texttt{bind/1}\ z\ \texttt{to}\ f\ \texttt{in}\ c_B\, \{Q \wedge (P_z \vee Q_z)\}\ \texttt{throws}\ \eta}\ (\textsc{SOR})$$

$$\frac{\Sigma, z : I_z \triangleright I_z;\Phi, f : I_z \triangleright I_z;\Psi_1 \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta \qquad \Sigma;\Phi, f : I_z \triangleright I_z;\Psi_2 \vdash \{I_z\}\, f(\,);\, \{I_z\}\ \texttt{throws}\ \eta \qquad \Sigma\ \textbf{stable}\ I_z}{\Sigma;\Phi, f : I_z \triangleright I_z;\Psi_1, \Psi_2 \vdash \{P \wedge I_z\}\, \texttt{bind}\ z\ \texttt{to}\ f\ \texttt{in}\ c_B\, \{Q \wedge I_z\}\ \texttt{throws}\ \eta}\ (\textsc{SBR})$$

Figure 9.17: Hoare logic rules - Part 1

$$\frac{\begin{array}{c}\Sigma, z : P_z \triangleright Q_z; \Phi; \Psi_1 \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta \\ \Sigma; \Phi; \Psi_2, f : P_z \triangleright Q_z \vdash \{P_z\}\, f(\,);\, \{Q_z\}\ \texttt{throws}\ \eta \\ \Sigma\ \textbf{stable}\ P_z \triangleright Q_z\end{array}}{\Sigma; \Phi; \Psi_1, \Psi_2, f : P_z \triangleright Q_z \vdash \{P \wedge P_z\}\ \texttt{bind/1}\ z\ \texttt{to}\ f\ \texttt{in}\ c_B\, \{Q \wedge (P_z \vee Q_z)\}\ \texttt{throws}\ \eta}$$

$$\frac{\begin{array}{c}\Sigma, z : I_z \triangleright I_z; \Phi; \Psi_1 \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta \\ \Sigma; \Phi; \Psi_2, f : I_z \triangleright I_z \vdash \{I_z\}\, f(\,);\, \{I_z\}\ \texttt{throws}\ \eta \\ \Sigma\ \textbf{stable}\ I_z\end{array}}{\Sigma; \Phi; \Psi_1, \Psi_2, f : I_z \triangleright I_z \vdash \{P \wedge I_z\}\ \texttt{bind}\ z\ \texttt{to}\ f\ \texttt{in}\ c_B\, \{Q \wedge I_z\}\ \texttt{throws}\ \eta}$$

$$\frac{\begin{array}{c}\Sigma; \Phi; \Psi \vdash \{P\}\, c\, \{Q\}\ \texttt{throws}\ \eta \\ P\ \textbf{stable}\ P_z \triangleright Q_z \qquad Q\ \textbf{stable}\ P_z \triangleright Q_z \qquad \eta\ \textbf{stable}\ P_z \triangleright Q_z\end{array}}{\Sigma, z : P_z \triangleright Q_z; \Phi; \Psi \vdash \{P\}\ \texttt{block}\ z\ \texttt{in}\ c\, \{Q\}\ \texttt{throws}\ \eta}$$

Figure 9.18: Hoare logic rules - Part 2

tion context). One may suggest the following rules:

$$\frac{\begin{array}{c}\Sigma; \Phi, f : P_f \triangleright Q_f; \Psi \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta \\ \emptyset; \Phi, f : P_f \triangleright Q_f; \Psi \vdash \{P_f\}\, c_f\, \{Q_f\}\ \texttt{throws}\ \eta\end{array}}{\Sigma; \Phi; \Psi \vdash \{P_B\}\ \texttt{fun}\ f\ (\,) = c_f\ \texttt{in}\ c_B\, \{Q_B\}\ \texttt{throws}\ \eta}$$

$$\frac{\begin{array}{c}\Sigma; \Phi, f : P_f \triangleright Q_f; \Psi \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta \\ \Sigma; \Phi, f : P_f \triangleright Q_f; \emptyset \vdash \{P_f\}\, c_f\, \{Q_f\}\ \texttt{throws}\ \eta\end{array}}{\Sigma; \Phi; \Psi \vdash \{P_B\}\ \texttt{fun}\ f\ (\,) = c_f\ \texttt{in}\ c_B\, \{Q_B\}\ \texttt{throws}\ \eta}$$

However, a command c_f could be non-atomic; thus, the emptiness of the signal context should be checked higher in the derivation tree of the c_f.

$$\frac{\Sigma;\Phi;\Psi \vdash \{P_1\}\, c_1\, \{P_2\}\ \texttt{throws}\ \eta \qquad \Sigma;\Phi;\Psi \vdash \{P_2\}\, c_2\, \{P_3\}\ \texttt{throws}\ \eta}{\Sigma;\Phi;\Psi \vdash \{P_1\}\, c_1\, ;\, c_2\, \{P_3\}\ \texttt{throws}\ \eta}\ (\textsc{Seq})$$

$$\frac{\begin{array}{c}\{P\}\, a\, \{Q\} \\ (P \triangleright Q)\ \textbf{stable}\ \Sigma \qquad \eta\ \textbf{stable}\ \Sigma \\ \forall z.\Sigma(z) = z : \forall \alpha.P_z \triangleright Q_z \qquad \exists E.Q \implies P_z\,[\,\alpha \mapsto E\,]\end{array}}{\Sigma;\Phi;\Psi \vdash \{P\}\, a\, \{Q\}\ \texttt{throws}\ \eta}\ (\textsc{Atomic})$$

$$\frac{\begin{array}{c}\{P\}\, x := E'\, \{Q\} \\ (P \triangleright Q)\ \textbf{stable}\ \Sigma \qquad \eta\ \textbf{stable}\ \Sigma \\ \forall z.\Sigma(z) = z : \forall \alpha.P_z \triangleright Q_z \qquad \exists E.Q \implies P_z\,[\,\alpha \mapsto E\,]\end{array}}{\Sigma;\Phi;\Psi \vdash \{P\}\, x := E'\, \{Q\}\ \texttt{throws}\ \eta}\ (\textsc{Assignment})$$

$$\frac{\eta\ \textbf{stable}\ \Sigma \qquad Q\ \textbf{stable}\ \Sigma}{\Sigma;\Phi;\Psi \vdash \{\eta(e_j)\}\ \texttt{throw}\ e_j\, \{Q\}\ \texttt{throws}\ \eta}\ (\textsc{Throw})$$

$$\frac{\Sigma;\Phi;\Psi \vdash \{P\}\, c_B\, \{Q_b\}\ \texttt{throws}\ \eta, e_k : Q_k \qquad \Sigma;\Phi;\Psi \vdash \{Q_k\}\, c_h\, \{Q_h\}\ \texttt{throws}\ \eta}{\Sigma;\Phi;\Psi \vdash \{P\}\ \texttt{try}\ c_B\ \texttt{handle}\ e_k\ \texttt{by}\ c_h\, \{Q_b \vee Q_h\}\ \texttt{throws}\ \eta}$$

$$\frac{\Sigma;\Phi;\Psi \vdash \{I \wedge E_B\}\, c\, \{I\}\ \texttt{throws}\ \eta \qquad \neg E_B\ \textbf{stable}\ \Sigma}{\Sigma;\Phi;\Psi \vdash \{I\}\ \texttt{while}\ E_B\ \texttt{do}\ c\, \{I \wedge \neg E_B\}\ \texttt{throws}\ \eta}\ (\textsc{WhileRule})$$

$$\frac{\Sigma;\Phi;\Psi \vdash \{E_B \wedge P\}\, c_1\, \{Q\}\ \texttt{throws}\ \eta \qquad \Sigma;\Phi;\Psi \vdash \{\neg E_B \wedge P\}\, c_2\, \{Q\}\ \texttt{throws}\ \eta}{\Sigma;\Phi;\Psi \vdash \{P\}\ \texttt{if}\ E_B\ \texttt{then}\ c_1\ \texttt{else}\ c_2\, \{Q\}\ \texttt{throws}\ \eta}$$

$$\frac{\begin{array}{c}\Sigma;\Phi;\Psi \vdash \{P\}\, c\, \{Q\}\ \texttt{throws}\ \eta \quad P' \Rightarrow P \quad Q \Rightarrow Q' \quad \eta \Rightarrow \eta' \\ P'\ \textbf{stable}\ \Sigma \qquad Q'\ \textbf{stable}\ \Sigma \qquad \eta'\ \textbf{stable}\ \Sigma\end{array}}{\Sigma;\Phi;\Psi \vdash \{P'\}\, c\, \{Q'\}\ \texttt{throws}\ \eta'}\ (\textsc{Conseq})$$

$$\frac{\Sigma;\Phi;\Psi \vdash \{P_1\}\, c\, \{Q_1\}\ \texttt{throws}\ \eta_1 \qquad \Sigma;\Phi;\Psi \vdash \{P_2\}\, c\, \{Q_2\}\ \texttt{throws}\ \eta_2}{\Sigma;\Phi;\Psi \vdash \{P_1 \wedge P_2\}\, c\, \{Q_1 \wedge Q_2\}\ \texttt{throws}\ \eta_1 \wedge \eta_2}\ (\textsc{Conj})$$

$$\frac{\Sigma;\Phi;\Psi \vdash \{P\}\, c\, \{Q\} \qquad y_j \notin \texttt{FV}(P) \qquad y_j \notin \texttt{FV}(Q)}{\Sigma;\Phi;\Psi \vdash \{P\}\ \texttt{local}\ y_1, y_2, \ldots\ \texttt{in}\ c\, \{Q\}}\ (\textsc{LocVar})$$

Figure 9.19: Hoare logic rules - Part 3

Function call

$$\frac{\Phi(f) = P \triangleright Q}{\Sigma; \Phi; \Psi \vdash \{P\}\, f(\,);\, \{Q\}\, \texttt{throws}\, \eta} \qquad \frac{\Psi(f) = P \triangleright Q}{\Sigma; \Phi; \Psi \vdash \{P\}\, f(\,);\, \{Q\}\, \texttt{throws}\, \eta}$$

Sequential composition

$$\frac{\Sigma; \Phi; \Psi \vdash \{P\}\, c_1\, \{P'\}\, \texttt{throws}\, \eta \qquad \Sigma; \Phi; \Psi \vdash \{P'\}\, c_2\, \{Q\}\, \texttt{throws}\, \eta}{\Sigma; \Phi; \Psi \vdash \{P\}\, c_1; c_2\, \{Q\}\, \texttt{throws}\, \eta}$$

Atomic commands

$$\{P\}\, a\, \{Q\}$$

$$(P \triangleright Q)\ \textbf{stable}\ \Sigma \qquad \eta\ \textbf{stable}\ \Sigma$$

$$\frac{\forall z.\Sigma(z) = z : \forall \alpha.P_z \triangleright Q_z \qquad \exists E.Q \implies P_z\,[\,\alpha \mapsto E\,]}{\Sigma; \Phi; \Psi \vdash \{P\}\, a\, \{Q\}\, \texttt{throws}\, \eta}\ (\textsc{Atomic})$$

Assignment

$$\{P\}\, x := E'\, \{Q\}$$

$$(P \triangleright Q)\ \textbf{stable}\ \Sigma \qquad \eta\ \textbf{stable}\ \Sigma$$

$$\frac{\forall z.\Sigma(z) = z : \forall \alpha.P_z \triangleright Q_z \qquad \exists E.Q \implies P_z\,[\,\alpha \mapsto E\,]}{\Sigma; \Phi; \Psi \vdash \{P\}\, x := E'\, \{Q\}\, \texttt{throws}\, \eta}\ (\textsc{Assignment})$$

Signal binding

One may suggest that by checking stability on the level of function definition, we may remove stability checks from the signal binding. However, the only significant difference would be more strict conditions for the functions, as potentially all of them might become signal handlers.

There is a design question, should we include $f \in \mathrm{dom}(\Phi)$ in the logic rules for signal binding? If one refer to the current convention regarding signals and functions, then only reentrant functions could be safely called from the signal handlers. Thus, including $f \in \mathrm{dom}(\Phi)$ sounds reasonable. However, in our logic we may weaken the limitations and still show that the program will run safely. The trick is in our approach, where we focus on the non-reentrant functions. We do not limit non-reentrant functions to be used in signal handlers, as it might be the case that the non-reentrant functions are not used in a block of code, or if they are used then the signal context is empty (all signals are blocked). Actually, the first step of defence is the function addition to the corresponding context. In other words, functions are checked for the reentrancy safety while lifted to the function context. Functions from Φ are safe; and functions from Ψ require an extra attention to ensure safety of the programs. Ψ splitting, as part of the Reentrant Linear Type System, takes care of the non-reentrant functions in signal handling. That could be classified as the second step of defence. Thus, including $f \in \mathrm{dom}(\Phi)$ is redundant. Further discussion is given in Section 9.2.7.

Updated rules for the signal binding are presented in Figure 9.20. There are four of them, as the standard rules for the one-shot and the persistent signal binding are extended with the notions of non-reentrant functions and Ψ splitting.

Blocking

$$\Sigma; \Phi; \Psi \vdash \{P\}\, c\, \{Q\}\ \texttt{throws}\ \eta$$

$$\frac{P\ \textbf{stable}\ P_z \triangleright Q_z \qquad Q\ \textbf{stable}\ P_z \triangleright Q_z \qquad \eta\ \textbf{stable}\ P_z \triangleright Q_z}{\Sigma, z : P_z \triangleright Q_z; \Phi; \Psi \vdash \{P\}\, \texttt{block}\, z\, \texttt{in}\, c\, \{Q\}\ \texttt{throws}\ \eta}$$

$$\frac{\begin{array}{c}\Sigma, z : P_z \triangleright Q_z; \Phi, f : P_z \triangleright Q_z; \Psi_1 \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta \\ \Sigma; \Phi, f : P_z \triangleright Q_z; \Psi_2 \vdash \{P_z\}\, f(\,);\, \{Q_z\}\ \texttt{throws}\ \eta \\ \Sigma\ \textbf{stable}\ P_z \triangleright Q_z\end{array}}{\Sigma; \Phi, f : P_z \triangleright Q_z; \Psi_1, \Psi_2 \vdash \{P \wedge P_z\}\, \texttt{bind/1}\ z\ \texttt{to}\ f\ \texttt{in}\ c_B\, \{Q \wedge (P_z \vee Q_z)\}\ \texttt{throws}\ \eta}$$

$$\frac{\begin{array}{c}\Sigma, z : I_z \triangleright I_z; \Phi, f : I_z \triangleright I_z; \Psi_1 \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta \\ \Sigma; \Phi, f : I_z \triangleright I_z; \Psi_2 \vdash \{I_z\}\, f(\,);\, \{I_z\}\ \texttt{throws}\ \eta \\ \Sigma\ \textbf{stable}\ I_z\end{array}}{\Sigma; \Phi, f : I_z \triangleright I_z; \Psi_1, \Psi_2 \vdash \{P \wedge I_z\}\, \texttt{bind}\ z\ \texttt{to}\ f\ \texttt{in}\ c_B\, \{Q \wedge I_z\}\ \texttt{throws}\ \eta}$$

$$\frac{\begin{array}{c}\Sigma, z : P_z \triangleright Q_z; \Phi; \Psi_1 \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta \\ \Sigma; \Phi; \Psi_2, f : P_z \triangleright Q_z \vdash \{P_z\}\, f(\,);\, \{Q_z\}\ \texttt{throws}\ \eta \\ \Sigma\ \textbf{stable}\ P_z \triangleright Q_z\end{array}}{\Sigma; \Phi; \Psi_1, \Psi_2, f : P_z \triangleright Q_z \vdash \{P \wedge P_z\}\, \texttt{bind/1}\ z\ \texttt{to}\ f\ \texttt{in}\ c_B\, \{Q \wedge (P_z \vee Q_z)\}\ \texttt{throws}\ \eta}$$

$$\frac{\begin{array}{c}\Sigma, z : I_z \triangleright I_z; \Phi; \Psi_1 \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta \\ \Sigma; \Phi; \Psi_2, f : I_z \triangleright I_z \vdash \{I_z\}\, f(\,);\, \{I_z\}\ \texttt{throws}\ \eta \\ \Sigma\ \textbf{stable}\ I_z\end{array}}{\Sigma; \Phi; \Psi_1, \Psi_2, f : I_z \triangleright I_z \vdash \{P \wedge I_z\}\, \texttt{bind}\ z\ \texttt{to}\ f\ \texttt{in}\ c_B\, \{Q \wedge I_z\}\ \texttt{throws}\ \eta}$$

Figure 9.20: Signal Binding Rules

Exception binding and throwing

$$\frac{\eta \textbf{ stable } \Sigma \qquad Q \textbf{ stable } \Sigma}{\Sigma; \Phi; \Psi \vdash \{\eta(e_j)\}\, \texttt{throw}\, e_j\, \{Q\}\, \texttt{throws}\, \eta}$$

$$\frac{\Sigma; \Phi; \Psi \vdash \{P\}\, c_B\, \{Q_b\}\, \texttt{throws}\, \eta, e_k : Q_k \quad \Sigma; \Phi; \Psi \vdash \{Q_k\}\, c_h\, \{Q_h\}\, \texttt{throws}\, \eta}{\Sigma; \Phi; \Psi \vdash \{P\}\, \texttt{try}\, c_B\, \texttt{handle}\, e_k\, \texttt{by}\, c_h\, \{Q_b \vee Q_h\}\, \texttt{throws}\, \eta}$$

Repetitive `while` construct

$$\frac{\Sigma; \Phi; \Psi \vdash \{I \wedge E_B\}\, c\, \{I\}\, \texttt{throws}\, \eta \qquad \neg E_B \textbf{ stable } \Sigma}{\Sigma; \Phi; \Psi \vdash \{I\}\, \texttt{while}\, E_B\, \texttt{do}\, c\, \{I \wedge \neg E_B\}\, \texttt{throws}\, \eta}$$

Conditional `if-else` construct

$$\frac{\Sigma; \Phi; \Psi \vdash \{E_B \wedge P\}\, c_1\, \{Q\}\, \texttt{throws}\, \eta \quad \Sigma; \Phi; \Psi \vdash \{\neg E_B \wedge P\}\, c_2\, \{Q\}\, \texttt{throws}\, \eta}{\Sigma; \Phi; \Psi \vdash \{P\}\, \texttt{if}\, E_B\, \texttt{then}\, c_1\, \texttt{else}\, c_2\, \{Q\}\, \texttt{throws}\, \eta}$$

Rule of consequence

$$\Sigma; \Phi; \Psi \vdash \{P\}\, c\, \{Q\}\, \texttt{throws}\, \eta \quad P' \Rightarrow P \quad Q \Rightarrow Q' \quad \eta \Rightarrow \eta'$$

$$\frac{P' \textbf{ stable } \Sigma \qquad Q' \textbf{ stable } \Sigma \qquad \eta' \textbf{ stable } \Sigma}{\Sigma; \Phi; \Psi \vdash \{P'\}\, c\, \{Q'\}\, \texttt{throws}\, \eta'}$$

Rule of conjunction

$$\frac{\Sigma; \Phi; \Psi \vdash \{P_1\}\, c\, \{Q_1\}\, \texttt{throws}\, \eta_1 \qquad \Sigma; \Phi; \Psi \vdash \{P_2\}\, c\, \{Q_2\}\, \texttt{throws}\, \eta_2}{\Sigma; \Phi; \Psi \vdash \{P_1 \wedge P_2\}\, c\, \{Q_1 \wedge Q_2\}\, \texttt{throws}\, \eta_1 \wedge \eta_2}$$

Local variables

Definition 9.2.2 for the free variables is given in Section 9.2.3.

$$\frac{\Sigma; \Phi; \Psi \vdash \{P\}\, c\, \{Q\} \qquad y_j \notin \texttt{FV}(P) \qquad y_j \notin \texttt{FV}(Q)}{\Sigma; \Phi; \Psi \vdash \{P\}\, \texttt{local}\, y_1, y_2, \ldots\, \texttt{in}\, c\, \{Q\}}$$

9.2.6 Implicit Versus Explicit Stability Assumptions

As signal handlers are functions, potentially, stability assumptions could be already embedded into the judgments.

$$\frac{\Sigma; f : I \triangleright I; \emptyset \vdash \{P \wedge I\}\,(\texttt{bind}\,z\,\texttt{to}\,f\,\texttt{in}\,c_B)\,\{Q \wedge I\}\,\texttt{throws}\,\eta \qquad \Sigma; f : I \triangleright I; \emptyset \vdash \{I\}\,(a_1; a_2)\,\{I\}\,\texttt{throws}\,\eta \qquad \Sigma; f : I \triangleright I; \emptyset \overset{R}{\vdash} (a_1; a_2)}{\Sigma; \emptyset; \emptyset \vdash \{P \wedge I\}\,\texttt{fun}\,f\,(\,) = (a_1; a_2)\,\texttt{in}\,(\texttt{bind}\,z\,\texttt{to}\,f\,\texttt{in}\,c_B)\,\{Q \wedge I\}\,\texttt{throws}\,\eta}$$

$$\frac{\dfrac{}{\Sigma; f : I \triangleright I; \emptyset \overset{R}{\vdash} a_1} \qquad \dfrac{}{\Sigma; f : I \triangleright I; \emptyset \overset{R}{\vdash} a_2}}{\Sigma; f : I \triangleright I; \emptyset \overset{R}{\vdash} (a_1; a_2)}$$

$$\frac{\dfrac{\{I\}\,a_1\,\{P'\} \qquad I\ \textbf{stable}\ \Sigma \qquad P'\ \textbf{stable}\ \Sigma}{\Sigma; f : I \triangleright I; \emptyset \vdash \{I\}\,a_1\,\{P'\}\,\texttt{throws}\,\eta} \qquad \dfrac{\{P'\}\,a_2\,\{I\} \qquad P'\ \textbf{stable}\ \Sigma \qquad I\ \textbf{stable}\ \Sigma}{\Sigma; f : I \triangleright I; \emptyset \vdash \{P'\}\,a_2\,\{I\}\,\texttt{throws}\,\eta}}{\Sigma; f : I \triangleright I; \emptyset \vdash \{I\}\,(a_1; a_2)\,\{I\}\,\texttt{throws}\,\eta}$$

$$\frac{\dfrac{\Sigma, z : I \triangleright I; f : I \triangleright I; \emptyset \vdash \{P\}\,c_B\,\{Q\}\,\texttt{throws}\,\eta \qquad f : I \triangleright I(f) = I \ \triangleright\ I}{\Sigma; f : I \triangleright I; \emptyset \vdash \{I\}\,f(\,);\,\{I\}\,\texttt{throws}\,\eta} \qquad \Sigma\ \textbf{stable}\ I}{\Sigma; f : I \triangleright I; \emptyset \vdash \{P \wedge I\}\,(\texttt{bind}\,z\,\texttt{to}\,f\,\texttt{in}\,c_B)\,\{Q \wedge I\}\,\texttt{throws}\,\eta}$$

As Σ **stable** $(I \triangleright P')$ and Σ **stable** $(P' \triangleright I)$ are checked when the function is added. Functions are used as signal handlers. Thus, the question is can we remove Σ **stable** I from the binding rule.

The case for the one-shot signal binding is analogous:

$$\dfrac{\Sigma; f : P_z \triangleright Q_z; \emptyset \vdash \{P \wedge P_z\}\,(\texttt{bind}\,z\,\texttt{to}\,f\,\texttt{in}\,c_B)\,\{Q \wedge (P_z \vee Q_z)\}\,\texttt{throws}\,\eta \qquad \Sigma; f : P_z \triangleright Q_z; \emptyset \vdash \{P_z\}\,(a_1; a_2)\,\{Q_z\}\,\texttt{throws}\,\eta \qquad \Sigma; f : P_z \triangleright Q_z; \emptyset \overset{R}{\vdash} (a_1; a_2)}{\Sigma; \emptyset; \emptyset \vdash \{P \wedge P_z\}\,\texttt{fun}\,f\,(\,) = (a_1; a_2)\,\texttt{in}\,(\texttt{bind/1}\,z\,\texttt{to}\,f\,\texttt{in}\,c_B)\,\{Q \wedge (P_z \vee Q_z)\}\,\texttt{throws}\,\eta}$$

$$\dfrac{\dfrac{}{\Sigma; f : P_z \triangleright Q_z; \emptyset \overset{R}{\vdash} a_1} \qquad \dfrac{}{\Sigma; f : P_z \triangleright Q_z; \emptyset \overset{R}{\vdash} a_2}}{\Sigma; f : P_z \triangleright Q_z; \emptyset \overset{R}{\vdash} (a_1; a_2)}$$

$$\dfrac{\dfrac{\{P_z\}\,a_1\,\{P'\} \qquad P_z\ \textbf{stable}\ \Sigma \qquad P'\ \textbf{stable}\ \Sigma \qquad \Sigma\ \textbf{stable}\ (P_z \triangleright P') \qquad \eta\ \textbf{stable}\ \Sigma}{\Sigma; f : P_z \triangleright Q_z; \emptyset \vdash \{P_z\}\,a_1\,\{P'\}\,\texttt{throws}\,\eta} \qquad \dfrac{\{P'\}\,a_2\,\{Q_z\} \qquad P'\ \textbf{stable}\ \Sigma \qquad Q_z\ \textbf{stable}\ \Sigma \qquad \Sigma\ \textbf{stable}\ (P' \triangleright Q_z) \qquad \eta\ \textbf{stable}\ \Sigma}{\Sigma; f : P_z \triangleright Q_z; \emptyset \vdash \{P'\}\,a_2\,\{Q_z\}\,\texttt{throws}\,\eta}}{\Sigma; f : P_z \triangleright Q_z; \emptyset \vdash \{P_z\}\,(a_1; a_2)\,\{Q_z\}\,\texttt{throws}\,\eta}$$

$$\dfrac{\dfrac{\Sigma, z : P_z \triangleright Q_z; f : P_z \triangleright Q_z; \emptyset \vdash \{P\}\,c_B\,\{Q\}\,\texttt{throws}\,\eta \qquad f : P_z \triangleright Q_z(f) = P_z\ \triangleright\ Q_z}{\Sigma; f : P_z \triangleright Q_z; \emptyset \vdash \{P_z\}\,f(\,);\,\{Q_z\}\,\texttt{throws}\,\eta} \qquad \Sigma\ \textbf{stable}\ P_z \triangleright Q_z}{\Sigma; f : P_z \triangleright Q_z; \emptyset \vdash \{P \wedge P_z\}\,(\texttt{bind}\,z\,\texttt{to}\,f\,\texttt{in}\,c_B)\,\{Q \wedge (P_z \vee Q_z)\}\,\texttt{throws}\,\eta}$$

Our assumption works in both cases; however, we need to check if it holds when the rule of consequence is used.

$$\dfrac{\Sigma; \Phi; \Psi \vdash \{P\}\,c\,\{Q\}\,\texttt{throws}\,\eta \quad P' \Rightarrow P \quad Q \Rightarrow Q' \quad \eta \Rightarrow \eta' \qquad P'\ \textbf{stable}\ \Sigma \qquad Q'\ \textbf{stable}\ \Sigma \qquad \eta'\ \textbf{stable}\ \Sigma}{\Sigma; \Phi; \Psi \vdash \{P'\}\,c\,\{Q'\}\,\texttt{throws}\,\eta'}$$

So let's consider the next example:

$$\frac{\begin{array}{c}\Sigma; f : I \triangleright I; \emptyset \vdash \{P \wedge I\}\, (\texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, c_B)\, \{Q \wedge I\}\, \texttt{throws}\, \eta \\ \Sigma; f : I \triangleright I; \emptyset \vdash \{I\}\, (a_1; a_2)\, \{I\}\, \texttt{throws}\, \eta \\ \Sigma; f : I \triangleright I; \emptyset \overset{R}{\vdash} (a_1; a_2)\end{array}}{\Sigma; \emptyset; \emptyset \vdash \{P \wedge I\}\, \texttt{fun}\, f\ (\ ) = (a_1; a_2)\ \texttt{in}\ (\texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, c_B)\, \{Q \wedge I\}\, \texttt{throws}\, \eta}$$

$$\frac{\dfrac{}{\Sigma; f : I \triangleright I; \emptyset \overset{R}{\vdash} a_1} \qquad \dfrac{}{\Sigma; f : I \triangleright I; \emptyset \overset{R}{\vdash} a_2}}{\Sigma; f : I \triangleright I; \emptyset \overset{R}{\vdash} (a_1; a_2)}$$

$$\frac{\dfrac{\begin{array}{c}\dfrac{\begin{array}{c}\{P_1\}\, a_1\, \{Q_1\} \\ P_1\ \textbf{stable}\ \Sigma \quad Q_1\ \textbf{stable}\ \Sigma \\ \Sigma\ \textbf{stable}\ (P_1 \triangleright Q_1) \quad \eta\ \textbf{stable}\ \Sigma\end{array}}{\Sigma; f : I \triangleright I; \emptyset \vdash \{P_1\}\, a_1\, \{Q_1\}\, \texttt{throws}\, \eta} \\ I \Rightarrow P_1 \quad Q_1 \Rightarrow P_2 \quad \eta \Rightarrow \eta \\ I\ \textbf{stable}\ \Sigma \quad P_2\ \textbf{stable}\ \Sigma \quad \eta\ \textbf{stable}\ \Sigma\end{array}}{\Sigma; f : I \triangleright I; \emptyset \vdash \{I\}\, a_1\, \{P_2\}\, \texttt{throws}\, \eta} \quad \dfrac{\begin{array}{c}\dfrac{\begin{array}{c}\{P_3\}\, a_2\, \{Q_3\} \\ P_3\ \textbf{stable}\ \Sigma \quad Q_3\ \textbf{stable}\ \Sigma \\ \Sigma\ \textbf{stable}\ (P_3 \triangleright Q_3) \quad \eta\ \textbf{stable}\ \Sigma\end{array}}{\Sigma; f : I \triangleright I; \emptyset \vdash \{P_3\}\, a_2\, \{Q_3\}\, \texttt{throws}\, \eta} \\ P_2 \Rightarrow P_3 \quad Q_3 \Rightarrow I \quad \eta \Rightarrow \eta \\ P_2\ \textbf{stable}\ \Sigma \quad I\ \textbf{stable}\ \Sigma \quad \eta\ \textbf{stable}\ \Sigma\end{array}}{\Sigma; f : I \triangleright I; \emptyset \vdash \{P_2\}\, a_2\, \{I\}\, \texttt{throws}\, \eta}}{\Sigma; f : I \triangleright I; \emptyset \vdash \{I\}\, (a_1; a_2)\, \{I\}\, \texttt{throws}\, \eta}$$

$$\frac{\begin{array}{c}\Sigma, z : I \triangleright I; f : I \triangleright I; \emptyset \vdash \{P\}\, c_B\, \{Q\}\, \texttt{throws}\, \eta \\ \dfrac{f : I \triangleright I(f) = I\ \triangleright\ I}{\Sigma; f : I \triangleright I; \emptyset \vdash \{I\}\, f(\ );\ \{I\}\, \texttt{throws}\, \eta} \\ \Sigma\ \textbf{stable}\ I\end{array}}{\Sigma; f : I \triangleright I; \emptyset \vdash \{P \wedge I\}\, (\texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, c_B)\, \{Q \wedge I\}\, \texttt{throws}\, \eta}$$

Thus, from this example we can conclude that Σ **stable** $(P_1 \triangleright Q_1)$ and Σ **stable** $(P_3 \triangleright Q_3)$ do not trivially imply Σ **stable** I. We know that $Q_1 \Rightarrow P_2$, $P_2 \Rightarrow P_3$ and $Q_3 \Rightarrow I$. By transitivity, we can replace Q_1 with P_3. We can also replace Q_3 with I. Thus, we get Σ **stable** $(P_1 \triangleright P_3)$ and Σ **stable** $(P_3 \triangleright I)$. However, $I \Rightarrow P_1$ does not imply $P_1 \Rightarrow I$; thus, we can not replace P_1 with I in Σ **stable** $(P_1 \triangleright P_3)$.

9.2.7 Reentrant (Φ) Versus Nonreentrant (Ψ) Functions

During the step of a function definition, we decide whether a particular function is reentrant-safe or not. According to this decision, the function is added to a corresponding function context. Thus, at any point of a tree derivation we know that the functions from Φ are reentrant-safe and from Ψ are not.

Actually, we omit '-safe' ending, in the context of this thesis. Thus, unless it is clearly stated that we are talking about the sole fact of re-entering, under reentrancy we understand that function (or handler) is reentrant safe.

To enforce the policy, currently advised in the real-life implementations, we may limit signal binding to the functions from the reentrant context only. This limitation could be addressed in the program logic even without use of the reentrancy linear type system. Thus, for the binding command of this form $\texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, c_B$ we need to add an extra check, such that $f \in \mathrm{dom}(\Phi)$. Then, the logic rules extended in this way could be used to verify the programs whether they comply with that policy or not.

This approach may form an incomplete understanding of the features provided by the linear type system. Moreover, limitations that could be supported by our program logic are only suggested by the community and not followed in many cases. Thus, the program logic will taint such programs that use non-reentrant functions in the signal handlers as unsafe. However, they are only potentially unsafe. Moreover, with use of the reentrancy linear type system, we may push our analysis further and understand if the program is actually unsafe or, despite the fact of using non-reentrant functions, is safe. Thus, use of the program logic extended with the reentrancy linear type system allows addressing much bigger set of programs.

Examples and corresponding discussions

To show what kind of programs will be missed without the linear type system, we provide a series of examples.

Example 1: Ψ split, whether main command use non-reentrant functions or not is the main concern. If a provider of the command c complies with its 'interface' in the judgement (function f is never used inside), then it is safe to use f as a signal handler.

$$\frac{\Sigma, z : I_z \triangleright I_z; \Phi; \Psi_1 \vdash \{P\}\, c\, \{Q\}\ \texttt{throws}\ \eta \qquad \dfrac{\Psi_2, f : I_z \triangleright I_z(f) = I_z \triangleright I_z}{\Sigma; \Phi; \Psi_2, f : I_z \triangleright I_z \vdash \{I_z\}\, f(\,);\, \{I_z\}\ \texttt{throws}\ \eta}}{\Sigma; \Phi; \Psi_1, \Psi_2, f : I_z \triangleright I_z \vdash \{P \wedge I_z\}\, \texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, c\, \{Q \wedge I_z\}\ \texttt{throws}\ \eta}$$

Example 2: Ψ splitting could be done in both directions (to the left branch: body of the command, and to the right branch: body of the signal handler).

$$\frac{\dfrac{\Sigma, z_1, z_2; \Phi; \Psi \vdash \{P\}\, c\, \{Q\} \qquad \dfrac{f_2 : I_z \triangleright I_z(f_2) = I_z \triangleright I_z}{\Sigma, z_1; \Phi; f_2 : I_z \triangleright I_z \vdash \{I_z\}\, f_2(\,);\, \{I_z\}\ \texttt{throws}\ \eta}}{\Sigma, z_1; \Phi; \Psi, f_2 : I_z \triangleright I_z \vdash \{P \wedge I_z\}\, \texttt{bind}\, z_2\, \texttt{to}\, f_2\, \texttt{in}\, c\, \{Q \wedge I_z\}} \qquad \dfrac{f_1 : I_z \triangleright I_z(f_1) = I_z \triangleright I_z}{\Sigma; \Phi; f_1 : I_z \triangleright I_z \vdash \{I_z\}\, f_1(\,);\, \{I_z\}\ \texttt{throws}\ \eta}}{\Sigma; \Phi; \Psi, f_1 : I_z \triangleright I_z, f_2 : I_z \triangleright I_z \vdash \{P \wedge I_z\}\, \texttt{bind}\, z_1\, \texttt{to}\, f_1\, \texttt{in}\, (\texttt{bind}\, z_2\, \texttt{to}\, f_2\, \texttt{in}\, c)\, \{Q \wedge I_z\}\ \texttt{throws}\ \eta}$$

This examples is used to show that if the non-reentrant function is not used in a particular signal handler, then it could be used in the main command, which in turn could consist of another signal binding that will consume another non-reentrant function.

We should clarify our definition of the non-reentrant function. We say that a function is non-reentrant if it is unsafe to run two entities of the same function concurrently (via nondeterministic interrupt in our case). Thus, non-reentrancy describes the function with respect to itself, and not with respect to the broader sense of unsafe functions. Sharing a resource (or not sharing) makes functions safe or unsafe with respect to each other. It is hard to address interference between two different unsafe functions that share a resource, as we don't have an explicit notion of a resource in our language. However, we can imitate global variables that will serve as a good example of a critical resource. Thus, any function that deals with it will be unsafe and non-reentrant. A related discussion

about granularity is continued in Section 9.2.9.

Various Ψ splittings

Case 1: Two signals are bound to the same non-reentrant function (via function name)

$$\frac{\Sigma, z_1, z_2; \Phi; \Psi \vdash \{P\}\, c\, \{Q\} \qquad \dfrac{\emptyset(f) = \lightning}{\Sigma, z_1; \Phi; \emptyset \vdash \{I_z\}\, f(\,);\, \{I_z\}}}{\Sigma, z_1; \Phi; \Psi \vdash \{P \wedge I_z\}\, \texttt{bind}\, z_2\, \texttt{to}\, f\, \texttt{in}\, c\, \{Q \wedge I_z\}}$$

$$\frac{\dfrac{f : I_z \triangleright I_z(f) = I_z \triangleright I_z}{\Sigma; \Phi; f : I_z \triangleright I_z \vdash \{I_z\}\, f(\,);\, \{I_z\}}}{\Sigma; \Phi; \Psi, f : I_z \triangleright I_z \vdash \{P \wedge I_z\}\, \texttt{bind}\, z_1\, \texttt{to}\, f\, \texttt{in}\, (\texttt{bind}\, z_2\, \texttt{to}\, f\, \texttt{in}\, c)\, \{Q \wedge I_z\}}$$

Case 2: It is assumed that the local variable x has been defined and initialised to 1, c_1 equals to $(\texttt{bind}\, z_1\, \texttt{to}\, f_1\, \texttt{in}\, (\texttt{bind}\, z_2\, \texttt{to}\, f_2\, \texttt{in}\, c_2))$. Two signals are bound to two different function names, but literally, they are identical in terms of commands.

There are two problems here. First of all, RLTS is not helpful when two functions with different names but identical non-reentrant code interfere. The second, stability assumption covers functions only at the high level but inner code remains unchecked by them.

$$\emptyset; z_1, z_2 : x = 1 \rhd x = 1; \emptyset \vdash \{(x=1)\}\, c_2\, \{(x=1)\}$$

$$\frac{(f_2 : x = 1 \rhd x = 1)(f_2) = (x = 1 \rhd x = 1)}{\emptyset; z_1; f_2 \vdash \{(x=1)\}\, f_2(\,);\, \{(x=1)\}}$$

$$\frac{(x=1)\ \mathbf{stable}\ (x=1) \rhd (x=1)}{\emptyset; z_1 : x = 1 \rhd x = 1; f_2 \vdash \{(x=1)\}\, \texttt{bind}\, z_2\, \texttt{to}\, f_2\, \texttt{in}\, c_2\, \{(x=1)\}}$$

$$\frac{\emptyset; z_1 : x = 1 \rhd x = 1; f_2 \vdash \{(x=1)\}\, \texttt{bind}\, z_2\, \texttt{to}\, f_2\, \texttt{in}\, c_2\, \{(x=1)\} \qquad \dfrac{(f_1 : x = 1 \rhd x = 1)(f_1) = (x = 1 \rhd x = 1)}{\emptyset; \emptyset; f_1 \vdash \{(x=1)\}\, f_1(\,);\, \{(x=1)\}}}{\emptyset; \emptyset; f_1 : x = 1 \rhd x = 1, f_2 : x = 1 \rhd x = 1 \vdash \{(x=1)\}\, \texttt{bind}\, z_1\, \texttt{to}\, f_1\, \texttt{in}\, (\texttt{bind}\, z_2\, \texttt{to}\, f_2\, \texttt{in}\, c_2)\, \{(x=1)\}}$$

$$\emptyset; \emptyset; f_1 : x = 1 \rhd x = 1, f_2 : x = 1 \rhd x = 1 \vdash \{(x=1)\}\, c_1\, \{(x=1)\}$$

$$\frac{\emptyset; \emptyset; f_1 : x = 1 \rhd x = 1, f_2 : x = 1 \rhd x = 1 \vdash \{(x=1)\}\, (x := 0; x++;)\, \{(x=1)\}}{\emptyset; \emptyset; f_1 : x = 1 \rhd x = 1 \vdash \{(x=1)\}\, \texttt{fuN}\ f_2\ (\,) = (x := 0; x++;)\ \texttt{in}\ c_1\, \{(x=1)\}}$$

$$\frac{\emptyset; \emptyset; f_1 : x = 1 \rhd x = 1 \vdash \{(x=1)\}\, x := 0;\, \{(x=0)\} \qquad \emptyset; \emptyset; f_1 : x = 1 \rhd x = 1 \vdash \{(x=0)\}\, x++;\, \{(x=1)\}}{\emptyset; \emptyset; f_1 : x = 1 \rhd x = 1 \vdash \{(x=1)\}\, (x := 0; x++;)\, \{(x=1)\}}$$

$$\frac{\emptyset; \emptyset; f_1 : x = 1 \rhd x = 1 \vdash \{(x=1)\}\, \texttt{fuN}\ f_2\ (\,) = (x := 0; x++;)\ \texttt{in}\ c_1\, \{(x=1)\} \qquad \emptyset; \emptyset; f_1 : x = 1 \rhd x = 1 \vdash \{(x=1)\}\, (x := 0; x++;)\, \{(x=1)\}}{\emptyset; \emptyset; \emptyset \vdash \{(x=1)\}\, \texttt{fuN}\ f_1\ (\,) = (x := 0; x++;)\ \texttt{in}\ (\texttt{fuN}\ f_2\ (\,) = (x := 0; x++;)\ \texttt{in}\ c_1)\, \{(x=1)\}}$$

Case 3: It is assumed that the local variable x has been defined and initialised to 1, c_1 equals to $(\texttt{fuN}\ f_2\ (\,) = (x := 0; x++;)\ \texttt{in}\ (\texttt{bind}\, z_2\, \texttt{to}\, f_2\, \texttt{in}\, c_2))$. Two signals are bound to two different function names, but literally, they are identical in terms of commands. The main difference with an example above is that function definition and signal binding are mixed.

$$\{(x=1)\}\, x := 0;\, \{(x=0)\}$$

$$(x=0)\ \textbf{stable}\ z_1 : x=1 \triangleright x=1$$

$$\frac{(x=1)\ \textbf{stable}\ (x=1) \triangleright (x=0)}{\emptyset; z_1; f_2 : x=1 \triangleright x=1 \vdash \{(x=1)\}\, x := 0;\, \{(x=0)\}}$$

$$\frac{\emptyset; z_1; f_2 : x=1 \triangleright x=1 \vdash \{(x=0)\}\, x{+}{+};\, \{(x=1)\}}{\emptyset; z_1; f_2 : x=1 \triangleright x=1 \vdash \{(x=1)\}\, (x := 0; x{+}{+};)\, \{(x=1)\}}$$

$$\emptyset; z_1; f_2 : x=1 \triangleright x=1 \vdash \{(x=1)\}\, \texttt{bind}\, z_2\, \texttt{to}\, f_2\, \texttt{in}\, c_2\, \{(x=1)\}$$

$$\frac{\emptyset; z_1; f_2 : x=1 \triangleright x=1 \vdash \{(x=1)\}\, (x := 0; x{+}{+};)\, \{(x=1)\}}{\emptyset; z_1; \emptyset \vdash \{(x=1)\}\, (\texttt{fuN}\ f_2\ (\,) = (x := 0; x{+}{+};)\ \texttt{in}\ (\texttt{bind}\, z_2\, \texttt{to}\, f_2\, \texttt{in}\, c_2))\, \{(x=1)\}}$$

$$\frac{\emptyset; z_1; \emptyset \vdash \{(x=1)\}\, c_1\, \{(x=1)\} \qquad \dfrac{(f_1 : x=1 \triangleright x=1)(f_1) = (x=1 \triangleright x=1)}{\emptyset; \emptyset; f_1 : x=1 \triangleright x=1 \vdash \{(x=1)\}\, f_1(\,);\, \{(x=1)\}}}{\emptyset; \emptyset; f_1 : x=1 \triangleright x=1 \vdash \{(x=1)\}\, \texttt{bind}\, z_1\, \texttt{to}\, f_1\, \texttt{in}\, c_1\, \{(x=1)\}}$$

$$\emptyset; \emptyset; f_1 : x=1 \triangleright x=1 \vdash \{(x=1)\}\, \texttt{bind}\, z_1\, \texttt{to}\, f_1\, \texttt{in}\, c_1\, \{(x=1)\}$$

$$\frac{\emptyset; \emptyset; f_1 : x=1 \triangleright x=1 \vdash \{(x=1)\}\, (x := 0; x{+}{+};)\, \{(x=1)\}}{\emptyset; \emptyset; \emptyset \vdash \{(x=1)\}\, \texttt{fuN}\ f_1\ (\,) = (x := 0; x{+}{+};)\ \texttt{in}\ (\texttt{bind}\, z_1\, \texttt{to}\, f_1\, \texttt{in}\, c_1)\, \{(x=1)\}}$$

In this example, it is clear that the program in unsafe. It was recognised when the second function f_2 was added to the function context. In particular, stability assumptions failed for the function f_2 body. Thus, further logic tree derivation (for the $\texttt{bind}\, z_2\, \texttt{to}\, f_2\, \texttt{in}\, c_2$) becomes unimportant.

It should be noted, that the high strictness of the stability assumptions in this particular example has a positive effect (but it might be too strict in other cases). Technically, alternating of the function definition and immediate signal binding corresponds to the signal binding with commands (without functions). Unless current function mechanism becomes updated to address this problem (Time of Stability Check to Time of Function

Call), a language with commands looks safer.

The main difference with the previous example, where the problem has not been recognised by the logic, is that in this example function definitions alternate with signal bindings. Therefore, during the second function definition the signal context is non-empty; and stability assumptions are checked at that step.

In previous example, all functions were defined in a scope with an empty signal context; thus, their potential interference via signal handlers, which were bound later, was not restricted by stability assumptions.

The problem arise due to the fact that functions become signal handlers in two steps: function definition, and signal binding. Stability is checked when the function body is no longer visible. Thus, we may update the function definition rule to check stability even before we bind it to a signal. For example:

$$\Sigma; \Phi; \Psi, n : P_n \triangleright Q_n \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta$$

$$\Sigma; \Phi; \Psi, n : P_n \triangleright Q_n \vdash \{P_n\}\, c_n\, \{Q_n\}\ \texttt{throws}\ \eta$$

$$\frac{\Psi\ \textbf{stable}\ P_n \triangleright Q_n \qquad P_n\ \textbf{stable}\ \Psi \qquad Q_n\ \textbf{stable}\ \Psi}{\Sigma; \Phi; \Psi \vdash \{P\}\, \texttt{fuN}\ n\ (\ ) = c_n\ \texttt{in}\ c_B\, \{Q\}\ \texttt{throws}\ \eta}$$

Unfortunately, this approach results in overhead, as not every function is used in signal handlers. Moreover, that may add even more strictness to the logic, what might make it less useful.

9.2.8 Examples and Tricky Questions

Example 1: What if we try to bind a non-reentrant function to a reentrant one? In this particular example, when the "reentrancy relation" is checked, the signal context is empty, thus $\overset{R}{\vdash}$ holds. The current strictness makes our logic work. If we will try to

weaken our requirement (empty signal context), the logic may break.

$$\frac{\begin{array}{c} z : I_z \triangleright I_z; \Phi, f : P_f \triangleright Q_f; \Psi, n : P_n \triangleright Q_n \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta \\ z : I_z \triangleright I_z; \Phi, f : P_f \triangleright Q_f; \Psi, n : P_n \triangleright Q_n \vdash \{P_f\}\, \texttt{block}\, z\, \texttt{in}\, n(\,);\, \{Q_f\}\ \texttt{throws}\ \eta \\ z : I_z \triangleright I_z; \Phi, f : P_f \triangleright Q_f; \Psi, n : P_n \triangleright Q_n \overset{R}{\vdash} \texttt{block}\, z\, \texttt{in}\, n(\,); \end{array}}{z : I_z \triangleright I_z; \Phi; \Psi, n : P_n \triangleright Q_n \vdash \{P\}\, \texttt{fun}\ f\ (\,) = (\texttt{block}\, z\, \texttt{in}\, n(\,);)\ \texttt{in}\ c_B\, \{Q\}\ \texttt{throws}\ \eta}$$

$$\frac{\emptyset; \Phi, f : P_f \triangleright Q_f; \Psi, n : P_n \triangleright Q_n \overset{R}{\vdash} n(\,);}{z : I_z \triangleright I_z; \Phi, f : P_f \triangleright Q_f; \Psi, n : P_n \triangleright Q_n \overset{R}{\vdash} \texttt{block}\, z\, \texttt{in}\, n(\,);}$$

Example 2: What if a non-reentrant function n binds a signal inside? Linear type system prevents reentrancy via nondeterministic signals. As when the rule for the signal binding is applied, the nonreentrant context is splitted between body of the command (in this case call of a function $n(\,)$;) and the signal handler (which in this example is also a function call $n(\,)$;). Thus, the nonreentrant function could be used only in the "body" (left subtree) or the handler (right subtree).

$$\frac{\begin{array}{c} z : I_z \triangleright I_z; \emptyset; \Psi_1 \vdash \{P\}\, n(\,);\, \{Q\}\ \texttt{throws}\ \eta \\ \emptyset; \emptyset; \Psi_2 \vdash \{I_z\}\, n(\,);\, \{I_z\}\ \texttt{throws}\ \eta \\ \emptyset; \emptyset; n : P_n \triangleright Q_n \vdash \{P_n \wedge I_z\}\, (\texttt{bind}\, z\, \texttt{to}\, n\, \texttt{in}\, n(\,);)\, \{Q_n \wedge I_z\}\ \texttt{throws}\ \eta \\ \emptyset; \emptyset; n : P_n \triangleright Q_n \vdash \{P \wedge I_z\}\, c_B\, \{Q \wedge I_z\}\ \texttt{throws}\ \eta \end{array}}{\emptyset; \emptyset; \emptyset \vdash \{P \wedge I_z\}\, \texttt{fuN}\ n\ (\,) = (\texttt{bind}\, z\, \texttt{to}\, n\, \texttt{in}\, n(\,);)\ \texttt{in}\ c_B\, \{Q \wedge I_z\}\ \texttt{throws}\ \eta}$$

Example 3: What if a reentrant function f binds a signal inside? In this example, when the signal is called, it will call the function f, and the function f will **bind/rebind** the signal to the same function f once again. Thus, the signal will be unblocked, what may result into the signal handler interruption by itself, but the function is reentrant! It is a feature of our language.

$\mathcal{A} =$

$$z : I_z \triangleright I_z; f : P_f \triangleright Q_f; \emptyset \vdash \{P\}\, f(\,);\, \{Q\}\, \texttt{throws}\, \eta$$

$$\emptyset; f : P_f \triangleright Q_f; \emptyset \vdash \{I_z\}\, f(\,);\, \{I_z\}\, \texttt{throws}\, \eta$$

$$\emptyset; f : P_f \triangleright Q_f; \emptyset \vdash \{P_f \wedge I_z\}\, (\texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);)\, \{Q_f \wedge I_z\}\, \texttt{throws}\, \eta$$

$$\emptyset; f : P_f \triangleright Q_f; \emptyset \vdash \{P \wedge I_z\}\, c_B\, \{Q \wedge I_z\}\, \texttt{throws}\, \eta \qquad \mathcal{A}$$

$$\emptyset; f : P_f \triangleright Q_f; \emptyset \overset{R}{\vdash} (\texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);)$$

$$\emptyset; \emptyset; \emptyset \vdash \{P \wedge I_z\}\, \texttt{fun}\, f\, (\,) = (\texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);)\, \texttt{in}\, c_B\, \{Q \wedge I_z\}\, \texttt{throws}\, \eta$$

$$z : I_z \triangleright I_z; f : P_f \triangleright Q_f; \emptyset \overset{R}{\vdash} f(\,); \qquad \emptyset; f : P_f \triangleright Q_f; \emptyset \overset{R}{\vdash} f(\,);$$

$$\emptyset; f : P_f \triangleright Q_f; \emptyset \overset{R}{\vdash} (\texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);)$$

Example 4: Binding a reentrant function to a reentrant one. In this example, a function n comes from the reentrant function context. Thus, when we use it inside of the definition for the function f, the $\overset{R}{\vdash}$ relation holds.

$$\Sigma, z : I_z \triangleright I_z; \Phi, n : P_n \triangleright Q_n, f : P_f \triangleright Q_f; \Psi \vdash \{P\}\, c_B\, \{Q\}\, \texttt{throws}\, \eta$$

$$\Sigma, z : I_z \triangleright I_z; \Phi, n : P_n \triangleright Q_n, f : P_f \triangleright Q_f; \Psi \vdash \{P_f\}\, n(\,);\, \{Q_f\}\, \texttt{throws}\, \eta$$

$$\Sigma, z : I_z \triangleright I_z; \Phi, n : P_n \triangleright Q_n, f : P_f \triangleright Q_f; \Psi \overset{R}{\vdash} n(\,);$$

$$\Sigma, z : I_z \triangleright I_z; \Phi, n : P_n \triangleright Q_n; \Psi \vdash \{P\}\, \texttt{fun}\, f\, (\,) = n(\,);\, \texttt{in}\, c_B\, \{Q\}\, \texttt{throws}\, \eta$$

9.2.9 Level of Granularity

What is the optimal level of granularity to address reentrancy and safety: signal handlers, functions or commands? Shortly, signal handlers are functions in our language; thus, reentrancy of the handlers depends on the reentrancy of the functions. Moreover, even if we limit self interruption of the signal handlers, the question of handlers reentrancy (technically function reentrancy) remains open, as two different signals might be bound to the same function. Same strategy could be applied to the functions, as the same block of code might be used in more than one function body. Thus, even if forbid recursive functions, the same block of code may interleave with itself via functions with different names. Functions in our language are encapsulated blocks of code with names. When we talk about function reentrancy, we mean reentrancy of the function by itself, or in other words, reentrancy of the block of code by itself. Thus, the functions level looks like a perfect choice of granularity to address reentrancy and corresponding safety in our language. Of course, the problem of reentrancy evolves from the shared resource concurrency, but we don't have instruments to address it on the level of access to the resources.

Granularity of Ψ splitting

Granularity of the non-reentrant functions splitting depends on the definition of function non-reentrancy (or reentrancy-safety). This discussion is highly related to the logic rules design of the reentrancy judgements (Section 9.2.7).

One may define that being a non-reentrant function means that it is unsafe to interfere with any function (including reentrant and non-reentrant). Then using any non-reentrant function as a signal handler will impose huge limitations to the main program code. To ensure safety of the program all function calls should be forbidden while such signal is installed. This approach will badly influence the code reuse, as most of the time the same blocks of commands will be used repeatedly instead of the function calls. Use

of reentrant functions as the signal handlers at least allows use of reentrant functions inside of the main program code. In this scenario, an explicit notion of the non-reentrant functions brings too little in compare to the overhead it creates. Thus, it is easier to forbid using non-reentrant functions in signal handlers. However, use of non-reentrant functions inside of the main code (even with reentrant signal handlers) is still unsafe. A suggested solution would be blocking all installed signals before the non-reentrant function call. If the reentrant functions only read global variables or do not access them at all, then this limitation (signal blocking) could be withdrawn.

One may define that being a non-reentrant function means that it is unsafe to interfere with any other non-reentrant functions, but safe to interfere with reentrant functions (an example of the reentrant function could be `int get();` function that just returns some value). That would lead to a rather unbalanced Ψ splitting. We may call it as a **coarse-grained splitting**. A RLTS enforces the Ψ splitting by exclusive separating non-reentrant functions between the main code and the signal handlers. With the current scenario in charge, only one signal handler may use non-reentrant functions in a particular scope, or the main program only. One may say that the non-reentrant context splitting is rather clumsy in this scenario, but it is much easier to define a reentrancy judgment with such coarse grained splitting (Section 9.2.7).

Finally, one may define that being a non-reentrant function means that it is unsafe to interfere with respect to itself. Actually, that is the most close scenario to the real-life implementations. For example, a discussion ([56, page 424]) of the function `crypt();`, which is not *async-safe*, is explained in terms of interfering with itself. The non-reentrant function context could be split between any number of signal handlers and the main code. We call such Ψ splitting as a **fine-grained splitting**. Unfortunately, even with such fine-grained splitting a problem may arise, if two non-reentrant functions contain identical code or use the same resource. `crypt();` is non-reentrant, because it statically allocates some variables. Thus, we may assume that the non-reentrant functions deal only with the global variables that are reachable only by them. Therefore, non-reentrant functions

will be non-reentrant to itself, but with bigger chances could be reentrant to each other. On the other hand, it is harder to define a reentrancy judgment with a fine-grained Ψ splitting (Section 9.2.7).

Shift to reentrant functions

We may try to project the ideas discussed in Section 9.2.9 to the reentrant functions. When we say that function is reentrant, then it is safe to interrupt this function even by non-reentrant functions. For example, a `set(int arg);` function, that only sets some value via assignment. Assignment is an atomic operation, and calling any non-reentrant function via the signal handling before or after it does not make `set(int arg);` unsafe.

However, if any non-reentrant function is interrupted via signals by some reentrant function, then the program safety might be violated. For example, `set(int arg);` reentrant function might corrupt some global variable on which the interrupted non-reentrant function depends. It should be noted that `set(int arg);` could be non-reentrant with respect to itself despite it is almost atomic. A postcondition of the function might be invalidated if it is interrupted via another instance of the function call. A function is a wrap around commands (unwrap and command run is not atomic). Thus, even if the function contains an assignment command, we may still study a reentrancy of it.

9.2.10 Interaction Between Functions and Signals

To see how non-reentrant and reentrant functions interact with signals, let's consider examples in this section. These examples also help to understand where the stability is checked.

Nonreentrant functions and signal binding

Let a_1 and a_2 are two atomic commands. Let c_B be $f(\,);f(\,);$.

$$\frac{\dfrac{\dfrac{\Psi, f : P_f \triangleright Q_f(f) = P_f \ \triangleright \ Q_f}{\Sigma; \Phi; \Psi, f : P_f \triangleright Q_f \vdash \{P\}\, f(\,);\, \{P'\}\ \texttt{throws}\ \eta} \quad \dfrac{\Psi, f : P_f \triangleright Q_f(f) = P_f \ \triangleright \ Q_f}{\Sigma; \Phi; \Psi, f : P_f \triangleright Q_f \vdash \{P'\}\, f(\,);\, \{Q\}\ \texttt{throws}\ \eta}}{\Sigma; \Phi; \Psi, f : P_f \triangleright Q_f \vdash \{P\}\, f(\,);\, f(\,);\, \{Q\}\ \texttt{throws}\ \eta}}$$

$$\frac{\{P_f\}\, a_1\, \{P'_f\} \quad \eta\ \textbf{stable}\ \Sigma \qquad \dfrac{P_f\ \textbf{stable}\ \Sigma \quad P'_f\ \textbf{stable}\ \Sigma \quad \Sigma\ \textbf{stable}\ (P_f \triangleright P'_f)}{\Sigma; \Phi; \Psi, f : P_f \triangleright Q_f \vdash \{P_f\}\, a_1\, \{P'_f\}\ \texttt{throws}\ \eta} \quad \{P'_f\}\, a_1\, \{Q_f\} \quad \eta\ \textbf{stable}\ \Sigma \qquad \dfrac{P'_f\ \textbf{stable}\ \Sigma \quad Q_f\ \textbf{stable}\ \Sigma \quad \Sigma\ \textbf{stable}\ (P'_f \triangleright Q_f)}{\Sigma; \Phi; \Psi, f : P_f \triangleright Q_f \vdash \{P'_f\}\, a_2\, \{Q_f\}\ \texttt{throws}\ \eta}}{\Sigma; \Phi; \Psi, f : P_f \triangleright Q_f \vdash \{P_f\}\, a_1; a_2\, \{Q_f\}\ \texttt{throws}\ \eta}$$

$$\frac{\Sigma; \Phi; \Psi, f : P_f \triangleright Q_f \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta \qquad \Sigma; \Phi; \Psi, f : P_f \triangleright Q_f \vdash \{P_f\}\, a_1; a_2\, \{Q_f\}\ \texttt{throws}\ \eta}{\Sigma; \Phi; \Psi \vdash \{P\}\, \texttt{fuN}\ f\ (\,) = a_1; a_2\ \texttt{in}\ c_B\, \{Q\}\ \texttt{throws}\ \eta}$$

Let c_B equals $\texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, c$. Assume that $P_f = Q_f = I_z$. We also need to ensure that I_z holds from the very beginning.

$$\frac{\Sigma; \Phi; \Psi, f : P_f \triangleright Q_f \vdash \{P \wedge I_z\}\, \texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, c\, \{Q \wedge I_z\}\ \texttt{throws}\ \eta \qquad \Sigma; \Phi; \Psi, f : P_f \triangleright Q_f \vdash \{P_f\}\, a_1; a_2\, \{Q_f\}\ \texttt{throws}\ \eta}{\Sigma; \Phi; \Psi \vdash \{P \wedge I_z\}\, \texttt{fuN}\ f\ (\,) = a_1; a_2\ \texttt{in}\ (\texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, c)\, \{Q \wedge I_z\}\ \texttt{throws}\ \eta}$$

Then there are two options for Ψ splitting.

Let c equals $f(\,)$;. Also assume that $P_f = P$ and $Q_f = Q$. Option $\Psi_1, f : P_f \triangleright Q_f$:

$$\frac{\dfrac{\dfrac{\Psi_1, f : P_f \triangleright Q_f(f) = P_f \triangleright Q_f}{\Sigma, z : I_z \triangleright I_z; \Phi; \Psi_1, f : P_f \triangleright Q_f \vdash \{P\}\, f(\,);\, \{Q\}\, \mathtt{throws}\, \eta} \quad \dfrac{\Psi_2(f) = \lightning}{\Sigma; \Phi; \Psi_2 \vdash \{I_z\}\, f(\,);\, \{I_z\}\, \mathtt{throws}\, \eta}}{\Sigma; \Phi; \Psi_1, \Psi_2 \vdash \{P \wedge I_z\}\, \mathtt{bind}\, z\, \mathtt{to}\, f\, \mathtt{in}\, c\, \{Q \wedge I_z\}\, \mathtt{throws}\, \eta}$$

Option $\Psi_2, f : P_f \triangleright Q_f$: So c can not have any function f calls.

$$\frac{\dfrac{\Sigma, z : I_z \triangleright I_z; \Phi; \Psi_1 \vdash \{P\}\, c\, \{Q\}\, \mathtt{throws}\, \eta \qquad \dfrac{\Psi_2, f : P_f \triangleright Q_f(f) = P_f \triangleright Q_f}{\Sigma; \Phi; \Psi_2, f : P_f \triangleright Q_f \vdash \{I_z\}\, f(\,);\, \{I_z\}\, \mathtt{throws}\, \eta}}{\Sigma; \Phi; \Psi_1, \Psi_2 \vdash \{P \wedge I_z\}\, \mathtt{bind}\, z\, \mathtt{to}\, f\, \mathtt{in}\, c\, \{Q \wedge I_z\}\, \mathtt{throws}\, \eta}$$

Reentrant functions and signal binding

Let's consider signal binding to reentrant functions. Let c equals $f(\,)$; and assume that $P = Q = P_f = Q_f = I_z$.

$$\frac{\dfrac{\dfrac{\dfrac{\Phi, f : P_f \triangleright Q_f(f) = P_f \triangleright Q_f}{\Sigma, z : I_z \triangleright I_z; \Phi; \Psi_1 \vdash \{P\}\, f(\,);\, \{Q\}\, \mathtt{throws}\, \eta} \quad \dfrac{\Phi, f : P_f \triangleright Q_f(f) = P_f \triangleright Q_f}{\Sigma; \Phi, f : P_f \triangleright Q_f; \Psi_2 \vdash \{I_z\}\, f(\,);\, \{I_z\}\, \mathtt{throws}\, \eta}}{\Sigma; \Phi, f : P_f \triangleright Q_f; \Psi_1, \Psi_2 \vdash \{P \wedge I_z\}\, \mathtt{bind}\, z\, \mathtt{to}\, f\, \mathtt{in}\, c\, \{Q \wedge I_z\}\, \mathtt{throws}\, \eta}}{\Sigma; \Phi, f : P_f \triangleright Q_f; \Psi \vdash \{P \wedge I_z\}\, \mathtt{bind}\, z\, \mathtt{to}\, f\, \mathtt{in}\, c\, \{Q \wedge I_z\}\, \mathtt{throws}\, \eta}$$

$$\frac{\Sigma; \Phi, f : P_f \triangleright Q_f; \Psi \vdash \{P_f\}\, a_1; a_2\, \{Q_f\}\, \mathtt{throws}\, \eta}{\Sigma; \Phi; \Psi \vdash \{P \wedge I_z\}\, \mathtt{fun}\, f\, (\,) = a_1; a_2\, \mathtt{in}\, (\mathtt{bind}\, z\, \mathtt{to}\, f\, \mathtt{in}\, c)\, \{Q \wedge I_z\}\, \mathtt{throws}\, \eta}$$

9.2.11 Motivational Examples

Ψ splitting

A function context Ψ splitting ensures that non-reentrant functions may not be called simultaneously in the main code and in the signal handler. Thus, if a particular non-reentrant function is called from the main code and during execution is interrupted by a signal that bound to the same function, then the safety of a program is jeopardised (first derivation tree). However, if the same signal interrupted the main program while another code was running, then the handler (with non-reentrant function(s)) run is safe (second derivation tree).

In a summary for the first two derivations, we can say that it is a chance for the program to preserve safety even signal handlers with non-reentrant functions are executed. One may try to avoid the calling non-reentrant functions from the main code. However, signal handlers are interruptible by another signals; thus, the same function could be called via signal handler interruption by another handler.

There is also an extreme case `bind` z `to` f `in` $f(\,)$;, where a signal is bound to the non-reentrant function in a block where that function is called (third and forth derivation trees). Thus, the non-reentrant function is definitely called at least once. So processing a signal z is not safe in any case, as non-reentrant function already has been used as linear resource.

$$\dfrac{\dfrac{\Sigma, z : I_z \triangleright I_z; \Phi; \Psi_1, f : I_z \triangleright I_z \vdash \{P\}\, c\, \{Q\}\ \texttt{throws}\ \eta \qquad \Psi_2(f) = \lightning}{\Sigma; \Phi; \Psi_2 \vdash \{I_z\}\, f(\,);\, \{I_z\}\ \texttt{throws}\ \eta}}{\Sigma; \Phi; \Psi_1, \Psi_2, f : I_z \triangleright I_z \vdash \{P \wedge I_z\}\, \texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, c\, \{Q \wedge I_z\}\ \texttt{throws}\ \eta}$$

$$\dfrac{\dfrac{\Sigma, z : I_z \triangleright I_z; \Phi; \Psi_1 \vdash \{P\}\, c\, \{Q\}\ \texttt{throws}\ \eta \qquad \Psi_2, f : I_z \triangleright I_z(f) = I_z \triangleright I_z}{\Sigma; \Phi; \Psi_2, f : I_z \triangleright I_z \vdash \{I_z\}\, f(\,);\, \{I_z\}\ \texttt{throws}\ \eta}}{\Sigma; \Phi; \Psi_1, \Psi_2, f : I_z \triangleright I_z \vdash \{P \wedge I_z\}\, \texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, c\, \{Q \wedge I_z\}\ \texttt{throws}\ \eta}$$

$$\dfrac{\dfrac{\Psi_1, f : I_z \triangleright I_z(f) = I_z \triangleright I_z}{\Sigma, z : I_z \triangleright I_z; \Phi; \Psi_1, f : I_z \triangleright I_z \vdash \{I_z\}\, f(\,);\, \{I_z\}\ \texttt{throws}\ \eta} \qquad \dfrac{\Psi_2(f) = \lightning}{\Sigma; \Phi; \Psi_2 \vdash \{I_z\}\, f(\,);\, \{I_z\}\ \texttt{throws}\ \eta}}{\Sigma; \Phi; \Psi_1, \Psi_2, f : I_z \triangleright I_z \vdash \{I_z\}\, \texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);\, \{I_z\}\ \texttt{throws}\ \eta}$$

$$\dfrac{\dfrac{\dfrac{\Psi_1(f) = \lightning}{\Sigma, z : I_z \triangleright I_z; \Phi; \Psi_1 \vdash \{I_z\}\, f(\,);\, \{I_z\}\ \texttt{throws}\ \eta} \qquad \Psi_2, f : I_z \triangleright I_z(f) = I_z \triangleright I_z}{\Sigma; \Phi; \Psi_2, f : I_z \triangleright I_z \vdash \{I_z\}\, f(\,);\, \{I_z\}\ \texttt{throws}\ \eta}}{\Sigma; \Phi; \Psi_1, \Psi_2, f : I_z \triangleright I_z \vdash \{I_z\}\, \texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);\, \{I_z\}\ \texttt{throws}\ \eta}$$

One may ask, do we actually need the forth derivation? The function f is called at least once in the main code. What if the signal arrives before the main command even started? It the automated verifier builds derivation trees, then it depends on the implementation whether it starts from the left or right subtree.

Reentrant functions and indirect recursion

The normal program style will have the form as in Figure 9.21 and Figure 9.22, where one define a function in an outer scope, then bind a signal to the function in the inner scope, and nesting may continue. In other words, one define a function set, then a set

of signals, and then the main code runs, which may call predefined function and handle registered signals.

In our language, there is no limitations on function code for using nested commands or signal binding; thus, one can bind a signal inside of the function body, as in Figure 9.23. An effect of such construct is that when the function will be called, a new signal will be bound, so the c_1 of a function will run in a scope with extended signal binding. As soon as control flow leaves the function scope, the recently installed signal will be automatically uninstalled.

One may decide to move all the code inside of the function body, and leave only the function call inside of the program body, as in Figure 9.24. In a sense, it is like a main function in many programming languages, the program execution starts from calling the main function.

If a function comes from the reentrant context, then the next code

$$\texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);$$

is not an extreme case (Figure 9.25). One may note that a call of a function f in a left subtree (main code) could be interrupted by a signal z. On the other hand, a call of the function f in a right subtree (handler code) could not be interrupted by itself via a signal z.

One may try to create an extreme case derived from the Figure 9.25 and to loop the function call with a signal binding. A trick is in putting binding scope into the function body: $\texttt{fun}\ f\ (\ ) = (\texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);)\ \texttt{in}\ f(\,);$. A corresponding derivation in logic is presented in Figure 9.26. Operationally, it would be almost meaningless, because a function call triggers a signal binding, which in its turn binds a signal to the same function and calls the same function recursively. Even without processing arriving signals, that will result in a loop. Anyway, our logic handles this case. If we use `bind/1` instead of `bind`, then we get another interesting situation. A one-shot signal will be reinstalled in

$$\dfrac{\dfrac{\Sigma, z : I_z \triangleright I_z; \Phi, f : I_z \triangleright I_z; \Psi_1 \vdash \{P\}\, c_2\, \{Q\}\ \texttt{throws}\ \eta \qquad \Phi, f : I_z \triangleright I_z(f) = I_z \triangleright I_z}{\Sigma; \Phi, f : I_z \triangleright I_z; \Psi_2 \vdash \{I_z\}\, f(\,);\, \{I_z\}\ \texttt{throws}\ \eta}}{\Sigma; \Phi, f : P_f \triangleright Q_f; \Psi \vdash \{P \wedge I_z\}\, \texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, c_2\, \{Q \wedge I_z\}\ \texttt{throws}\ \eta} \qquad \Sigma; \Phi, f : P_f \triangleright Q_f; \Psi \vdash \{P \wedge I_z\}\, \texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, c_2\, \{Q \wedge I_z\}\ \texttt{throws}\ \eta \qquad \Sigma; \Phi, f : P_f \triangleright Q_f; \Psi \vdash \{P_f\}\, c_1\, \{Q_f\}\ \texttt{throws}\ \eta}{\Sigma; \Phi; \Psi \vdash \{P \wedge I_z\}\, \texttt{fun}\ f\ (\,) = c_1\ \texttt{in}\ (\texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, c_2)\, \{Q \wedge I_z\}\ \texttt{throws}\ \eta}$$

Figure 9.21: Example 01

$$\dfrac{\dfrac{\Sigma, z : P_z \triangleright Q_z; \Phi, f : P_z \triangleright Q_z; \Psi \vdash \{P\}\, c_2\, \{Q\}\ \texttt{throws}\ \eta \qquad \Phi, f : P_z \triangleright Q_z(f) = P_z \triangleright Q_z}{\Sigma; \Phi, f : P_z \triangleright Q_z; \Psi \vdash \{P_z\}\, f(\,);\, \{Q_z\}\ \texttt{throws}\ \eta}}{\Sigma; \Phi, f : P_z \triangleright Q_z; \Psi \vdash \{P \wedge P_z\}\, \texttt{bind/1}\, z\, \texttt{to}\, f\, \texttt{in}\, c_2\, \{Q \wedge (P_z \vee Q_z)\}\ \texttt{throws}\ \eta} \qquad \Sigma; \Phi, f : P_z \triangleright Q_z; \Psi \vdash \{P \wedge P_z\}\, \texttt{bind/1}\, z\, \texttt{to}\, f\, \texttt{in}\, c_2\, \{Q \wedge (P_z \vee Q_z)\}\ \texttt{throws}\ \eta \qquad \Sigma; \Phi, f : P_z \triangleright Q_z; \Psi \vdash \{P_z\}\, c_1\, \{Q_z\}\ \texttt{throws}\ \eta}{\Sigma; \Phi; \Psi \vdash \{P \wedge P_z\}\, \texttt{fun}\ f\ (\,) = c_1\ \texttt{in}\ (\texttt{bind/1}\, z\, \texttt{to}\, f\, \texttt{in}\, c_2)\, \{Q \wedge (P_z \vee Q_z)\}\ \texttt{throws}\ \eta}$$

Figure 9.22: Example 02

every nested scope, thus it can run only once in every scope, but from the point of view of the whole program, it will be executed more than one times.

What if we just write $\texttt{fun}\ f\ (\,) = f(\,);\ \texttt{in}\ f(\,);$? Let's consider a derivation tree in Figure 9.27. One can observe that the loop could be contracted without signal binding mechanism, as the recursive functions are permitted in our language.

Some problems arise even without signal binding. The program itself could be designed to be bad, without respect to the logic. For example, check Figure 9.28.

$$\dfrac{\Sigma; \Phi, f : P_f \triangleright Q_f; \Psi \vdash \{P\}\, c_2\, \{Q\}\ \texttt{throws}\ \eta \qquad \Sigma; \Phi, f : P_f \triangleright Q_f; \Psi \vdash \{P \wedge I_z\}\, \texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, c_1\, \{Q \wedge I_z\}\ \texttt{throws}\ \eta}{\Sigma; \Phi; \Psi \vdash \{P \wedge I_z\}\, \texttt{fun}\ f\ (\,) = (\texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, c_1)\ \texttt{in}\ c_2\, \{Q \wedge I_z\}\ \texttt{throws}\ \eta}$$

Figure 9.23: Example 03

$$\frac{\Sigma; \Phi, f : P_f \triangleright Q_f; \Psi \vdash \{P_f\}\, f(\,);\, \{Q_f\}\, \texttt{throws}\, \eta \qquad \Sigma; \Phi, f : P_f \triangleright Q_f; \Psi \vdash \{P_f\}\, c\, \{Q_f\}\, \texttt{throws}\, \eta}{\Sigma; \Phi; \Psi \vdash \{P_f\}\, \texttt{fun}\ f\ (\,) = c\ \texttt{in}\ f(\,);\, \{Q_f\}\, \texttt{throws}\, \eta}$$

Figure 9.24: Example 04

$$\frac{\dfrac{\Phi, f : I_z \triangleright I_z(f) = I_z \triangleright I_z}{\Sigma, z : I_z \triangleright I_z; \Phi, f : I_z \triangleright I_z; \Psi_1 \vdash \{I_z\}\, f(\,);\, \{I_z\}\, \texttt{throws}\, \eta} \qquad \dfrac{\Phi, f : I_z \triangleright I_z(f) = I_z \triangleright I_z}{\Sigma; \Phi, f : I_z \triangleright I_z; \Psi_2 \vdash \{I_z\}\, f(\,);\, \{I_z\}\, \texttt{throws}\, \eta}}{\Sigma; \Phi, f : I_z \triangleright I_z; \Psi_1, \Psi_2 \vdash \{I_z\}\, \texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);\, \{I_z\}\, \texttt{throws}\, \eta}$$

Figure 9.25: Example 05

$$\frac{\Sigma; \Phi, f : P_f \triangleright Q_f; \Psi \vdash \{P_f\}\, f(\,);\, \{Q_f\}\, \texttt{throws}\, \eta \qquad \Sigma; \Phi, f : P_f \triangleright Q_f; \Psi \vdash \{P \wedge I_z\}\, \texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);\, \{Q \wedge I_z\}\, \texttt{throws}\, \eta}{\Sigma; \Phi; \Psi \vdash \{P \wedge I_z\}\, \texttt{fun}\ f\ (\,) = (\texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);)\ \texttt{in}\ f(\,);\, \{Q \wedge I_z\}\, \texttt{throws}\, \eta}$$

Figure 9.26: Example 06

$$\frac{\Sigma; \Phi, f : I \triangleright I; \Psi \vdash \{I\}\, f(\,);\, \{I\}\, \texttt{throws}\, \eta \qquad \Sigma; \Phi, f : I \triangleright I; \Psi \vdash \{I\}\, f(\,);\, \{I\}\, \texttt{throws}\, \eta \qquad \Sigma; \Phi, f : I \triangleright I; \Psi \overset{R}{\vdash} f(\,);}{\Sigma; \Phi; \Psi \vdash \{I\}\, \texttt{fun}\ f\ (\,) = f(\,);\ \texttt{in}\ f(\,);\, \{I\}\, \texttt{throws}\, \eta}$$

Figure 9.27: Example 07

$\mathcal{A} =$

$\emptyset; f : P_f \triangleright Q_f; \emptyset \vdash \{P \wedge I_z\}\, \texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, (x := 1; f(\,); x := 0;)\, \{Q \wedge I_z\}\, \texttt{throws}\, \eta$

$\mathcal{B} =$

$\emptyset; f : x = 0 \triangleright x = 0; \emptyset \vdash \{x = 0\}\, (x := 1; f(\,); x := 0;)\, \{x = 0\}\, \texttt{throws}\, \eta$

$$\frac{\mathcal{A} \qquad \mathcal{B}}{\emptyset; \emptyset; \emptyset \vdash \{P \wedge I_z\}\, \texttt{fun}\, f\, (\,) = (x := 1; f(\,); x := 0;)\, \texttt{in}\, (\texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, (x := 1; f(\,); x := 0;))\, \{Q \wedge I_z\}\, \texttt{throws}\, \eta}$$

Subtree $\mathcal{B}$ derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\{x = 0\}\, x := 1;\, \{x = 1\}}{\emptyset; f : x = 0 \triangleright x = 0; \emptyset \vdash \{x = 0\}\, x := 1;\, \{x = 1\}\, \texttt{throws}\, \eta} \quad (f : x = 0 \triangleright x = 0)(f) = (x = 0 \triangleright x = 0)}{\emptyset; f : x = 0 \triangleright x = 0; \emptyset \vdash \{x = 1\}\, f(\,);\, \{x = \_\}\, \texttt{throws}\, \eta}}{\emptyset; f : x = 0 \triangleright x = 0; \emptyset \vdash \{x = 0\}\, (x := 1; f(\,);)\, \{x = \_\}\, \texttt{throws}\, \eta} \\ \emptyset; f : x = 0 \triangleright x = 0; \emptyset \vdash \{x = 0\}\, (x := 1; f(\,);)\, \{x = 0\}\, \texttt{throws}\, \eta}{\emptyset; f : x = 0 \triangleright x = 0; \emptyset \vdash \{x = \_\}\, x := 0;\, \{x = 0\}\, \texttt{throws}\, \eta}}{\emptyset; f : x = 0 \triangleright x = 0; \emptyset \vdash \{x = 0\}\, (x := 1; f(\,); x := 0;)\, \{x = 0\}\, \texttt{throws}\, \eta}$$

Figure 9.28: Example 08

$$\dfrac{\dfrac{z : x > 0 \triangleright x > 0; f : x > 0 \triangleright x > 0; \emptyset \vdash \{x > 0\}\, (x + +;)\, \{x > 0\} \quad \Phi(f) = (x > 0) \triangleright (x > 0)}{\emptyset; f : x > 0 \triangleright x > 0; \emptyset \vdash \{x > 0\}\, f(\,);\, \{x > 0\}}}{\emptyset; f : x > 0 \triangleright x > 0; \emptyset \vdash \{x > 0\}\, (\texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, x + +;)\, \{x > 0\}}$$

$$\dfrac{\dfrac{\Phi(f) = (x > 0) \triangleright (x > 0)}{\emptyset; f : x > 0 \triangleright x > 0; \emptyset \vdash \{x > 0\}\, f(\,);\, \{x > 0\}} \quad \emptyset; f : x > 0 \triangleright x > 0; \emptyset \vdash \{x > 0\}\, (\texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, x + +;)\, \{x > 0\}}{\emptyset; \emptyset; \emptyset \vdash \{(x > 0)\}\, \texttt{fun}\, f\, (\,) = (\texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, x + +;)\, \texttt{in}\, f(\,);\, \{(x > 0)\}}$$

Figure 9.29: Example 09

$$\dfrac{\dfrac{F(f) = c_f \quad \dfrac{[\,z \mapsto f\,];\emptyset;\,[\,f \mapsto (\texttt{bind}\,z\,\texttt{to}\,f\,\texttt{in}\,x{+}{+};)\,] \Vdash s_1, x{+}{+};\Downarrow s_2}{\emptyset;\emptyset;\,[\,f \mapsto (\texttt{bind}\,z\,\texttt{to}\,f\,\texttt{in}\,x{+}{+};)\,] \Vdash s_1, (\texttt{bind}\,z\,\texttt{to}\,f\,\texttt{in}\,x{+}{+};) \Downarrow s_2}}{\emptyset;\emptyset;\,[\,f \mapsto (c_f)\,] \Vdash s_1, f(\,);\Downarrow s_2} \quad \dfrac{[\,z \mapsto f\,](z) = f \quad \emptyset;\emptyset;\,[\,f \mapsto (c_f)\,] \Vdash s_1, f(\,);\Downarrow s_2 \quad [\,z \mapsto f\,];\emptyset;\,[\,f \mapsto (c_f)\,] \Vdash s_2, x{+}{+};\Downarrow s_3}{\dfrac{[\,z \mapsto f\,];\emptyset;\,[\,f \mapsto (\texttt{bind}\,z\,\texttt{to}\,f\,\texttt{in}\,x{+}{+};)\,] \Vdash s_1, x{+}{+};\Downarrow s_3}{F(f) = c_f \quad \emptyset;\emptyset;\,[\,f \mapsto (\texttt{bind}\,z\,\texttt{to}\,f\,\texttt{in}\,x{+}{+};)\,] \Vdash s_1, (\texttt{bind}\,z\,\texttt{to}\,f\,\texttt{in}\,x{+}{+};) \Downarrow s_3}}}{\emptyset;\emptyset;\,[\,f \mapsto (\texttt{bind}\,z\,\texttt{to}\,f\,\texttt{in}\,x{+}{+};)\,] \Vdash s_1, f(\,);\Downarrow s_3}}{\emptyset;\emptyset;\emptyset \Vdash s_1, \texttt{fun}\ f\ (\,) = (\texttt{bind}\,z\,\texttt{to}\,f\,\texttt{in}\,x{+}{+};)\ \texttt{in}\ f(\,);\Downarrow s_3}$$

Figure 9.30: Example 10

$$\dfrac{\dfrac{\Phi(f) = (x > 0) \triangleright (x > 0)}{z : x > 0 \triangleright x > 0; f : x > 0 \triangleright x > 0; \emptyset \vdash \{x > 0\}\, f(\,);\, \{x > 0\}} \quad \dfrac{\Phi(f) = (x > 0) \triangleright (x > 0)}{\emptyset; f : x > 0 \triangleright x > 0; \emptyset \vdash \{x > 0\}\, f(\,);\, \{x > 0\}}}{\emptyset; f : x > 0 \triangleright x > 0; \emptyset \vdash \{x > 0\}\, (\texttt{bind}\,z\,\texttt{to}\,f\,\texttt{in}\,f(\,);)\, \{x > 0\}}$$

$$\dfrac{\dfrac{\Phi(f) = (x > 0) \triangleright (x > 0)}{\emptyset; f : x > 0 \triangleright x > 0; \emptyset \vdash \{x > 0\}\, f(\,);\, \{x > 0\}} \quad \emptyset; f : x > 0 \triangleright x > 0; \emptyset \vdash \{x > 0\}\, (\texttt{bind}\,z\,\texttt{to}\,f\,\texttt{in}\,f(\,);)\, \{x > 0\}}{\emptyset; \emptyset; \emptyset \vdash \{(x > 0)\}\, \texttt{fun}\ f\ (\,) = (\texttt{bind}\,z\,\texttt{to}\,f\,\texttt{in}\,f(\,);)\ \texttt{in}\ f(\,);\, \{(x > 0)\}}$$

Figure 9.31: Example 11

In Figure 9.29, $\texttt{throws}\eta$ is omitted for space saving. In Figure 9.30, if no signal arrives, then the result of execution $\texttt{fun}\ f\ (\,) = (\texttt{bind}\,z\,\texttt{to}\,f\,\texttt{in}\,x{+}{+};)\ \texttt{in}\ f(\,);$ is an increment of the initial value of x by one. For every signal interrupt and consequent recursive function call, a value of x will be incremented by one. Signal arrives nondeterministically, thus the final value of x is unpredictable. However, x could be used as a counter of arrived and processed signals. Note: $c_f = \texttt{bind}\,z\,\texttt{to}\,f\,\texttt{in}\,x{+}{+};$

In Figure 9.31, $\texttt{throws}\eta$ is omitted for space saving. Even without signals, $(\texttt{bind}\,z\,\texttt{to}\,f\,\texttt{in}\,f(\,);)$ will never terminate (Figure 9.32).

$$\dfrac{F(f) = c_f \qquad \dfrac{[\,z \mapsto f\,];\emptyset;\,[\,f \mapsto c_f\,] \Vdash s_2, f(\,);\Downarrow s_3}{[\,z \mapsto f\,];\emptyset;\,[\,f \mapsto c_f\,] \Vdash s_2, (\texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);\,) \Downarrow s_3}}{[\,z \mapsto f\,];\emptyset;\,[\,f \mapsto (c_f)\,] \Vdash s_2, f(\,);\Downarrow s_3}$$

$$\dfrac{F(f) = c_f \qquad \dfrac{\dfrac{[\,z \mapsto f\,](z) = f \quad \emptyset;\emptyset;\,[\,f \mapsto (c_f)\,] \Vdash s_1, f(\,);\Downarrow s_2 \quad [\,z \mapsto f\,];\emptyset;\,[\,f \mapsto (c_f)\,] \Vdash s_2, f(\,);\Downarrow s_3}{[\,z \mapsto f\,];\emptyset;\,[\,f \mapsto (\texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);\,)\,] \Vdash s_1, f(\,);\Downarrow s_3}}{\emptyset;\emptyset;\,[\,f \mapsto (\texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);\,)\,] \Vdash s_1, (\texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);\,) \Downarrow s_3}}{\emptyset;\emptyset;\,[\,f \mapsto (\texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);\,)\,] \Vdash s_1, f(\,);\Downarrow s_3}$$

$$\emptyset;\emptyset;\emptyset \Vdash s_1, \texttt{fun}\ f\ (\,) = (\texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);\,)\ \texttt{in}\ f(\,);\Downarrow s_3$$

Figure 9.32: Example 12

9.2.12 Non-linear Interference

The aim of these examples is to show what may happen when a function non-linearly interfere with itself. Therefore, there is no division between reentrant and non-reentrant functions in the version of the logic used in these examples.

Initial set up where every nested scope presented separately. We define a set of variables x, y, t in the most outer scope. Thus, from any inner scope these variables will be considered as global.

$$\texttt{local}\ x, y, t\ \texttt{in}\ c_1$$

Then we define a non-reentrant function n that nullifies the variable t (in c_1). (`fun` in operational semantics, and `fuN` in logic;)

$$\texttt{fun}\ n\ (\,) = (t := 0;\,)\ \texttt{in}\ c_2$$

as a space saving measure, we may use n_b notation to express body of the function n.

Then we define a function f (in c_2), such that

$$\texttt{fun}\ f\ (\,) = (n(\,); t := x; x := y; y := t; n(\,);\,)\ \texttt{in}\ c_3$$

as a space saving measure, we may use f_b notation to express body of the function f.

Then we bind f to a signal z (in c_3):

$$\texttt{bind/1}\ z\ \texttt{to}\ f\ \texttt{in}\ c_4$$

And the main program sets initial values and calls f (in c_4):

$$x := 1; y := 9; f();$$

Thus, we have the next derivation tree (it is actually a subtree, as we trimmed the set up bits) in case if signal z never arrives: (signal handling is not addressed in the example below, but we still have to show the omega splitting explicitly)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset;\,[\,z \mapsto f\,];\,[\,n \mapsto n_b\,]\,[\,f \mapsto f_b\,] \Vdash s, x := 1; \Downarrow s\,[\,x \mapsto 1\,]} \quad \dfrac{}{\emptyset;\emptyset;\,[\,n \mapsto n_b\,]\,[\,f \mapsto f_b\,] \Vdash s\,[\,x \mapsto 1\,], y := 9; \Downarrow s\,[\,x \mapsto 1\,]\,[\,y \mapsto 9\,]}}{\emptyset;\,[\,z \mapsto f\,];\,[\,n \mapsto n_b\,]\,[\,f \mapsto f_b\,] \Vdash s, x := 1; y := 9; \Downarrow s\,[\,x \mapsto 1\,]\,[\,y \mapsto 9\,]} \quad \dfrac{}{(\,[\,n \mapsto n_b\,]\,[\,f \mapsto f_b\,])(f) = f_b} \quad \dfrac{}{\emptyset;\emptyset;\,[\,n \mapsto n_b\,]\,[\,f \mapsto f_b\,] \Vdash s\,[\,x \mapsto 1\,]\,[\,y \mapsto 9\,], (n(\,); t := x; x := y; y := t; n(\,);) \Downarrow s\,[\,x \mapsto 9\,]\,[\,y \mapsto 1\,]}}{\emptyset;\emptyset;\,[\,n \mapsto n_b\,]\,[\,f \mapsto f_b\,] \Vdash s\,[\,x \mapsto 1\,]\,[\,y \mapsto 9\,], f(); \Downarrow s\,[\,x \mapsto 9\,]\,[\,y \mapsto 1\,]} \quad \dfrac{}{\emptyset;\,[\,z \mapsto f\,];\,[\,n \mapsto n_b\,]\,[\,f \mapsto f_b\,] \Vdash s, x := 1; y := 9; \Downarrow s\,[\,x \mapsto 1\,]\,[\,y \mapsto 9\,]} \quad \dfrac{}{\emptyset;\emptyset;\,[\,n \mapsto n_b\,]\,[\,f \mapsto f_b\,] \Vdash s\,[\,x \mapsto 1\,]\,[\,y \mapsto 9\,], f(); \Downarrow s\,[\,x \mapsto 9\,]\,[\,y \mapsto 1\,]}}{\emptyset;\,[\,z \mapsto f\,];\,[\,n \mapsto n_b\,]\,[\,f \mapsto f_b\,] \Vdash s, x := 1; y := 9; f(); \Downarrow s\,[\,x \mapsto 9\,]\,[\,y \mapsto 1\,]}$$

What does happen if signal arrives? For simplicity, assume that f_b equals to $(t := x; x := y; y := t;)$.

Without signal handling:

$$\dfrac{\dfrac{\dfrac{}{\emptyset;\,[z\mapsto f];\,[f\mapsto f_b]\Vdash s\,[x\mapsto 1]\,[y\mapsto 9], t:=x;\Downarrow s\,[x\mapsto 1]\,[y\mapsto 9]\,[t\mapsto 1]}\quad \dfrac{}{\emptyset;\emptyset;\,[f\mapsto f_b]\Vdash s\,[x\mapsto 1]\,[y\mapsto 9]\,[t\mapsto 1], x:=y;\Downarrow s\,[x\mapsto 9]\,[y\mapsto 9]\,[t\mapsto 1]}}{\emptyset;\,[z\mapsto f];\,[f\mapsto f_b]\Vdash s\,[x\mapsto 1]\,[y\mapsto 9], (t:=x;x:=y;)\Downarrow s\,[x\mapsto 9]\,[y\mapsto 9]\,[t\mapsto 1]}\quad \dfrac{\dfrac{}{\emptyset;\,[z\mapsto f];\,[f\mapsto f_b]\Vdash s\,[x\mapsto 1]\,[y\mapsto 9], (t:=x;x:=y;)\Downarrow s\,[x\mapsto 9]\,[y\mapsto 9]\,[t\mapsto 1]}\quad \dfrac{}{\emptyset;\emptyset;\,[f\mapsto f_b]\Vdash s\,[x\mapsto 9]\,[y\mapsto 9]\,[t\mapsto 1], y:=t;\Downarrow s\,[x\mapsto 9]\,[y\mapsto 1]\,[t\mapsto 1]}}{\emptyset;\,[z\mapsto f];\,[f\mapsto f_b]\Vdash s\,[x\mapsto 1]\,[y\mapsto 9], (t:=x;x:=y;y:=t;)\Downarrow s\,[x\mapsto 9]\,[y\mapsto 1]\,[t\mapsto 1]}$$

One may observe that when the program terminates the values of x and y are swapped, and as side effect the variable t contains the last value of y.

With signal handling just before the first command:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset;\emptyset;\,[f\mapsto f_b]\Vdash s\,[x\mapsto 1]\,[y\mapsto 9], t:=x;\Downarrow s\,[x\mapsto 1]\,[y\mapsto 9]\,[t\mapsto 1]}\quad \dfrac{}{\emptyset;\emptyset;\,[f\mapsto f_b]\Vdash s\,[x\mapsto 1]\,[y\mapsto 9]\,[t\mapsto 1], x:=y;\Downarrow s\,[x\mapsto 9]\,[y\mapsto 9]\,[t\mapsto 1]}}{\emptyset;\emptyset;\,[f\mapsto f_b]\Vdash s\,[x\mapsto 1]\,[y\mapsto 9], (t:=x;x:=y;)\Downarrow s\,[x\mapsto 9]\,[y\mapsto 9]\,[t\mapsto 1]}\quad \dfrac{\dfrac{}{\emptyset;\emptyset;\,[f\mapsto f_b]\Vdash s\,[x\mapsto 1]\,[y\mapsto 9], (t:=x;x:=y;)\Downarrow s\,[x\mapsto 9]\,[y\mapsto 9]\,[t\mapsto 1]}\quad \dfrac{}{\emptyset;\emptyset;\,[f\mapsto f_b]\Vdash s\,[x\mapsto 9]\,[y\mapsto 9]\,[t\mapsto 1], y:=t;\Downarrow s\,[x\mapsto 9]\,[y\mapsto 1]\,[t\mapsto 1]}}{\emptyset;\emptyset;\,[f\mapsto f_b]\Vdash s\,[x\mapsto 1]\,[y\mapsto 9], (t:=x;x:=y;y:=t;)\Downarrow s\,[x\mapsto 9]\,[y\mapsto 1]\,[t\mapsto 1]}\quad [f\mapsto f_b](f)=f_b}{\emptyset;\emptyset;\,[f\mapsto f_b]\Vdash s\,[x\mapsto 1]\,[y\mapsto 9], (t:=x;x:=y;y:=t;)\Downarrow s\,[x\mapsto 9]\,[y\mapsto 1]\,[t\mapsto 1]}\quad [z\mapsto f](z)=f$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset;\emptyset;\,[f\mapsto f_b]\Vdash s\,[x\mapsto 1]\,[y\mapsto 9], f(\,);\Downarrow s\,[x\mapsto 9]\,[y\mapsto 1]\,[t\mapsto 1]}\quad \dfrac{}{\emptyset;\emptyset;\,[f\mapsto f_b]\Vdash s\,[x\mapsto 9]\,[y\mapsto 1]\,[t\mapsto 1], t:=x;\Downarrow s\,[x\mapsto 9]\,[y\mapsto 1]\,[t\mapsto 9]}}{\emptyset;\,[z\mapsto f];\,[f\mapsto f_b]\Vdash s\,[x\mapsto 1]\,[y\mapsto 9], t:=x;\Downarrow s\,[x\mapsto 9]\,[y\mapsto 1]\,[t\mapsto 9]}\quad \dfrac{}{\emptyset;\,[z\mapsto f];\,[f\mapsto f_b]\Vdash s\,[x\mapsto 1]\,[y\mapsto 9], t:=x;\Downarrow s\,[x\mapsto 9]\,[y\mapsto 1]\,[t\mapsto 9]}\quad \dfrac{}{\emptyset;\emptyset;\,[f\mapsto f_b]\Vdash s\,[x\mapsto 9]\,[y\mapsto 1]\,[t\mapsto 9], x:=y;\Downarrow s\,[x\mapsto 1]\,[y\mapsto 1]\,[t\mapsto 9]}}{\emptyset;\,[z\mapsto f];\,[f\mapsto f_b]\Vdash s\,[x\mapsto 1]\,[y\mapsto 9], (t:=x;x:=y;)\Downarrow s\,[x\mapsto 1]\,[y\mapsto 1]\,[t\mapsto 9]}\quad \dfrac{}{\emptyset;\,[z\mapsto f];\,[f\mapsto f_b]\Vdash s\,[x\mapsto 1]\,[y\mapsto 9], (t:=x;x:=y;)\Downarrow s\,[x\mapsto 1]\,[y\mapsto 1]\,[t\mapsto 9]}\quad \dfrac{}{\emptyset;\emptyset;\,[f\mapsto f_b]\Vdash s\,[x\mapsto 1]\,[y\mapsto 1]\,[t\mapsto 9], y:=t;\Downarrow s\,[x\mapsto 1]\,[y\mapsto 9]\,[t\mapsto 9]}}{\emptyset;\,[z\mapsto f];\,[f\mapsto f_b]\Vdash s\,[x\mapsto 1]\,[y\mapsto 9], (t:=x;x:=y;y:=t;)\Downarrow s\,[x\mapsto 1]\,[y\mapsto 9]\,[t\mapsto 9]}$$

In the example above, a signal z triggered a signal handler, which is a function f, before the first command of the program.

Assume there is an invariant such that $\{I\}\,(t := x; x := y; y := t;)\,\{I\}$ or $\{I\}\,f(\,);\,\{I\}$ holds. With pre- and postconditions, it will have the next form: $\{x = X \wedge y = Y\}\,f(\,);\,\{x = Y \wedge y = X\}$. So, in terms of pre- and postconditions, for the inner function call f (via signal handler that arrived at the very beginning) the judgement $\{x = 1 \wedge y = 9\}\,f(\,);\,\{x = 9 \wedge y = 1\}$ is true. However, for the (outer) function call f from a body of the program, the judgement $\{x = X \wedge y = Y\}\,(t := x; x := y; y := t;)\,\{x = Y \wedge y = X\}$ does not hold as we get $\{x = 1 \wedge y = 9\}\,(t := x; x := y; y := t;)\,\{x = 1 \wedge y = 9\}$.

If the pre- and post conditions will have the next form $(x = X \wedge y = Y)$ for P and $(x = X \wedge y = Y) \vee (x = Y \wedge y = X)$ for Q, then $\{x = 1 \wedge y = 9\}\,(t := x; x := y; y := t;)\,\{x = 1 \wedge y = 9\}$ is true.

With signal handling in between: What if the signal z arrives during execution of f?

If we apply the logic to the example below, then obviously $\{(x = X \wedge y = Y)\}\,(t := x; x := y; y := t;)\,\{(x = X \wedge y = Y) \vee (x = Y \wedge y = X)\}$ is not true, as we got $\{(x = 1 \wedge y = 9)\}\,(t := x; x := y; y := t;)\,\{(x = 1 \wedge y = 1)\}$ at the end.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset;\emptyset;\,[f\mapsto f_b]\Vdash s\,[x\mapsto 1]\,[y\mapsto 9]\,[t\mapsto 1], t:=x;\Downarrow s\,[x\mapsto 1]\,[y\mapsto 9]\,[t\mapsto 1]}\quad \emptyset;\emptyset;\,[f\mapsto f_b]\Vdash s\,[x\mapsto 1]\,[y\mapsto 9]\,[t\mapsto 1], x:=y;\Downarrow s\,[x\mapsto 9]\,[y\mapsto 9]\,[t\mapsto 1]}{\emptyset;\emptyset;\,[f\mapsto f_b]\Vdash s\,[x\mapsto 1]\,[y\mapsto 9]\,[t\mapsto 1], (t:=x;x:=y;)\Downarrow s\,[x\mapsto 9]\,[y\mapsto 9]\,[t\mapsto 1]}\quad \emptyset;\emptyset;\,[f\mapsto f_b]\Vdash s\,[x\mapsto 9]\,[y\mapsto 9]\,[t\mapsto 1], y:=t;\Downarrow s\,[x\mapsto 9]\,[y\mapsto 1]\,[t\mapsto 1]}{\emptyset;\emptyset;\,[f\mapsto f_b]\Vdash s\,[x\mapsto 1]\,[y\mapsto 9]\,[t\mapsto 1], (t:=x;x:=y;y:=t;)\Downarrow s\,[x\mapsto 9]\,[y\mapsto 1]\,[t\mapsto 1]}\quad [f\mapsto f_b](f)=(t:=x;x:=y;y:=t;)}{\emptyset;\emptyset;\,[f\mapsto f_b]\Vdash s\,[x\mapsto 1]\,[y\mapsto 9]\,[t\mapsto 1], (t:=x;x:=y;y:=t;)\Downarrow s\,[x\mapsto 9]\,[y\mapsto 1]\,[t\mapsto 1]}}{\emptyset;\emptyset;\,[f\mapsto f_b]\Vdash s\,[x\mapsto 1]\,[y\mapsto 9]\,[t\mapsto 1], f(\,);\Downarrow s\,[x\mapsto 9]\,[y\mapsto 1]\,[t\mapsto 1]}\quad [z\mapsto f](z)=f}{\emptyset;\emptyset;\,[f\mapsto f_b]\Vdash s\,[x\mapsto 1]\,[y\mapsto 9]\,[t\mapsto 1], f(\,);\Downarrow s\,[x\mapsto 9]\,[y\mapsto 1]\,[t\mapsto 1]}\quad \emptyset;\emptyset;\,[f\mapsto f_b]\Vdash s\,[x\mapsto 9]\,[y\mapsto 1]\,[t\mapsto 1], x:=y;\Downarrow s\,[x\mapsto 1]\,[y\mapsto 1]\,[t\mapsto 1]}{\emptyset;\,[z\mapsto f];\,[f\mapsto f_b]\Vdash s\,[x\mapsto 1]\,[y\mapsto 9]\,[t\mapsto 1], x:=y;\Downarrow s\,[x\mapsto 1]\,[y\mapsto 1]\,[t\mapsto 1]}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset;\emptyset;\,[f\mapsto f_b]\Vdash s\,[x\mapsto 1]\,[y\mapsto 9]\,[t\mapsto 1], t:=x;\Downarrow s\,[x\mapsto 1]\,[y\mapsto 9]\,[t\mapsto 1]}\quad \emptyset;\,[z\mapsto f];\,[f\mapsto f_b]\Vdash s\,[x\mapsto 1]\,[y\mapsto 9]\,[t\mapsto 1], x:=y;\Downarrow s\,[x\mapsto 1]\,[y\mapsto 1]\,[t\mapsto 1]}{\emptyset;\,[z\mapsto f];\,[f\mapsto f_b]\Vdash s\,[x\mapsto 1]\,[y\mapsto 9], (t:=x;x:=y;)\Downarrow s\,[x\mapsto 1]\,[y\mapsto 1]\,[t\mapsto 1]}\quad \emptyset;\,[z\mapsto f];\,[f\mapsto f_b]\Vdash s\,[x\mapsto 1]\,[y\mapsto 9], (t:=x;x:=y;)\Downarrow s\,[x\mapsto 1]\,[y\mapsto 1]\,[t\mapsto 1]}{\emptyset;\,[z\mapsto f];\,[f\mapsto f_b]\Vdash s\,[x\mapsto 1]\,[y\mapsto 9], (t:=x;x:=y;y:=t;)\Downarrow s\,[x\mapsto 1]\,[y\mapsto 1]\,[t\mapsto 1]}$$

Precondition (or invariant) of the signal in the signal binding should hold from the very beginning. In case with function, there is no such limitation, as programmer has control over when he/she calls the function. However, as we use functions in signal bindings, the preconditions for them should hold from the very beginning as well. (This applies only to the functions that are used as signal handlers.)

$$\Sigma, z : P_z \triangleright Q_z; \Phi \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta$$

$$\Sigma; \Phi \vdash \{P_z\}\, f(\,);\, \{Q_z\}\ \texttt{throws}\ \eta$$

$$\frac{\Sigma\ \textbf{stable}\ P_z \triangleright Q_z}{\Sigma; \Phi \vdash \{P \wedge P_z\}\, \texttt{bind/1}\ z\ \texttt{to}\ f\ \texttt{in}\ c_B\, \{Q \wedge (P_z \vee Q_z)\}\ \texttt{throws}\ \eta}$$

$$z : P_z \triangleright Q_z; f : P_z \triangleright Q_z \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta$$

$$\emptyset; f : P_z \triangleright Q_z \vdash \{P_z\}\, f(\,);\, \{Q_z\}\ \texttt{throws}\ \eta$$

$$\frac{\emptyset\ \textbf{stable}\ P_z \triangleright Q_z}{\emptyset; f : P_z \triangleright Q_z \vdash \{P \wedge P_z\}\, \texttt{bind/1}\ z\ \texttt{to}\ f\ \texttt{in}\ c_B\, \{Q \wedge (P_z \vee Q_z)\}\ \texttt{throws}\ \eta}$$

Let $c_B = f(\,);$, but first, let $c_B = (t := x; x := y; y := t;)$; this is just to avoid long trees at this stage. Actually, $(t := x; x := y; y := t;)$ is a body of the function f. Let P and $P_z = \{x = X \wedge y = Y\}$, Q and $Q_z = \{x = Y \wedge y = X\}$. Let's consider the left subtree (c_B):

$$\frac{z : P_z \triangleright Q_z; f : P_z \triangleright Q_z \vdash \{P_z\}\, t := x;\, \{P_1\}\ \texttt{throws}\ \eta \qquad z : P_z \triangleright Q_z; f : P_z \triangleright Q_z \vdash \{P_1\}\, x := y;\, \{P_2\}\ \texttt{throws}\ \eta}{z : P_z \triangleright Q_z; f : P_z \triangleright P_2 \vdash \{P_z\}\, (t := x; x := y;)\, \{P_2\}\ \texttt{throws}\ \eta}$$

$$\frac{z : P_z \triangleright Q_z; f : P_z \triangleright Q_z \vdash \{P_z\}\, (t := x; x := y;)\, \{P_2\}\ \texttt{throws}\ \eta \qquad z : P_z \triangleright Q_z; f : P_z \triangleright Q_z \vdash \{P_2\}\, y := t;\, \{Q_z\}\ \texttt{throws}\ \eta}{z : P_z \triangleright Q_z; f : P_z \triangleright Q_z \vdash \{P_z\}\, (t := x; x := y; y := t;)\, \{Q_z\}\ \texttt{throws}\ \eta}$$

What does happen when we substitute the place holders?

$$\dfrac{\dfrac{\{x = X \wedge y = Y\}\, t := x;\, \{x = X \wedge y = Y\}}{(x = X \wedge y = Y)\ \textbf{stable}\ z : P_z \triangleright Q_z \qquad \ldots}}{z : P_z \triangleright Q_z; f : P_z \triangleright Q_z \vdash \{x = X \wedge y = Y\}\, t := x;\, \{P_1\}\, \texttt{throws}\ \eta}$$

$$\dfrac{\dfrac{z : P_z \triangleright Q_z; f : P_z \triangleright Q_z \vdash \{x = X \wedge y = Y\}\, t := x;\, \{P_1\}\, \texttt{throws}\ \eta}{z : P_z \triangleright Q_z; f : P_z \triangleright Q_z \vdash \{P_1\}\, x := y;\, \{P_2\}\, \texttt{throws}\ \eta}}{z : P_z \triangleright Q_z; f : P_z \triangleright P_2 \vdash \{x = X \wedge y = Y\}\, (t := x; x := y;)\, \{P_2\}\, \texttt{throws}\ \eta}$$

$$\dfrac{\dfrac{z : P_z \triangleright Q_z; f : P_z \triangleright Q_z \vdash \{x = X \wedge y = Y\}\, (t := x; x := y;)\, \{P_2\}\, \texttt{throws}\ \eta}{z : P_z \triangleright Q_z; f : P_z \triangleright Q_z \vdash \{P_2\}\, y := t;\, \{x = Y \wedge y = X\}\, \texttt{throws}\ \eta}}{z : P_z \triangleright Q_z; f : P_z \triangleright Q_z \vdash \{x = X \wedge y = Y\}\, (t := x; x := y; y := t;)\, \{x = Y \wedge y = X\}\, \texttt{throws}\ \eta}$$

We cannot derive this tree as stability assumption does not hold; $(x = X \wedge y = Y)$ **stable** $z : P_z \triangleright Q_z$ is equivalent to $(x = X \wedge y = Y)$ **stable** $z : (x = X \wedge y = Y) \triangleright (x = Y \wedge y = X)$.

9.2.13 Non-linear Interference - Part 2

In Section 9.2.12, we considered scenarios where an interrupting instance of the function (let's say f) corrupts (via signal handler) the interrupted instance of f; thus, the interrupted instance of the function f returns unreliable results. In this Section 9.2.13, we will address a situation where the interrupting function f returns unreliable results, as its preconditions do not hold inside of the interrupted function f; in other words, inside of its own body.

Programmer has control over the functions, such that where to call them in the program. The preconditions of the function should be known as well, it is its interface in some sense. Thus, before one calls the function he should satisfy the function's preconditions. Then, he may expect a reliable outcome of the function. However, when the function is interrupted via signal handler, programmer has no control over which preconditions are satisfied, as signals arrive nondeterministically. Assume that the function f performs an

operation of division by a variable y. Thus, it's precondition could have the next form $y \neq 0$ (or $y > 0$). So, when one calls the function f form the body of a command c, he should check that y is not equal to zero or assign any value that is not equal to zero. When the function is called via the signal handler, y could have any value.

To discuss this example, we use the following the binding rule.

$$z : I_z \triangleright I_z; f : I_z \triangleright I_z \vdash \{P\}\, c_B\, \{Q\}\, \texttt{throws}\ \eta$$

$$z : I_z \triangleright I_z; f : I_z \triangleright I_z \vdash \{I_z\}\, f(\,);\, \{I_z\}\, \texttt{throws}\ \eta$$

$$\frac{I_z \textbf{ stable } I_z \triangleright I_z}{\emptyset; f : I_z \triangleright I_z \vdash \{P \wedge I_z\}\, \texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, c_B\, \{Q \wedge I_z\}\, \texttt{throws}\ \eta}$$

When one binds a new signal hander, its preconditions should hold at that point. In this version of the language signal handlers are functions, so the function's preconditions (function that is used as a signal handler) should hold at the point of signal binding. Moreover, according to the stability assumptions (implicit and explicit), that preconditions should hold even during execution of the main command. For example, if c_B assigns zero to y, then stability assumptions (in a derivation tree for the atomic command) will not hold.

Example Let's consider this situation in terms of operational semantics. We define a local variable x in the most outer scope: $\texttt{local}\ x\ \texttt{in}\ c_1$. Thus, from any inner scope these variables will be considered as global. Assume that in c_1 we have a function f defined in the next form $\texttt{fun}\ f\ (\,) = (x := 0; x{+}{+};)\ \texttt{in}\ c_2$. Finally, we define c_2 as $\texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, c_3$, where c_3 is $f(\,);$. Thus, a function call f might be interrupted by itself. If signal arrives before the first command of the function runs or after function call completes, then the result of such interleaving is equal to sequentially composed function calls. Shortly, x will be equal 1 after every function call. However, if signal arrives during the function f call, then the result will be different (e.g.: $x = 2$).

We skip straightforward steps and start from c_3 execution level. For space saving, f_b

$$\dfrac{\begin{array}{c}[\,z \mapsto f\,];\emptyset;\,[\,f \mapsto f_b\,] \Vdash\, [\,x \mapsto \_\,], x := 0; \Downarrow s\,[\,x \mapsto 0\,] \\ [\,z \mapsto f\,];\emptyset;\,[\,f \mapsto f_b\,] \Vdash\, [\,x \mapsto 0\,], x{+}{+}; \Downarrow s\,[\,x \mapsto 1\,]\end{array}}{[\,z \mapsto f\,];\emptyset;\,[\,f \mapsto f_b\,] \Vdash\, [\,x \mapsto \_\,], (x := 0; x{+}{+};) \Downarrow s\,[\,x \mapsto 1\,]}$$

$$\dfrac{\begin{array}{c}[\,f \mapsto f_b\,](f) = (x := 0; x{+}{+};) \\ [\,z \mapsto f\,];\emptyset;\,[\,f \mapsto f_b\,] \Vdash\, [\,x \mapsto \_\,], (x := 0; x{+}{+};) \Downarrow s\,[\,x \mapsto 1\,]\end{array}}{[\,z \mapsto f\,];\emptyset;\,[\,f \mapsto f_b\,] \Vdash\, [\,x \mapsto \_\,], f(\,); \Downarrow s\,[\,x \mapsto 1\,]}$$

Figure 9.33: Signal z never arrives

replaces $x := 0; x{+}{+};$.

Please note: it is not important whether the signal z is one-shot or persistent in these examples, as the body interfere with the signal handler in non-linear way. Thus, it is enough for any signal to interrupt the body at least once to have a destructive effect.

Case 1: signal z never arrives, Figure 9.33.

Case 2: signal is handled before the function call, Figure 9.34.

Case 3: signal is handled during the function call, Figure 9.35.

Case *: corresponding program addressed in logic, Figure 9.36.

To address the following program

$$\texttt{fun}\ f\ (\,) = (x := 0; x{+}{+};)\ \texttt{in}\ (\texttt{bind}\ z\ \texttt{to}\ f\ \texttt{in}\ f(\,);)$$

in our logic, we cannot skip the initial steps, where the signals are bound and functions are defined. If we start from the same level as operational semantics examples, we can get the derivation tree as in Figure 9.36.

However, a derived subtree in Figure 9.36, does not depict all the details. So let's derive a tree from the very beginning, but after the local variable x has been defined and initialised to 1, Figure 9.37. Stability assumption holds trivially, as a signal context is empty before a signal z binding.

The function f is defined before the signal binding; thus, in the right subtree, where the stability assumptions of the function body should be checked, the signal context is

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset;\emptyset;\,[\,f \mapsto f_b\,] \Vdash\, [\,x \mapsto \_\,], x := 0; \Downarrow s\,[\,x \mapsto 0\,]} \quad \dfrac{}{\emptyset;\emptyset;\,[\,f \mapsto f_b\,] \Vdash\, [\,x \mapsto 0\,], x{+}{+}; \Downarrow s\,[\,x \mapsto 1\,]}}{\emptyset;\emptyset;\,[\,f \mapsto f_b\,] \Vdash\, [\,x \mapsto \_\,], (x := 0; x{+}{+};) \Downarrow s\,[\,x \mapsto 1\,]} \quad [\,f \mapsto f_b\,](f) = (x := 0; x{+}{+};)}{\emptyset;\emptyset;\,[\,f \mapsto f_b\,] \Vdash s\,[\,x \mapsto \_\,], f(\,); \Downarrow s\,[\,x \mapsto 1\,]} \quad [\,z \mapsto f\,](z) = f}{\emptyset;\emptyset;\,[\,f \mapsto f_b\,] \Vdash s\,[\,x \mapsto \_\,], f(\,); \Downarrow s\,[\,x \mapsto 1\,]} \quad \dfrac{\dfrac{}{\emptyset;\,[\,z \mapsto f\,];\,[\,f \mapsto f_b\,] \Vdash\, [\,x \mapsto 1\,], x := 0; \Downarrow s\,[\,x \mapsto 0\,]}}{[\,z \mapsto f\,];\emptyset;\,[\,f \mapsto f_b\,] \Vdash\, [\,x \mapsto \_\,], x := 0; \Downarrow s\,[\,x \mapsto 0\,]}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{[\,z \mapsto f\,];\emptyset;\,[\,f \mapsto f_b\,] \Vdash\, [\,x \mapsto \_\,], x := 0; \Downarrow s\,[\,x \mapsto 0\,]} \quad \dfrac{}{[\,z \mapsto f\,];\emptyset;\,[\,f \mapsto f_b\,] \Vdash\, [\,x \mapsto 0\,], x{+}{+}; \Downarrow s\,[\,x \mapsto 1\,]}}{[\,z \mapsto f\,];\emptyset;\,[\,f \mapsto f_b\,] \Vdash\, [\,x \mapsto \_\,], (x := 0; x{+}{+};) \Downarrow s\,[\,x \mapsto 1\,]} \quad [\,f \mapsto f_b\,](f) = (x := 0; x{+}{+};)}{[\,z \mapsto f\,];\emptyset;\,[\,f \mapsto f_b\,] \Vdash\, [\,x \mapsto \_\,], f(\,); \Downarrow s\,[\,x \mapsto 1\,]}$$

Figure 9.34: Signal is handled before the function call

$$\dfrac{\dfrac{\emptyset;\emptyset;\,[\,f \mapsto f_b\,] \Vdash\, [\,x \mapsto 0\,], x := 0; \Downarrow s\,[\,x \mapsto 0\,] \qquad \emptyset;\emptyset;\,[\,f \mapsto f_b\,] \Vdash\, [\,x \mapsto 0\,], x{+}{+}; \Downarrow s\,[\,x \mapsto 1\,]}{\emptyset;\emptyset;\,[\,f \mapsto f_b\,] \Vdash\, [\,x \mapsto 0\,], (x := 0; x{+}{+};) \Downarrow s\,[\,x \mapsto 1\,]} \qquad [\,f \mapsto f_b\,](f) = (x := 0; x{+}{+};)}{\emptyset;\emptyset;\,[\,f \mapsto f_b\,] \Vdash s\,[\,x \mapsto 0\,], f(\,); \Downarrow s\,[\,x \mapsto 1\,]}$$

$$\dfrac{\dfrac{\emptyset;\,[\,z \mapsto f\,];\,[\,f \mapsto f_b\,] \Vdash\, [\,x \mapsto \_\,], x := 0; \Downarrow s\,[\,x \mapsto 0\,] \qquad [\,z \mapsto f\,](z) = f}{[\,z \mapsto f\,];\emptyset;\,[\,f \mapsto f_b\,] \Vdash\, [\,x \mapsto \_\,], x := 0; \Downarrow s\,[\,x \mapsto 1\,]} \qquad \dfrac{\dfrac{[\,z \mapsto f\,];\emptyset;\,[\,f \mapsto f_b\,] \Vdash\, [\,x \mapsto \_\,], x := 0; \Downarrow s\,[\,x \mapsto 1\,] \qquad [\,z \mapsto f\,];\emptyset;\,[\,f \mapsto f_b\,] \Vdash\, [\,x \mapsto 1\,], x{+}{+}; \Downarrow s\,[\,x \mapsto 2\,]}{[\,z \mapsto f\,];\emptyset;\,[\,f \mapsto f_b\,] \Vdash\, [\,x \mapsto \_\,], (x := 0; x{+}{+};) \Downarrow s\,[\,x \mapsto 2\,]} \qquad [\,f \mapsto f_b\,](f) = (x := 0; x{+}{+};)}{[\,z \mapsto f\,];\emptyset;\,[\,f \mapsto f_b\,] \Vdash\, [\,x \mapsto \_\,], f(\,); \Downarrow s\,[\,x \mapsto 2\,]}$$

Figure 9.35: Signal is handled during the function call

$$\dfrac{(f : I_z \triangleright I_z)(f) = (I_z \triangleright I_z)}{z : I_z \triangleright I_z; f : I_z \triangleright I_z \vdash \{I_z\}\, f(\,);\, \{I_z\}\, \texttt{throws}\, \eta}$$

assume that x initilalised to 1, and $I_z = (x = 1)$

$$\dfrac{(f : x = 1 \triangleright x = 1)(f) = (x = 1 \triangleright x = 1)}{z : x = 1 \triangleright x = 1; f : x = 1 \triangleright x = 1 \vdash \{x = 1\}\, f(\,);\, \{x = 1\}\, \texttt{throws}\, \eta}$$

Figure 9.36: Non-reentrant function bound to a signal

$$\dfrac{\dfrac{(f : x = 1 \triangleright x = 1)(f) = (x = 1 \triangleright x = 1)}{z : x = 1 \triangleright x = 1; f : x = 1 \triangleright x = 1 \vdash \{(x = 1)\}\, f(\,);\, \{(x = 1)\}} \quad \dfrac{(f : x = 1 \triangleright x = 1)(f) = (x = 1 \triangleright x = 1)}{\emptyset; f : x = 1 \triangleright x = 1 \vdash \{(x = 1)\}\, f(\,);\, \{(x = 1)\}}}{\emptyset; f : x = 1 \triangleright x = 1 \vdash \{(x = 1)\}\, \texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);\, \{(x = 1)\}} \quad \dfrac{\emptyset; f : x = 1 \triangleright x = 1 \vdash \{(x = 1)\}\, x := 0;\, \{(x = 0)\} \quad \emptyset; f : x = 1 \triangleright x = 1 \vdash \{(x = 0)\}\, x{+}{+};\, \{(x = 1)\}}{\emptyset; f : x = 1 \triangleright x = 1 \vdash \{(x = 1)\}\, (x := 0; x{+}{+};)\, \{(x = 1)\}}$$

$$\dfrac{\emptyset; f : x = 1 \triangleright x = 1 \vdash \{(x = 1)\}\, \texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);\, \{(x = 1)\} \quad \emptyset; f : x = 1 \triangleright x = 1 \vdash \{(x = 1)\}\, (x := 0; x{+}{+};)\, \{(x = 1)\}}{\emptyset; \emptyset \vdash \{(x = 1)\}\, \texttt{fun}\ f\ (\,) = (x := 0; x{+}{+};)\ \texttt{in}\ (\texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);)\, \{(x = 1)\}}$$

Figure 9.37: Non-reentrant function bound to a signal - simplified

empty. Later in the tree, when the function f is called, the signal context is no longer empty. Actually, even if the signal context is nonempty initially, the function body will miss all signals that will be installed later. It is neither a feature nor a bug, it is a limitation of the logic. For more details please see Section 9.4.1.

9.2.14 Commands Instead of Functions

Let's compare examples of a language with functions in Figure 9.38 and Figure 9.39 to a language without functions in Figure 9.40 and Figure 9.41.

In the examples in Figure 9.39 and Figure 9.41, a signal context Σ was non-empty from the very beginning. Let's address almost the same program ($\texttt{bind}\, z\, \texttt{to}\, (x := 0; x{+}{+};)\, \texttt{in}\, (x := 0; x{+}{+};)$) that was studied above, but without functions.

Case 1: signal z never arrives, Figure 9.42.

Case 2: signal is handled before the first command, Figure 9.43.

Case 3: signal is handled after the first command, Figure 9.44.

Case *: corresponding program addressed in logic, Figure 9.45.

To address this program ($\texttt{bind}\, z\, \texttt{to}\, (x := 0; x{+}{+};)\, \texttt{in}\, (x := 0; x{+}{+};)$) in our logic, we start almost from the very beginning. We assume that x initialised to 1 beforehand,

$$\frac{\begin{array}{c}\Sigma, z_1 : I_z \triangleright I_z, z_2 : I_z \triangleright I_z; \Phi; \emptyset \vdash \{P\}\, c\, \{Q\}\ \texttt{throws}\ \eta \\ \Sigma, z_1 : I_z \triangleright I_z; \Phi; \emptyset \vdash \{I_z\}\, f(\,);\, \{I_z\}\ \texttt{throws}\ \eta \\ \Sigma, z_2 : I_z \triangleright I_z\ \textbf{stable}\ I_z\end{array}}{\begin{array}{c}\Sigma, z_1 : I_z \triangleright I_z; \Phi; \emptyset \vdash \{P \wedge I_z\}\, \texttt{bind}\, z_2\, \texttt{to}\, f\, \texttt{in}\, c\, \{Q \wedge I_z\}\ \texttt{throws}\ \eta \\ \Sigma, z_1 : I_z \triangleright I_z; \Phi; \emptyset \vdash \{P \wedge I_z\}\, \texttt{bind}\, z_2\, \texttt{to}\, f\, \texttt{in}\, c\, \{Q \wedge I_z\}\ \texttt{throws}\ \eta\end{array}}$$

$$\frac{\begin{array}{c}\Sigma; \Phi; \emptyset \vdash \{I_z\}\, f(\,);\, \{I_z\}\ \texttt{throws}\ \eta \\ \Sigma\ \textbf{stable}\ I_z\end{array}}{\begin{array}{c}\Sigma; \Phi; \emptyset \vdash \{P \wedge I_z\}\, \texttt{bind}\, z_1\, \texttt{to}\, f\, \texttt{in}\, (\texttt{bind}\, z_2\, \texttt{to}\, f\, \texttt{in}\, c)\, \{Q \wedge I_z\}\ \texttt{throws}\ \eta \\ \Sigma; f : I_z \triangleright I_z; \emptyset \vdash \{P \wedge I_z\}\, \texttt{bind}\, z_1\, \texttt{to}\, f\, \texttt{in}\, (\texttt{bind}\, z_2\, \texttt{to}\, f\, \texttt{in}\, c)\, \{Q \wedge I_z\}\ \texttt{throws}\ \eta\end{array}}$$

$$\frac{\begin{array}{c}\Sigma; f : I_z \triangleright I_z; \emptyset \vdash \{I_z\}\, c_f\, \{I_z\}\ \texttt{throws}\ \eta \\ \Sigma; f : I_z \triangleright I_z; \emptyset \overset{R}{\vdash} c_f\end{array}}{\Sigma; \emptyset; \emptyset \vdash \{P \wedge I_z\}\, \texttt{fun}\ f\ (\,) = c_f\ \texttt{in}\ (\texttt{bind}\, z_1\, \texttt{to}\, f\, \texttt{in}\, (\texttt{bind}\, z_2\, \texttt{to}\, f\, \texttt{in}\, c))\, \{Q \wedge I_z\}\ \texttt{throws}\ \eta}$$

Figure 9.38: In language with functions - Op. sem.

Let $c = a_1$ and $c_f = a_2$.

$$\frac{\begin{array}{c}\{I_z\}\, a_2\, \{I_z\} \\ I_z\ \textbf{stable}\ \Sigma \qquad I_z\ \textbf{stable}\ \Sigma \qquad \Sigma\ \textbf{stable}\ (I_z \triangleright I_z) \qquad \eta\ \textbf{stable}\ \Sigma\end{array}}{\Sigma; f : I_z \triangleright I_z; \emptyset \vdash \{I_z\}\, a_2\, \{I_z\}\ \texttt{throws}\ \eta}$$

$$\frac{(f : I_z \triangleright I_z)(f) = I_z\ \triangleright\ I_z}{\Sigma; f : I_z \triangleright I_z; \emptyset \vdash \{I_z\}\, f(\,);\, \{I_z\}\ \texttt{throws}\ \eta}$$

$$\frac{(f : I_z \triangleright I_z)(f) = I_z\ \triangleright\ I_z}{\Sigma, z_1 : I_z \triangleright I_z; f : I_z \triangleright I_z; \emptyset \vdash \{I_z\}\, f(\,);\, \{I_z\}\ \texttt{throws}\ \eta}$$

Figure 9.39: In language with functions - Logic

$$\frac{\begin{array}{c}\Sigma, z_1 : I_z \triangleright I_z, z_2 : I_z \triangleright I_z \vdash \{P\}\, c\, \{Q\}\ \texttt{throws}\ \eta \\ \Sigma, z_1 : I_z \triangleright I_z \vdash \{I_z\}\, c_h\, \{I_z\}\ \texttt{throws}\ \eta \\ \Sigma, z_2 : I_z \triangleright I_z\ \textbf{stable}\ I_z\end{array}}{\begin{array}{c}\Sigma, z_1 : I_z \triangleright I_z \vdash \{P \wedge I_z\}\, \texttt{bind}\, z_2\, \texttt{to}\, c_h\, \texttt{in}\, c\, \{Q \wedge I_z\}\ \texttt{throws}\ \eta \\ \Sigma, z_1 : I_z \triangleright I_z \vdash \{P \wedge I_z\}\, \texttt{bind}\, z_2\, \texttt{to}\, c_h\, \texttt{in}\, c\, \{Q \wedge I_z\}\ \texttt{throws}\ \eta \\ \Sigma \vdash \{I_z\}\, c_h\, \{I_z\}\ \texttt{throws}\ \eta \\ \Sigma\ \textbf{stable}\ I_z\end{array}}$$

$$\Sigma \vdash \{P \wedge I_z\}\, \texttt{bind}\, z_1\, \texttt{to}\, c_h\, \texttt{in}\, (\texttt{bind}\, z_2\, \texttt{to}\, c_h\, \texttt{in}\, c)\, \{Q \wedge I_z\}\ \texttt{throws}\ \eta$$

Figure 9.40: In language with commands - Op.sem

Let $c = a_1$ and $c_h = a_2$.

$$\frac{\begin{array}{c}\{I_z\}\, a_2\, \{I_z\} \\ I_z\ \textbf{stable}\ \Sigma \qquad I_z\ \textbf{stable}\ \Sigma \qquad \Sigma\ \textbf{stable}\ (I_z \triangleright I_z) \qquad \eta\ \textbf{stable}\ \Sigma\end{array}}{\Sigma \vdash \{I_z\}\, a_2\, \{I_z\}\ \texttt{throws}\ \eta}$$

$$\frac{\begin{array}{c}\{I_z\}\, a_2\, \{I_z\} \\ I_z\ \textbf{stable}\ \Sigma, z_1 : I_z \triangleright I_z \qquad I_z\ \textbf{stable}\ \Sigma, z_1 : I_z \triangleright I_z \\ \Sigma, z_1 : I_z \triangleright I_z\ \textbf{stable}\ (I_z \triangleright I_z) \qquad \eta\ \textbf{stable}\ \Sigma, z_1 : I_z \triangleright I_z\end{array}}{\Sigma, z_1 : I_z \triangleright I_z \vdash \{I_z\}\, a_2\, \{I_z\}\ \texttt{throws}\ \eta}$$

Figure 9.41: In language with commands - Logic

$$\frac{\dfrac{[\, z \mapsto (x := 0; x + +;)\,]; \emptyset \Vdash [\, x \mapsto \_\,], x := 0; \Downarrow s\, [\, x \mapsto 0\,] \qquad [\, z \mapsto (x := 0; x + +;)\,]; \emptyset \Vdash [\, x \mapsto 0\,], x + +; \Downarrow s\, [\, x \mapsto 1\,]}{[\, z \mapsto (x := 0; x + +;)\,]; \emptyset \Vdash [\, x \mapsto \_\,], (x := 0; x + +;) \Downarrow s\, [\, x \mapsto 1\,]}}{\emptyset; \emptyset \Vdash [\, x \mapsto \_\,], \texttt{bind}\, z\, \texttt{to}\, (x := 0; x + +;)\, \texttt{in}\, (x := 0; x + +;) \Downarrow s\, [\, x \mapsto 1\,]}$$

Figure 9.42: Signal z never arrives

$$\frac{\emptyset;\emptyset \Vdash s\,[\,x \mapsto \_\,], x := 0; \Downarrow s\,[\,x \mapsto 0\,] \qquad \emptyset;\emptyset \Vdash s\,[\,x \mapsto 0\,], x + +; \Downarrow s\,[\,x \mapsto 1\,]}{\emptyset;\emptyset \Vdash s\,[\,x \mapsto \_\,], (x := 0; x + +;) \Downarrow s\,[\,x \mapsto 1\,]}$$

$$\frac{\begin{array}{l}[\,z \mapsto (x := 0; x + +;)\,](z) = (x := 0; x + +;) \\ \quad \emptyset;\emptyset \Vdash s\,[\,x \mapsto \_\,], (x := 0; x + +;) \Downarrow s\,[\,x \mapsto 1\,] \\ \quad [\,z \mapsto (x := 0; x + +;)\,];\emptyset \Vdash\ [\,x \mapsto 1\,], x := 0; \Downarrow s\,[\,x \mapsto 0\,]\end{array}}{\begin{array}{l}[\,z \mapsto (x := 0; x + +;)\,];\emptyset \Vdash\ [\,x \mapsto \_\,], x := 0; \Downarrow s\,[\,x \mapsto 0\,] \\ \quad [\,z \mapsto (x := 0; x + +;)\,];\emptyset \Vdash\ [\,x \mapsto \_\,], x := 0; \Downarrow s\,[\,x \mapsto 0\,] \\ \qquad [\,z \mapsto (x := 0; x + +;)\,];\emptyset \Vdash\ [\,x \mapsto 0\,], x + +; \Downarrow s\,[\,x \mapsto 1\,]\end{array}}$$

$$\frac{\begin{array}{l}[\,z \mapsto (x := 0; x + +;)\,];\emptyset \Vdash\ [\,x \mapsto \_\,], (x := 0; x + +;) \Downarrow s\,[\,x \mapsto 1\,]\end{array}}{\emptyset;\emptyset \Vdash\ [\,x \mapsto \_\,], \texttt{bind}\, z\, \texttt{to}\, (x := 0; x + +;)\, \texttt{in}\, (x := 0; x + +;) \Downarrow s\,[\,x \mapsto 1\,]}$$

Figure 9.43: Signal is handled before the first command

$$\frac{\emptyset;\emptyset \Vdash s\,[\,x \mapsto 0\,], x := 0; \Downarrow s\,[\,x \mapsto 0\,] \qquad \emptyset;\emptyset \Vdash s\,[\,x \mapsto 0\,], x + +; \Downarrow s\,[\,x \mapsto 1\,]}{\emptyset;\emptyset \Vdash s\,[\,x \mapsto 0\,], (x := 0; x + +;) \Downarrow s\,[\,x \mapsto 1\,]}$$

$$\frac{\begin{array}{l}[\,z \mapsto (x := 0; x + +;)\,];\emptyset \Vdash\ [\,x \mapsto \_\,], x := 0; \Downarrow s\,[\,x \mapsto 0\,] \\ \qquad [\,z \mapsto (x := 0; x + +;)\,](z) = (x := 0; x + +;) \\ \qquad\qquad \emptyset;\emptyset \Vdash s\,[\,x \mapsto 0\,], (x := 0; x + +;) \Downarrow s\,[\,x \mapsto 1\,]\end{array}}{[\,z \mapsto (x := 0; x + +;)\,];\emptyset \Vdash\ [\,x \mapsto \_\,], x := 0; \Downarrow s\,[\,x \mapsto 1\,]}$$

$$\frac{\begin{array}{l}[\,z \mapsto (x := 0; x + +;)\,];\emptyset \Vdash\ [\,x \mapsto \_\,], x := 0; \Downarrow s\,[\,x \mapsto 1\,] \\ \quad [\,z \mapsto (x := 0; x + +;)\,];\emptyset \Vdash\ [\,x \mapsto 1\,], x + +; \Downarrow s\,[\,x \mapsto 2\,]\end{array}}{\begin{array}{l}[\,z \mapsto (x := 0; x + +;)\,];\emptyset \Vdash\ [\,x \mapsto \_\,], (x := 0; x + +;) \Downarrow s\,[\,x \mapsto 2\,]\end{array}}$$

$$\frac{\begin{array}{l}[\,z \mapsto (x := 0; x + +;)\,];\emptyset \Vdash\ [\,x \mapsto \_\,], (x := 0; x + +;) \Downarrow s\,[\,x \mapsto 2\,]\end{array}}{\emptyset;\emptyset \Vdash\ [\,x \mapsto \_\,], \texttt{bind}\, z\, \texttt{to}\, (x := 0; x + +;)\, \texttt{in}\, (x := 0; x + +;) \Downarrow s\,[\,x \mapsto 2\,]}$$

Figure 9.44: Signal is handled after the first command

and $I_z = (x = 1)$ as shown in Figure 9.45. All the problems with stability assumptions for this program arise because the program's code is **non-reentrant**, but the RLTS could prevent this. For details please refer to Section 9.2.11.

Even if the preconditions are weakened a little bit, for example, for any initial value of x the given code returns 1, it will still fail. We use _ to represent any value. For an example with weakened preconditions, see Figure 9.46.

9.2.15 Functions and RLTS

Let's apply the ideas that appeared in Section 9.2.11, for the program

$$\texttt{fuN}\ f\ (\ ) = (x := 0; x{+}{+};)\ \texttt{in}\ (\texttt{bind}\ z\ \texttt{to}\ f\ \texttt{in}\ f(\ );)$$

Operational semantics tree derivations are identical to the cases 1, 2 and 3 in Section 9.2.13. Logic rules with embedded RLTS are described in Section 9.2.5.

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{(x=1)\}\, x := 0;\, \{(x=0)\}}}{(x=1)\ \mathbf{stable}\ z : x=1 \triangleright x=1}}{(x=0)\ \mathbf{stable}\ z : x=1 \triangleright x=1}}{(x=1)\ \mathbf{stable}\ ((x=1) \triangleright (x=0))}}{(x=1)\ \mathbf{stable}\ ((x=1) \triangleright (x=0))}}{z : x=1 \triangleright x=1 \vdash \{(x=1)\}\, x := 0;\, \{(x=0)\}} \qquad \frac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{(x=0)\}\, x :{+}{+};\, \{(x=1)\}}}{(x=0)\ \mathbf{stable}\ z : x=1 \triangleright x=1}}{(x=1)\ \mathbf{stable}\ z : x=1 \triangleright x=1}}{(x=1)\ \mathbf{stable}\ ((x=0) \triangleright (x=1))}}{(x=1)\ \mathbf{stable}\ ((x=0) \triangleright (x=1))}}{z : x=1 \triangleright x=1 \vdash \{(x=0)\}\, x{+}{+};\, \{(x=1)\}}$$

$$\frac{z : x=1 \triangleright x=1 \vdash \{(x=1)\}\, x := 0;\, \{(x=0)\} \qquad z : x=1 \triangleright x=1 \vdash \{(x=0)\}\, x{+}{+};\, \{(x=1)\}}{z : x=1 \triangleright x=1 \vdash \{(x=1)\}\, (x := 0; x{+}{+};)\, \{(x=1)\}}$$

$$\frac{\emptyset \vdash \{(x=1)\}\, x := 0;\, \{(x=0)\} \qquad \emptyset \vdash \{(x=0)\}\, x{+}{+};\, \{(x=1)\}}{\emptyset \vdash \{(x=1)\}\, (x := 0; x{+}{+};)\, \{(x=1)\}}$$

$$\frac{z : x=1 \triangleright x=1 \vdash \{(x=1)\}\, (x := 0; x{+}{+};)\, \{(x=1)\} \qquad \emptyset \vdash \{(x=1)\}\, (x := 0; x{+}{+};)\, \{(x=1)\}}{\emptyset \vdash \{(x=1)\}\, \mathtt{bind}\, z\, \mathtt{to}\, (x := 0; x{+}{+};)\, \mathtt{in}\, (x := 0; x{+}{+};)\, \{(x=1)\}}$$

Figure 9.45: Non-reentrant code bound to a signal

$$
\frac{\begin{array}{c}
\{(x = \_)\}\, x := 0;\ \{(x = 0)\} \\
(x = \_)\ \textbf{stable}\ z : x = \_ \triangleright x = 1 \\
(x = 0)\ \textbf{stable}\ z : x = \_ \triangleright x = 1 \\
(x = \_)\ \textbf{stable}\ ((x = \_) \triangleright (x = 0)) \\
(x = 1)\ \textbf{stable}\ ((x = \_) \triangleright (x = 0))
\end{array}}{z : x = \_ \triangleright x = 1 \vdash \{(x = \_)\}\, x := 0;\ \{(x = 0)\}}
$$

$$
\frac{\begin{array}{c}
\{(x = 0)\}\, x : ++;\ \{(x = 1)\} \\
(x = 0)\ \textbf{stable}\ z : x = \_ \triangleright x = 1 \\
(x = 1)\ \textbf{stable}\ z : x = \_ \triangleright x = 1 \\
(x = \_)\ \textbf{stable}\ ((x = 0) \triangleright (x = 1)) \\
(x = 1)\ \textbf{stable}\ ((x = 0) \triangleright (x = 1))
\end{array}}{z : x = \_ \triangleright x = 1 \vdash \{(x = 0)\}\, x++;\ \{(x = 1)\}}
$$

$$
\frac{\begin{array}{c}
z : x = \_ \triangleright x = 1 \vdash \{(x = \_)\}\, x := 0;\ \{(x = 0)\} \\
z : x = \_ \triangleright x = 1 \vdash \{(x = 0)\}\, x++;\ \{(x = 1)\}
\end{array}}{z : x = \_ \triangleright x = 1 \vdash \{(x = \_)\}\, (x := 0; x++;)\, \{(x = 1)\}}
$$

$$
\frac{\emptyset \vdash \{(x = \_)\}\, x := 0;\ \{(x = 0)\} \qquad \emptyset \vdash \{(x = 0)\}\, x++;\ \{(x = 1)\}}{\emptyset \vdash \{(x = \_)\}\, (x := 0; x++;)\, \{(x = 1)\}}
$$

$$
\frac{\begin{array}{c}
z : x = \_ \triangleright x = 1 \vdash \{(x = \_)\}\, (x := 0; x++;)\, \{(x = 1)\} \\
\emptyset \vdash \{(x = \_)\}\, (x := 0; x++;)\, \{(x = 1)\}
\end{array}}{\emptyset \vdash \{(x = \_)\}\, \texttt{bind}\, z\, \texttt{to}\, (x := 0; x++;)\, \texttt{in}\, (x := 0; x++;)\, \{(x = 1)\}}
$$

Figure 9.46: Non-reentrant code bound to a signal. Weakened preconditions.

It is assumed that the local variable x has been defined and initialised to 1.

$$\dfrac{\dfrac{(f : x = 1 \triangleright x = 1)(f) = (x = 1 \triangleright x = 1)}{z : x = 1 \triangleright x = 1; \emptyset; f : x = 1 \triangleright x = 1 \vdash \{(x = 1)\}\, f(\,);\, \{(x = 1)\}} \quad \dfrac{(\emptyset)(f) = \lightning}{\emptyset; \emptyset; \emptyset \vdash \{(x = 1)\}\, f(\,);\, \{(x = 1)\}}}{\emptyset; \emptyset; f : x = 1 \triangleright x = 1 \vdash \{(x = 1)\}\, \texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,); \{(x = 1)\}}$$

or

$$\dfrac{\dfrac{(\emptyset)(f) = \lightning}{z : x = 1 \triangleright x = 1; \emptyset; \emptyset \vdash \{(x = 1)\}\, f(\,);\, \{(x = 1)\}} \quad \dfrac{(f : x = 1 \triangleright x = 1)(f) = (x = 1 \triangleright x = 1)}{\emptyset; \emptyset; f : x = 1 \triangleright x = 1 \vdash \{(x = 1)\}\, f(\,);\, \{(x = 1)\}}}{\emptyset; \emptyset; f : x = 1 \triangleright x = 1 \vdash \{(x = 1)\}\, \texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,); \{(x = 1)\}}$$

$$\dfrac{\emptyset; \emptyset; f : x = 1 \triangleright x = 1 \vdash \{(x = 1)\}\, x := 0;\, \{(x = 0)\} \qquad \emptyset; \emptyset; f : x = 1 \triangleright x = 1 \vdash \{(x = 0)\}\, x + +;\, \{(x = 1)\}}{\emptyset; \emptyset; f : x = 1 \triangleright x = 1 \vdash \{(x = 1)\}\, (x := 0; x + +;)\, \{(x = 1)\}} \quad \emptyset; \emptyset; f : x = 1 \triangleright x = 1 \vdash \{(x = 1)\}\, \texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,); \{(x = 1)\}$$

$$\dfrac{\emptyset; \emptyset; f : x = 1 \triangleright x = 1 \vdash \{(x = 1)\}\, (x := 0; x + +;)\, \{(x = 1)\}}{\emptyset; \emptyset; \emptyset \vdash \{(x = 1)\}\, \texttt{fuN}\ f\ (\,) = (x := 0; x + +;)\ \texttt{in}\ (\texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);)\, \{(x = 1)\}}$$

Despite the fact that stability assumption holds trivially (initially empty signal context), RLTS indicates that this program is unsafe.

9.2.16 Motivation of the Reentrancy Granularity

In C language, `static` variables are not stored in stack. Thus, every instance of the function uses the same piece of memory. On the other hand, `automatic` variables (a default option) are placed on stack. Thus, whenever the function is called and the new stack frame is created, a new piece of memory is allocated for the `automatic` variables.

If an `automatic` variable is declared inside of a function, then the scope and visibility of that variable is limited to the body of the function. The visibility and scope of that variables could be limited, if another variable with the same name is declared later in the code of the function body.

Functions that use `automatic` variables only are **reentrant**. When another (second) instance of the same function interrupts execution of itself (first instance), a separate piece of memory is provided to the `automatic` variables in a freshly created stack frame.

Visibility and scope of the `static` variables differs form the `automatic` ones. If the `static` variable is declared inside of the function body, then the scope of that variable is limited to the body of the function and the visibility of that variable is limited to all instances of the function. In other words, the same piece of memory that corresponds to the `static` variable will be shared between all instances of the function where it has been declared. However, the `static` variable declared in one function is not visible from another one. If the `static` variable is declared in an outer scope of the function definition, then the scope and the visibility of that variable is limited to the outer scope. That means the variable is visible from any instance of any function defined in that scope.

Functions that use `static` variables in non-atomic way are **non-reentrant**. If two instances of the same function will interfere concurrently with each other, the outcome is no longer reliable, as the same piece of memory is shared concurrently. The `global` and the `static` variables are different, but have some similarities in terms of scope and visibility when declared in an outer scope that in turn can have function definitions.

To address reentrancy closer to the real-life implementation, we need to imitate the `static` variables. Actually, our definition and understanding of the function reentrancy strongly relies on the visibility of the `static` variables. A function is non-reentrant, if two instances of it have access to the same piece of memory. In real life situations, that is the case when the function adopts `static` variables. The same situation could happen, it the function works with the `global` variables; `global` from the function's point of view.

In our language, we do not have explicit `global` or `static` variables, but we can

imitate the required settings of visibility by declaring local variables in outer scopes. Therefore, in our language, we can study the problem of reentrancy that corresponds to the real-life situations, despite the fact that we don't have an explicit notion of the `static` variables.

Example: Idealisation for the static variable

We consider two examples, with and without RLTS in place.

Remark 9.2.6 In logic with signals, exceptions and RLTS the format of a judgement is

$$\Sigma; \Phi; \Psi \vdash \{P\}\, c\, \{Q\}\ \texttt{throws}\ \eta$$

And in logic with signals and exceptions, but without RLTS the format of a judgement is as follows

$$\Sigma; \Phi \vdash \{P\}\, c\, \{Q\}\ \texttt{throws}\ \eta$$

Remark 9.2.7 Rules for the operational semantics are given in Section 9.1.

In logic (with RLTS):

$$\dfrac{\dfrac{(f : P_f \triangleright Q_f)(f) = P_f \ \triangleright\ Q_f}{z : P_f \triangleright Q_f; \emptyset; f : P_f \triangleright Q_f \vdash \{P_f\}\, f(\,);\, \{Q_f\}\ \texttt{throws}\ \eta} \qquad \dfrac{\emptyset(f) = \lightning}{\emptyset; \emptyset; \emptyset; \vdash \{P_f\}\, f(\,);\, \{Q_f\}\ \texttt{throws}\ \eta}}{\emptyset; \emptyset; f : P_f \triangleright Q_f \vdash \{P_f\}\, \texttt{bind/1}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);\, \{Q_f\}\ \texttt{throws}\ \eta}$$

$$\emptyset; \emptyset; f : P_f \triangleright Q_f \vdash \{P_f\}\, \texttt{bind/1}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);\, \{Q_f\}\ \texttt{throws}\ \eta$$

$$\emptyset; \emptyset; f : P_f \triangleright Q_f \vdash \{P_f\}\, (x := 0; x + +;)\, \{Q_f\}\ \texttt{throws}\ \eta$$

$$\dfrac{\emptyset; \emptyset; f : P_f \triangleright Q_f \vdash \{P_f\}\, (\texttt{fuN}\ f\ (\,) = (x := 0; x + +;)\ \texttt{in}\ (\texttt{bind/1}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);))\, \{Q_f\}\ \texttt{throws}\ \eta}{\emptyset; \emptyset; \emptyset \vdash \{P_f\}\, (\texttt{fuN}\ f\ (\,) = (x := 0; x + +;)\ \texttt{in}\ (\texttt{bind/1}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);))\, \{Q_f\}\ \texttt{throws}\ \eta}$$

$$\emptyset; \emptyset; \emptyset \vdash \{P_f\}\, (\texttt{fuN}\ f\ (\,) = (x := 0; x + +;)\ \texttt{in}\ (\texttt{bind/1}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);))\, \{Q_f\}\ \texttt{throws}\ \eta$$

$$x \notin \texttt{FV}(P_f)$$

$$x \notin \texttt{FV}(Q_f)$$

$$\emptyset; \emptyset; \emptyset \vdash \{P_f\}\, \texttt{local}\ x\ \texttt{in}\ (\texttt{fuN}\ f\ (\,) = (x := 0; x + +;)\ \texttt{in}\ (\texttt{bind/1}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);))\, \{Q_f\}\ \texttt{throws}\ \eta$$

In logic (without RLTS):

$$\dfrac{\dfrac{(f : P_f \triangleright Q_f)(f) = P_f \ \triangleright \ Q_f}{z : P_f \triangleright Q_f; f : P_f \triangleright Q_f \vdash \{P_f\}\, f(\,);\, \{Q_f\}\, \texttt{throws}\, \eta}}{\dfrac{\dfrac{(f : P_f \triangleright Q_f)(f) = P_f \ \triangleright \ Q_f}{\emptyset; f : P_f \triangleright Q_f \vdash \{P_f\}\, f(\,);\, \{Q_f\}\, \texttt{throws}\, \eta}}{\emptyset; f : P_f \triangleright Q_f \vdash \{P_f\}\, \texttt{bind/1}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);\, \{Q_f\}\, \texttt{throws}\, \eta}}$$

$$\dfrac{\dfrac{\emptyset; f : P_f \triangleright Q_f \vdash \{P_f\}\, \texttt{bind/1}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);\, \{Q_f\}\, \texttt{throws}\, \eta}{\emptyset; f : P_f \triangleright Q_f \vdash \{P_f\}\, (x := 0; x + +;)\, \{Q_f\}\, \texttt{throws}\, \eta}}{\emptyset; \emptyset \vdash \{P_f\}\, (\texttt{fuN}\ f\ (\,) = (x := 0; x + +;)\ \texttt{in}\ (\texttt{bind/1}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);))\, \{Q_f\}\, \texttt{throws}\, \eta}$$

$$\dfrac{\dfrac{\emptyset; \emptyset \vdash \{P_f\}\, (\texttt{fuN}\ f\ (\,) = (x := 0; x + +;)\ \texttt{in}\ (\texttt{bind/1}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);))\, \{Q_f\}\, \texttt{throws}\, \eta}{x \notin \texttt{FV}(P_f)}}{x \notin \texttt{FV}(Q_f)}$$

$$\emptyset; \emptyset \vdash \{P_f\}\, \texttt{local}\ x\ \texttt{in}\ (\texttt{fuN}\ f\ (\,) = (x := 0; x + +;)\ \texttt{in}\ (\texttt{bind/1}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);))\, \{Q_f\}\, \texttt{throws}\, \eta$$

In op.sem.: (safe run)

$$\dfrac{\dfrac{\dfrac{[\,z \mapsto f\,]; \emptyset; [\,f \mapsto (x := 0; x + +;)\,] \Vdash s_1\,[\,p_1 \mapsto 0\,], x := 0; [\,x \mapsto p_1\,] \Downarrow s_1\,[\,p_1 \mapsto 0\,] \quad [\,z \mapsto f\,]; \emptyset; [\,f \mapsto (x := 0; x + +;)\,] \Vdash s_1\,[\,p_1 \mapsto 0\,], x + +; [\,x \mapsto p_1\,] \Downarrow s_1\,[\,p_1 \mapsto 1\,]}{[\,z \mapsto f\,]; \emptyset; [\,f \mapsto (x := 0; x + +;)\,] \Vdash s_1\,[\,p_1 \mapsto 0\,], (x := 0; x + +;)\,[\,x \mapsto p_1\,] \Downarrow s_1\,[\,p_1 \mapsto 1\,]} \quad F(f) = (x := 0; x + +;)}{[\,z \mapsto f\,]; \emptyset; [\,f \mapsto (x := 0; x + +;)\,] \Vdash s_1\,[\,p_1 \mapsto 0\,], f(\,); [\,x \mapsto p_1\,] \Downarrow s_1\,[\,p_1 \mapsto 1\,]} \quad f \in \mathrm{dom}(F)}{\emptyset; \emptyset; [\,f \mapsto (x := 0; x + +;)\,] \Vdash s_1\,[\,p_1 \mapsto 0\,], \texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);\, [\,x \mapsto p_1\,] \Downarrow s_1\,[\,p_1 \mapsto 1\,]}$$

$$\dfrac{\dfrac{\emptyset; \emptyset; \emptyset \Vdash s_1\,[\,p_1 \mapsto 0\,], \texttt{fun}\ f\ (\,) = (x := 0; x + +;)\ \texttt{in}\ (\texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);)\,[\,x \mapsto p_1\,] \Downarrow s_1\,[\,p_1 \mapsto 1\,]}{\emptyset; \emptyset; \emptyset \Vdash s_1, \texttt{local}\ x\ \texttt{in}\ (\texttt{fun}\ f\ (\,) = (x := 0; x + +;)\ \texttt{in}\ (\texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, f(\,);)) \Downarrow s_1}$$

In op.sem.: (interference with the signal handler)

$$\dfrac{\dfrac{\dfrac{}{F(f) = (x := 0; x + +;);}\quad \dfrac{}{\emptyset;\emptyset;\,[\,f \mapsto (x := 0; x + +;)\,] \Vdash s_1\,[\,p_1 \mapsto 0\,], (x := 0; x + +;)\,[\,x \mapsto p_1\,] \Downarrow s_1\,[\,p_1 \mapsto 1\,]}}{\emptyset;\emptyset;\,[\,f \mapsto (x := 0; x + +;)\,] \Vdash s_1\,[\,p_1 \mapsto 0\,], f(\,);[\,x \mapsto p_1\,] \Downarrow s_1\,[\,p_1 \mapsto 1\,]}\quad \dfrac{}{[\,z \mapsto f\,];\emptyset;\,[\,f \mapsto (x := 0; x + +;)\,] \Vdash s_1\,[\,p_1 \mapsto 0\,], x := 0;\,[\,x \mapsto p_1\,] \Downarrow s_1\,[\,p_1 \mapsto 0\,]}}{[\,z \mapsto f\,];\emptyset;\,[\,f \mapsto (x := 0; x + +;)\,] \Vdash s_1\,[\,p_1 \mapsto 0\,], x := 0;\,[\,x \mapsto p_1\,] \Downarrow s_1\,[\,p_1 \mapsto 1\,]}$$

$$\dfrac{\dfrac{\dfrac{}{([\,z \mapsto f\,])(z) = f}\quad \dfrac{}{\emptyset;\emptyset;\,[\,f \mapsto (x := 0; x + +;)\,] \Vdash s_1\,[\,p_1 \mapsto 0\,], f(\,);[\,x \mapsto p_1\,] \Downarrow s_1\,[\,p_1 \mapsto 1\,]}}{[\,z \mapsto f\,];\emptyset;\,[\,f \mapsto (x := 0; x + +;)\,] \Vdash s_1\,[\,p_1 \mapsto 0\,], x := 0;\,[\,x \mapsto p_1\,] \Downarrow s_1\,[\,p_1 \mapsto 1\,]}\quad \dfrac{}{[\,z \mapsto f\,];\emptyset;\,[\,f \mapsto (x := 0; x + +;)\,] \Vdash s_1\,[\,p_1 \mapsto 1\,], x + +;\,[\,x \mapsto p_1\,] \Downarrow s_1\,[\,p_1 \mapsto 2\,]}}{[\,z \mapsto f\,];\emptyset;\,[\,f \mapsto (x := 0; x + +;)\,] \Vdash s_1\,[\,p_1 \mapsto 0\,], (x := 0; x + +;)\,[\,x \mapsto p_1\,] \Downarrow s_1\,[\,p_1 \mapsto 2\,]}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{F(f) = (x := 0; x + +;)}\quad \dfrac{}{[\,z \mapsto f\,];\emptyset;\,[\,f \mapsto (x := 0; x + +;)\,] \Vdash s_1\,[\,p_1 \mapsto 0\,], (x := 0; x + +;)\,[\,x \mapsto p_1\,] \Downarrow s_1\,[\,p_1 \mapsto 2\,]}}{[\,z \mapsto f\,];\emptyset;\,[\,f \mapsto (x := 0; x + +;)\,] \Vdash s_1\,[\,p_1 \mapsto 0\,], f(\,);\,[\,x \mapsto p_1\,] \Downarrow s_1\,[\,p_1 \mapsto 2\,]}\quad \dfrac{}{f \in \mathrm{dom}(F)}}{[\,z \mapsto f\,];\emptyset;\,[\,f \mapsto (x := 0; x + +;)\,] \Vdash s_1\,[\,p_1 \mapsto 0\,], f(\,);\,[\,x \mapsto p_1\,] \Downarrow s_1\,[\,p_1 \mapsto 2\,]}}{\emptyset;\emptyset;\,[\,f \mapsto (x := 0; x + +;)\,] \Vdash s_1\,[\,p_1 \mapsto 0\,], \mathtt{bind}\, z\, \mathtt{to}\, f\, \mathtt{in}\, f(\,);\,[\,x \mapsto p_1\,] \Downarrow s_1\,[\,p_1 \mapsto 2\,]}}{\emptyset;\emptyset;\emptyset \Vdash s_1\,[\,p_1 \mapsto 0\,], \mathtt{fun}\ f\ (\,) = (x := 0; x + +;)\ \mathtt{in}\ (\mathtt{bind}\, z\, \mathtt{to}\, f\, \mathtt{in}\, f(\,);)\,[\,x \mapsto p_1\,] \Downarrow s_1\,[\,p_1 \mapsto 2\,]}}{\emptyset;\emptyset;\emptyset \Vdash s_1, \mathtt{local}\ x\ \mathtt{in}\ (\mathtt{fun}\ f\ (\,) = (x := 0; x + +;)\ \mathtt{in}\ (\mathtt{bind}\, z\, \mathtt{to}\, f\, \mathtt{in}\, f(\,);)) \Downarrow s_1}$$

9.3 Experimental Material

In this section we present and discuss some experimental material to show how the logic with Reentrancy Linear Type System may be further developed in the future.

9.3.1 Signal Binding and Functions

Functions become signal handlers in two steps. First of all, a function f is added into the function context Φ by a function definition rule. Then, a signal z is bound to the function f from the function context Φ by a signal binding rule. In the previous version of the logic rules (Logic Rules are presented in Section 9.2.5), stability is checked when a function body c_f is no longer visible. The function body c_f is visible only during the function definition rule. Thus, the rules were providing a ground for the TOCTOU situation. For examples, please refer to the Section 9.2.7 and Section 9.2.13.

Auxiliary definitions

We have to add auxiliary definitions (Definition 5.3.1) for the new forms of stability in our program logic; thus, we extend Definition 5.3.1 with two extra points.

Definition 9.3.1 (Stability conditions Extended)

1. For a function context Φ (analogous for Ψ), we write P **stable** Φ if for all $f_j \in \mathrm{dom}(\Phi)$ with $\Phi(f_j) = (P_j \triangleright Q_j)$, it is the case that P **stable** $(P_j \triangleright Q_j)$.

2. We write Φ **stable** $(P \triangleright Q)$ if for all f_j in $\mathrm{dom}(\Phi)$ with $\Phi(f_j) = (P_j \triangleright Q_j)$, we have P_j **stable** $(P \triangleright Q)$ and Q_j **stable** $(P \triangleright Q)$.

9.3.2 Variations of the Logic Rule Updates

Stability checks during function definition

We may update the logic rules to check stability for the functions at the level of the function definition, even before we bind it to the signals. Thus, we may remove the stability checks at the level of signal bindings, as all signal handlers are functions; therefore, their interference with each other was already controlled by the stability checks during the function definitions.

$$\frac{\Sigma;\Phi, f : P_f \triangleright Q_f;\Psi \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta \qquad \Sigma;\Phi, f : P_f \triangleright Q_f;\Psi \overset{R}{\vdash} c_f \qquad \Phi \cup \Psi\ \textbf{stable}\ P_f \triangleright Q_f \qquad P_f\ \textbf{stable}\ \Phi \cup \Psi \qquad Q_f\ \textbf{stable}\ \Phi \cup \Psi}{\Sigma;\Phi;\Psi \vdash \{P\}\, \texttt{fun}\ f\ (\,) = c_f\ \texttt{in}\ c_B\, \{Q\}\ \texttt{throws}\ \eta}$$

$$\frac{\Sigma;\Phi;\Psi, n : P_n \triangleright Q_n \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta \qquad \Sigma;\Phi;\Psi, n : P_n \triangleright Q_n \vdash \{P_n\}\, c_n\, \{Q_n\}\ \texttt{throws}\ \eta \qquad \Phi \cup \Psi\ \textbf{stable}\ P_n \triangleright Q_n \qquad P_n\ \textbf{stable}\ \Phi \cup \Psi \qquad Q_n\ \textbf{stable}\ \Phi \cup \Psi}{\Sigma;\Phi;\Psi \vdash \{P\}\, \texttt{fuN}\ n\ (\,) = c_n\ \texttt{in}\ c_B\, \{Q\}\ \texttt{throws}\ \eta}$$

$$\frac{\Sigma, z : I_z \triangleright I_z;\Phi;\Psi_1 \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta \qquad \Sigma;\Phi;\Psi_2 \vdash \{I_z\}\, f(\,);\, \{I_z\}\ \texttt{throws}\ \eta \qquad \cancel{\Sigma\ \textbf{stable}\ I_z}}{\Sigma;\Phi;\Psi_1, \Psi_2 \vdash \{P \wedge I_z\}\, \texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, c_B\, \{Q \wedge I_z\}\ \texttt{throws}\ \eta}$$

$$\frac{\Sigma, z : P_z \triangleright Q_z;\Phi;\Psi_1 \vdash \{P\}\, c_B\, \{Q\}\ \texttt{throws}\ \eta \qquad \Sigma;\Phi;\Psi_2 \vdash \{P_z\}\, f(\,);\, \{Q_z\}\ \texttt{throws}\ \eta \qquad \cancel{\Sigma\ \textbf{stable}\ P_z \triangleright Q_z}}{\Sigma;\Phi;\Psi_1, \Psi_2 \vdash \{P \wedge P_z\}\, \texttt{bind/1}\, z\, \texttt{to}\, f\, \texttt{in}\, c_B\, \{Q \wedge (P_z \vee Q_z)\}\ \texttt{throws}\ \eta}$$

That may look as an overhead, because not every function becomes a signal handler. However, the "early" stability checks make sense, as when we enforce stability checks at the same time the functions are lifted into the function context, we ensure that no matter how the functions will be used in this environment/context (as signal handlers or just as functions), safety of the program will be satisfied.

Examples

Example 1: It is assumed that the local variable x has been defined and initialised to 1, c_1 equals to $(\texttt{bind}\, z_1\, \texttt{to}\, f_1\, \texttt{in}\, (\texttt{bind}\, z_2\, \texttt{to}\, f_2\, \texttt{in}\, c_2))$. Two signals are bound to two different function names, but literally, they are identical in terms of commands.

With the previous version of the rules, there were two problems: interference of two functions with different names, but identical non-reentrant code was unaddressed and stability assumptions covered the functions at the high level only, but the inner code remained unchecked. The same problems exist with the current version of the rule.

$$\cfrac{\emptyset; z_1, z_2 : x=1 \rhd x=1; \emptyset \vdash \{(x=1)\}\, c_2\, \{(x=1)\} \qquad \cfrac{(f_2 : x=1 \rhd x=1)(f_2) = (x=1 \rhd x=1)}{\emptyset; z_1; f_2 \vdash \{(x=1)\}\, f_2(\,);\, \{(x=1)\}} \qquad (x=1)\ \textbf{stable}\ (x=1) \rhd (x=1)}{\emptyset; z_1 : x=1 \rhd x=1; f_2 \vdash \{(x=1)\}\, \texttt{bind}\, z_2\, \texttt{to}\, f_2\, \texttt{in}\, c_2\, \{(x=1)\}}$$

$$\cfrac{\emptyset; z_1 : x=1 \rhd x=1; f_2 \vdash \{(x=1)\}\, \texttt{bind}\, z_2\, \texttt{to}\, f_2\, \texttt{in}\, c_2\, \{(x=1)\} \qquad \cfrac{(f_1 : x=1 \rhd x=1)(f_1) = (x=1 \rhd x=1)}{\emptyset; \emptyset; f_1 \vdash \{(x=1)\}\, f_1(\,);\, \{(x=1)\}}}{\emptyset; \emptyset; f_1 : x=1 \rhd x=1, f_2 : x=1 \rhd x=1 \vdash \{(x=1)\}\, \texttt{bind}\, z_1\, \texttt{to}\, f_1\, \texttt{in}\, (\texttt{bind}\, z_2\, \texttt{to}\, f_2\, \texttt{in}\, c_2)\, \{(x=1)\}}$$

$$\cfrac{\emptyset; \emptyset; f_1 : x=1 \rhd x=1, f_2 : x=1 \rhd x=1 \vdash \{(x=1)\}\, x := 0;\, \{(x=0)\} \qquad \emptyset; \emptyset; f_1 : x=1 \rhd x=1, f_2 : x=1 \rhd x=1 \vdash \{(x=0)\}\, x++;\, \{(x=1)\}}{\emptyset; \emptyset; f_1 : x=1 \rhd x=1, f_2 : x=1 \rhd x=1 \vdash \{(x=1)\}\, (x := 0; x++;)\, \{(x=1)\}}$$

$$\cfrac{\begin{array}{c}\emptyset; \emptyset; f_1 : x=1 \rhd x=1, f_2 : x=1 \rhd x=1 \vdash \{(x=1)\}\, c_1\, \{(x=1)\} \\ \emptyset; \emptyset; f_1 : x=1 \rhd x=1, f_2 : x=1 \rhd x=1 \vdash \{(x=1)\}\, (x := 0; x++;)\, \{(x=1)\} \\ f_1 : x=1 \rhd x=1\ \textbf{stable}\ f_2 : x=1 \rhd x=1 \qquad (x=1)\ \textbf{stable}\ f_1 : x=1 \rhd x=1\end{array}}{\emptyset; \emptyset; f_1 : x=1 \rhd x=1 \vdash \{(x=1)\}\, \texttt{fuN}\ f_2\ (\,) = (x := 0; x++;)\ \texttt{in}\ c_1\, \{(x=1)\}}$$

$$\cfrac{\emptyset; \emptyset; f_1 : x=1 \rhd x=1 \vdash \{(x=1)\}\, x := 0;\, \{(x=0)\} \qquad \emptyset; \emptyset; f_1 : x=1 \rhd x=1 \vdash \{(x=0)\}\, x++;\, \{(x=1)\}}{\emptyset; \emptyset; f_1 : x=1 \rhd x=1 \vdash \{(x=1)\}\, (x := 0; x++;)\, \{(x=1)\}}$$

$$\cfrac{\begin{array}{c}\emptyset; \emptyset; f_1 : x=1 \rhd x=1 \vdash \{(x=1)\}\, \texttt{fuN}\ f_2\ (\,) = (x := 0; x++;)\ \texttt{in}\ c_1\, \{(x=1)\} \\ \emptyset; \emptyset; f_1 : x=1 \rhd x=1 \vdash \{(x=1)\}\, (x := 0; x++;)\, \{(x=1)\} \\ \emptyset\ \textbf{stable}\ f_1 : (x=1) \rhd (x=1) \qquad (x=1)\ \textbf{stable}\ \emptyset\end{array}}{\emptyset; \emptyset; \emptyset \vdash \{(x=1)\}\, \texttt{fuN}\ f_1\ (\,) = (x := 0; x++;)\ \texttt{in}\ (\texttt{fuN}\ f_2\ (\,) = (x := 0; x++;)\ \texttt{in}\ c_1)\, \{(x=1)\}}$$

Adding stability checks at the level of the function definitions does not work, as the function bodies are non-atomic. The resulting effect is very similar to the original version of the rules, where stability was checked in the signal binding rule. Thus, it could be

a designer's choice: to check stability for a function only if it is registered as a signal handler (via signal binding rule), or to check stability among all functions (via function definition rule). In the second case, more checks to be done, but as a positive effect any function may become a signal handler at any stage later. It should be also noted, that in the second case functions become more restrictive.

Stability checks for the function context as part of the atomic rule

A problem with the previous approach, is that when we add new function to the context, we have access to the function body, as to the block of code. Therefore, we can not check the influence of each atomic command that function body consists of at this level.

Actually, we are trying to address an extreme case when the same non-reentrant block of code is used in two functions with different names. If one follows the basic programming practices, there will be just one function that is attempted to be used in an unsafe way, and the RLTS will determine that the program is unsafe.

However, this is an extreme case of another scenario as well. The same non-reentrant block of code could be used in two different functions, that perform different tasks. Still, one may suggest to adhere to the good programming practices, so the repeated non-reentrant block of code will be encapsulated into the functions; thus, RLTS will be able to check the program for unsafe use of the non-reentrant functions.

We may try to update the logic rules to check stability of the functions at the level of the atomic commands.

$$\{P\}\, a\, \{Q\}$$

$$\frac{\cancel{P \textbf{ stable } \Sigma} \qquad \cancel{Q \textbf{ stable } \Sigma} \qquad \cancel{\Sigma \textbf{ stable } (P \triangleright Q)} \qquad \eta \textbf{ stable } \Sigma \qquad \Phi \cup \Psi \textbf{ stable } P \triangleright Q \qquad P \textbf{ stable } \Phi \cup \Psi \qquad Q \textbf{ stable } \Phi \cup \Psi}{\Sigma; \Phi; \Psi \vdash \{P\}\, a\, \{Q\}\, \texttt{throws}\, \eta}$$

$$\frac{\Sigma, z : I_z \triangleright I_z; \Phi; \Psi_1 \vdash \{P\}\, c_B\, \{Q\}\, \texttt{throws}\, \eta \qquad \Sigma; \Phi; \Psi_2 \vdash \{I_z\}\, f(\,);\, \{I_z\}\, \texttt{throws}\, \eta \qquad \cancel{\Sigma \textbf{ stable } I_z}}{\Sigma; \Phi; \Psi_1, \Psi_2 \vdash \{P \wedge I_z\}\, \texttt{bind}\, z\, \texttt{to}\, f\, \texttt{in}\, c_B\, \{Q \wedge I_z\}\, \texttt{throws}\, \eta}$$

$$\frac{\Sigma, z : P_z \triangleright Q_z; \Phi; \Psi_1 \vdash \{P\}\, c_B\, \{Q\}\, \texttt{throws}\, \eta \qquad \Sigma; \Phi; \Psi_2 \vdash \{P_z\}\, f(\,);\, \{Q_z\}\, \texttt{throws}\, \eta \qquad \cancel{\Sigma \textbf{ stable } P_z \triangleright Q_z}}{\Sigma; \Phi; \Psi_1, \Psi_2 \vdash \{P \wedge P_z\}\, \texttt{bind/1}\, z\, \texttt{to}\, f\, \texttt{in}\, c_B\, \{Q \wedge (P_z \vee Q_z)\}\, \texttt{throws}\, \eta}$$

We check $\Phi \cup \Psi$ **stable** $P \triangleright Q$ instead of Σ **stable** $(P \triangleright Q)$, as the functions could be lifted into the signal context later. We do want to keep P **stable** Σ and Q **stable** Σ, but for the same reason as mentioned above, we might need to replace them with P **stable** $\Phi \cup \Psi$ and Q **stable** $\Phi \cup \Psi$. However, it looks like P **stable** $\Phi \cup \Psi$ and Q **stable** $\Phi \cup \Psi$ enforce too strict restrictions on the use of the functions.

Examples

Example 2: It is assumed that the local variable x has been defined and initialised to 1, c_1 equals to $(\texttt{bind}\, z_1\, \texttt{to}\, f_1\, \texttt{in}\, (\texttt{bind}\, z_2\, \texttt{to}\, f_2\, \texttt{in}\, c_2))$. Two signals are bound to two different function names, but literally, they are identical in terms of commands.

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{(x=1)\}\, x := 0;\, \{(x=0)\}}}{(x=1)\ \textbf{stable}\ (x=1) \triangleright (x=0)}}{(x=1)\ \textbf{stable}\ (x=1) \triangleright (x=1)}}{(x=0)\ \textbf{stable}\ (x=1) \triangleright (x=1)}}{\emptyset;\emptyset;f_1 : x=1 \triangleright x=1 \vdash \{(x=1)\}\, x := 0;\, \{(x=0)\}}}$$

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{(x=0)\}\, x{+}{+};\, \{(x=1)\}}}{(x=1)\ \textbf{stable}\ (x=0) \triangleright (x=1)}}{(x=0)\ \textbf{stable}\ (x=1) \triangleright (x=1)}}{(x=1)\ \textbf{stable}\ (x=1) \triangleright (x=1)}}{\emptyset;\emptyset;f_1 : x=1 \triangleright x=1 \vdash \{(x=0)\}\, x{+}{+};\, \{(x=1)\}}$$

$$\frac{\emptyset;\emptyset;f_1 : x=1 \triangleright x=1 \vdash \{(x=1)\}\, x := 0;\, \{(x=0)\} \qquad \emptyset;\emptyset;f_1 : x=1 \triangleright x=1 \vdash \{(x=0)\}\, x{+}{+};\, \{(x=1)\}}{\emptyset;\emptyset;f_1 : x=1 \triangleright x=1 \vdash \{(x=1)\}\, (x := 0; x{+}{+};)\, \{(x=1)\}}$$

$$\frac{\emptyset;\emptyset;f_1 : x=1 \triangleright x=1 \vdash \{(x=1)\}\, \texttt{fuN}\ f_2\ (\,) = (x := 0; x{+}{+};)\ \texttt{in}\ c_1\, \{(x=1)\} \qquad \emptyset;\emptyset;f_1 : x=1 \triangleright x=1 \vdash \{(x=1)\}\, (x := 0; x{+}{+};)\, \{(x=1)\}}{\emptyset;\emptyset;\emptyset \vdash \{(x=1)\}\, \texttt{fuN}\ f_1\ (\,) = (x := 0; x{+}{+};)\ \texttt{in}\ (\texttt{fuN}\ f_2\ (\,) = (x := 0; x{+}{+};)\ \texttt{in}\ c_1)\, \{(x=1)\}}$$

It does not work with the current configuration. The first explanation is that the function f_1 has been defined as a non-reentrant function. There is no need to check its interference with itself, as by definition it is non-reentrant. Another explanation, is that RLTS would prevent sharing the function f_1 between the main program and the signal handler; thus, the stability checks in the example above are redundant. Therefore, one should not re-check the interference of the function with itself as it is covered by RLTS.

And once again, not every function becomes a signal handler, so enforcing stability checks for every function in the atomic commands is too restrictive. Actually, this approach enforces to use reentrant functions only.

Enforcing stability for the every function during atomic commands run is too restric-

tive. Enforcing stability without accessing function body does not allow to cover all the cases, especially when the function definition and registering of the signal handler are spaced out (do not come one after another). Thus, with current logic we can address reentrancy of the same functions, but not reentrancy of the concurrent code. There would be no TOCTOU situation, if the two functions f_1 and f_2 had the same name (what actually corresponds to a good programming practice and idea of using functions). If the functions were called f, then RLTS would recognise unsafe code in the program.

9.4 Pros and Cons

To address interference between various non-reentrant (unsafe) functions one will need to operate on the level of resources. On the other hand, our RLTS allows addressing function self-reentrancy, which corresponds to the real-life implementations. Studying reentrancy at the level of code and functions is still a challenging task.

What if a reentrant function f binds a signal inside? For example, please see Section 9.2.8, Example 3. When the signal is called, it calls the function f, and the function f binds/rebinds the signal to the same function f once again. Thus, the signal will be unblocked, what may result into the signal handler interruption by itself. Nevertheless, the function is reentrant and such behaviour is a feature of our language.

9.4.1 Limitations of the Logic with RLTS

Please consider the example in Figure 9.37. Stability assumption holds trivially, as a signal context is empty before a signal z binding. The function f is defined before the signal binding; thus, in the right subtree, where the stability assumptions of the function body should be checked, the signal context is empty. Later in the tree, when the function f is called, the signal context is no longer empty. Actually, even if the signal context is nonempty initially, the function body will miss all signals that will be installed later. It is neither a feature nor a bug, it is a limitation of the logic.

CHAPTER 10

LITERATURE REVIEW

In this chapter we summarise relevant findings and supporting ideas we built on from the literature of various domains. Articles have been organised in a few groups, thought the groups are not disjoint and may overlap.

10.1 Exception Handling

Exception handling is not just error signalling mechanism but efficient and flexible control structure that could be used to construct a complex control flow.

Asynchronous Exceptions in Haskell [64] by Simon Marlow, Simon Peyton Jones, Andrew Moran and John Reppy. In most programming languages asynchronous exceptions are heavily restricted, as their improper use may badly influence the reliability of the programs. Haskell probably is the only language that provides both full support and a formal semantics for asynchronous interrupts. In this paper [64], authors use the notions of asynchronous exceptions, asynchronous signalling and interrupts almost interchangeably. The motivational example behind this work is truly asynchronous and nondeterministic signalling of one thread by another.

Concurrent Haskell is an extension of the standard Haskell that is capable of creating new threads and performing communication between threads. This work presents an operational semantics for Concurrent Haskell and extends it with asynchronous exceptions. Extension includes addition of the primitive that enables one thread to asynchronously raise an exception in another thread, block and unblock operations to enable or disable

interrupts in a particular scope.

Exceptional Syntax [7] by Benton et. al. Beside the general introduction to the idea of exception handling, this article extended the usual simply-typed lambda calculus with exceptions (names, types, constructs, rules and etc). Moreover, the corresponding big-step operational semantics has been presented. Finally, authors discussed alternative to ML style handle construct.

Implementation of exception handling [16] by David Chase. This article presents core idea of exception handling and provides some examples at the machine code level. Moreover, various implementation techniques for exception handling have been presented and discussed.

Generic Exception Handling and the Java Monad [63] by Lutz Schröder and Till Mossakowski. In this article, authors characterised Moggi's exception monad transformer by an equational theory, and presented calculi for exception monads that take into account both normal and abrupt termination.

Compiling Exceptions Correctly [44] by Graham Hutton and Joel Wright. This article discusses basic method of compiling exceptions using stack unwinding. Moreover, authors explain and verify that method using functional programming techniques. For this purpose, authors developed a compiler for a small language extended with exceptions, and gave a proof of its correctness with respect to a formal semantics of the language.

Calculating an Exceptional Machine [45] by Graham Hutton and Joel Wright. This article is a continuation of the previous work [44]. Authors developed an abstract machine for evaluation of expressions in the previously designed language with exceptions. The key program transformation technique used in this work is a defunctionalisation that, according to authors, was neglected in recent years.

Exception handling for copyless messaging [53] by Svetlana Jakšić and Luca Padovani. This paper addresses a combination of exceptions and copyless messaging mechanism, where only pointers to messages are exchanged between two processes and the messages themselves are stored in an exchange heap. Authors assume exceptions are

in general unpredictable, where in our work exceptions are triggered at the specific point of the code. Their model includes message sending over channels that are established between peers (endpoints), where we focus on a receiving side only.

Through out the paper, the focus from exceptions slightly moves towards the ideas of transactional memory. Combination of exceptions and resource management is a tricky task. There is a need to keep a track of resources, as in case of abnormal process termination and jump of the control flow via exception, the resources should be handled correctly (destructed or reallocated depending on the situation). Another complication is that handling of an exception at a closest try might not be desirable in a specific circumstances and exception should be propagated further. The existing solution is a proper use of `try-final` block but it has its own drawbacks. As an alternative solution, authors combine static analysis and transaction-like semantics of `try` blocks. Authors study interaction between `try-catch-finally` construct and "resource management", where we are address interaction between signals and exceptions.

The Definition of Standard ML [69] by Milner et. al. This article presents a classic style of exception throwing and handling. We have adopted this style to add exceptions to a big-step operational semantics.

In the lecture slides based on **Handlers of Algebraic Effects** [79] by Gordon D. Plotkin and Matija Pretnar, the focus is on exception handlers that are addressed with Monad. Authors show an explicit interest in designing of the general operational semantics for handling mechanism and combining signals with other control structures in the same language. It was also noted that addressing the notion of "recursion" is important but requires some extra efforts.

10.2 Ghost Variables

Ghost variables are clearly defined in papers like [85, 97, 105] and compared to other forms of variables such as "logical", "auxiliary", "freeze" or "rigid" in [97, 105]. One may

observe that these forms could be identical or orthogonal to the idea of ghost variables.

In **The Spirit of Ghost Code** [33], authors provide a simple ML-style programming language with mutable state and ghost code. The non-interference is enforced by a type system with effects, which also allows the same data types and functions to be used in both regular and ghost code. Authors also discussed a procedure of ghost code erasure and proved its safety using bisimulation.

In **A Marriage of Rely/Guarantee and Separation Logic** [98, 99] and **Modular fine-grained concurrency verification** [97], authors mention that the *ghost* variables become unavoidable when unary postcondition is used. These papers show how to define a two-state predicate from the single-state predicates. This method relies on the next bit of notation '$\leftharpoonup$', which is used to define the state just before the action. Thus, $\overleftarrow{x}$ and x denote the value of the program variable x before and after the action respectively. Authors provide a definition of stability for the binary relations, and using the facts mentioned above, transform it to the form that describes stability of a single state predicate P under a binary relation R.

Authors in **The Rely-Guarantee Method for Verifying Shared Variable Concurrent Programs** [105] briefly introduce the notion of the ghost variables According to this paper, ghost variables might be also called as *logical*, *freeze* or *rigid* variables. In this work, y is used to denote a vector of program variables, whereas the logic variables are indexed with 0. The definition of stability in this paper adopts ghost variables instead of the harpooned ones.

In **The craft of programming** [84], towards the proper procedure declaration, author defines the notions of formal and actual parameters, and addresses the interference between formal parameters and global identifiers The notions of identifier, environment, and state are crucial. `Environment` maps identifiers into their meanings. `State` maps variables into their values. `Variables` is a part of the state of the computation and `Identifiers` are phrases of a program that denote variables. `Expression` is a phrase that describes the computation of a value that depends upon the state of the computation.

For example, let's consider what does $x = y = 17$ mean in an environment η and in a state σ. There are two variants: η maps identifier x to a variable a and y to b, where σ maps both variables a and b into the same value 17; η maps identifiers x and y into the variable a, where σ maps variable a into the value 17. We can observe that identifiers and variables form two levels of abstraction.

Theories of programming languages [85] uses a notion of the ghost variables. This was motivated by the fact that single state assertions can not directly describe a relationship between two states. This is a fundamental book which can open a door to the theory of programming languages. It starts from basic concepts as predicate logic and finally shows how functional language might be build. Author explains all essential concepts such as continuations, concurrency, type systems and polymorphism in detail.

In **A Hoare Logic for Call-by-Value Functional Programs** [82], authors provide a detailed explanation of ghost variables and ghost parameters. One of the key ideas of the parameterization of a function with ghost variables is that when the set of remaining elements is implicit in the stack, a ghost variable might be used in order to refer to it. One may conclude that we may call a ghost identifier as a ghost variable if it is used in the context of an expression, and as a ghost parameter if it is used in the context of a function.

In **Elimination of ghost variables in program logics** [40], authors present a formal model of the ghost variables. However, it is just an introduction before discussion of the problems that emerge with the ghost variables. Therefore, the work towards of elimination of the ghost variables is a key part of the paper. Authors also address semantics of ghost variables and modelling of the extra-functional properties. *Auxiliary variables* are parameters in assertions used in Hoare logic to relate values of program variables to their initial values. They are scoped across one assertion (Hoare triple). According to authors, in concurrency the meanings of auxiliary variable and ghost variable are swapped. Thus, when ghost variables appear in the shared-variable concurrency, they may appear under the name of auxiliary variables.

In **Hoare Type Theory, Polymorphism and Separation** [74], authors shortly introduce ghost variables and mention the logic variables as an equivalent name. Furthermore, the notions of the ghost heap variable and the fresh ghost variable are discussed. In later work, Towards Type-theoretic Semantics for Transactional Concurrency [73], authors describe ghost variables and binary postconditions.

Authors make no difference between "dummy variables", ghost or auxiliary variables in **The "Hoare Logic" of CSP, and All That** [59]. It is also mentioned that some authors avoid ghost variables, whether others find it inelegant to use dummy variables in a program when their values can be easily defined.

In the **Order Theory for Big-Step Semantics** [101], authors provide a big-step semantics for a small call-by-need calculus that supports first-class functions and pairs. Actually, syntax is as for a standard λ calculus with pairs. To address non-termination with a big-step semantics, one should specify a coinductive relation for non-terminating programs, beside inductive relation for terminating ones. Authors present a deterministic call-by-value calculi and a typing system with the soundness proof.

In **Rely-Guarantee References for Refinement Types Over Aliased Mutable Data** [35], authors explain that to preserve assumptions about aliases, all side effects should be restricted in some way. Analogously, in our logic we use stability assumptions for these purposes. In this work on aliases and references, authors adopt ideas of multithreading rely/guarantee. Also, there is an explicit use of the stability. Thus, a brief explanation of the stability in terms of actions through the aliases is presented.

In **Variables as Resource in Hoare Logics** [77], authors use variable-resource descriptions in assertions. They also present two constructs for variable and procedure declaration. It is not mentioned explicitly, but according to the 'local - in - end' construct, authors deal with the notion of "local" variables. It should be noted that comparing to our approach, authors do not provide any kind of resource context splitting for parallel processes. Instead, they introduced a resource context that maps all resource identifiers to their corresponding invariants.

10.3 Understanding Reentrancy

In the real world implementations, there exists a list of asynchronous safe functions that is safe to use inside of the signal handlers [56, 87, 91, 88, 9, 51]. On the contrary, in our approach, we keep track of functions that are unsafe to use inside of the signal handlers and call them non-reentrant. As reentrancy happens nondeterministically via a signal handler, "safety" of the reentrancy should be checked. If the functions that interrupt each other are not from the list of non-reentrant functions, then the reentrancy is safe. If the non-reentrant function is interrupted by any non-reentrant function, then such reentrancy is not safe.

The reentrancy could be defined via graphs [68]. There is a notion of object race, which is not equivalent to a data race, but it is its prerequisite. The relation between object and object calls could be represented via graphs. Therefore, when we talk about reentrancy or nonreentrancy in this approach, technically we mean reentrancy/nonreentrancy of edges of the object graph. Nonreentrancy is important for the object race analysis, as it indicates that the execution of events in two threads does not yield to the object race. Reentrancy in OOP could be defined without graphs, but still points-to graphs are used for the reentrancy analysis [29].

There exists a definition of the reentrant procedure in the OS with multiple users support [90]. One may try to understand the definition of the reentrancy via a definition of the reentrant procedures in operating systems. According to the author [90], there are few obligations to achieve procedure reentrancy. The local data for every user should be stored separately and the program code shouldn't modify itself. The idea of the reentrant procedure is that it can be safely interrupted by another program that calls the same procedure. Safety in this context means that both procedures (interrupted and interrupting) are execute correctly, as if they were executed sequentially. According to the author [90], a reentrant code allows an efficient use of memory, as only a single copy of the code is kept in the main memory, while many applications can call that code.

In **All Unix kernels are reentrant** [11] by Bovet, author addresses kernel reentrancy. In Unix, several processes may be executing in Kernel Mode simultaneously. Nonreentrancy of the kernel means that a process can only be suspended while it is in User mode. To achieve a reentrant kernel only the reentrant functions should be used, or the locking mechanism should be used to ensure that only one process executes a nonreentrant function at a time.

In **Safe and Structured Use of Interrupts in Real-Time and Embedded Software** [81] by John Regehr, nested and reentrant interrupts are compared. The difference between reentrant interrupts and reentrant functions is explained in detail. Overall, this work provides technical and detailed introduction to interrupts.

A Static Analysis to Detect Re-Entrancy in Object Oriented Programs [29] by Manuel Fähndrich, Diego Garbervetsky, and Wolfram Schulte. Authors present their work towards re-entrancy analysis of the object oriented programs. The aim of the analysis, which is based on the pointer analysis, is in detecting of the inconsistent reentrant calls in programs.

Controlling Aspect Reentrancy [94] by Éric Tanter. This work is in domain of the aspect-oriented programming and focused on the reentrant application of aspects. Author classifies the reentrancy, discusses how to avoid the reentrancy, and proposes how to control the reentrancy.

Reasoning about Java's Reentrant Locks [39] by Christian Haack, Marieke Huisman, and Clément Hurlin. Authors develop a verification technique, which is based on a concurrent separation logic, for a concurrent language with reentrant locks. Locks are associated with the resources and reentrancy may lead to the situation when resources are reacquired. The proposed technique is designed to detect the resource reacquisition.

In **A Modality for Safe Resource Sharing and Code Reentrancy** [89], authors introduce and formalize a sharing modality in support of sharing linear resources. One of the interesting approaches used in this work is that sharing is supported without using locks. The code reentrancy is studied in scope of the developed modality.

10.4 Abstract Machines

Abstract machines for programming language implementation [25] by Diehl et. al. This article explains what are the abstract machines and how they could be used. Authors provide an annotated review of various abstract machines designed for different programming paradigms such as imperative, object oriented, functional, logic and concurrent ones.

A Functional Correspondence between Monadic Evaluators and Abstract Machines for Languages with Computational Effects [2] by Ager et. al. In this article, authors construct CEK machines from monadic evaluators for the computational lambda calculus. Furthermore, an abstract machine for stack inspection and exceptions has been presented.

A Resource-Aware Semantics and Abstract Machine for a Functional Language with Explicit Deallocation [71] by Montenegro et. al. In this work, authors step by step built an imperative abstract machine starting from a big-step operational semantics for the first-order eager language. This semantics has been extended with memory consumption annotations and it correctness has been proven with respect to the abstract machine.

The Persistent Abstract Machine [20] by Connor et. al. In this article, heap based storage architecture together with stack frames are explained in detail. In the presented architecture, the stack frame format contains 13 fields. Whereas in signal abstract machine, the stack frame has only 2-3 fields. This article is very useful for understanding of general concepts of abstract machines.

In **From Natural Semantics to Abstract Machines** [1], authors present an approach to the construction of abstract machines from natural semantics descriptions. They start from introducing a class of L-attributed natural semantics. An algorithm for extracting abstract machines from the natural semantics with a correctness proof is presented. Authors discuss applications of the extraction and limitations of the approach.

In **Coinductive big-step operational semantics** [60] by Xavier Leroy and Hervé Grall, authors address connections between the coinductive big-step semantics and the standard small-step semantics. An equivalence of them has been proven and discussed. A small-step semantics is a common choice for proving soundness of type systems, and a big-step semantics is a choice for proving the correctness of program transformations (proof that the program preserves its behaviour). Authors combine two interpretations of the reduction rules (finite and infinite) into the third coinductive interpretation of the rule, which covers finite and infinite reductions. Finally, authors push their big-step semantics further, and extend it with traces. From our experience, traces are a very powerful instrument.

A Simple Semantics and Static Analysis for Stack Inspection [5] describes an access control mechanism realised via the run-time stack inspection, which is a common feature of the JVM and the .NET platforms. Authors discuss a static analysis of safety, which they denote as the absence of security errors. To remove run-time checks, several program transformations are identified and explained. Finally, authors provide a denotational semantics in "eager" form and show its equivalence to the "lazy" semantics via stack inspection.

10.5 Separation Logic and Stability

Local Action and Abstract Separation Logic [14] by Calcagno et. al. This is rather theoretical paper that presents sequential abstract separation logic, trace semantics, and concurrency model. Authors abstracted from the usual definition of the separation logic. Thus, there is no use of a domain of heaps or partial operators in this work. In this article, separation algebra is a cancellative partial commutation monoid.

Certifying low-level programs with hardware interrupts and preemptive threads [32] by Feng et. al. This work presents a program logic for assembly language with interrupts. In their semantics, blocking interrupts transfers ownership of parts of

the heap among interrupt handlers, in the style of concurrent separation logic. Resource separation between the handler and the main body of the program would greatly simplify the stability conditions that need to be checked.

Precision and the Conjunction Rule in Concurrent Separation Logic [36] by Gotsman et. al. The soundness proof for the conjunction rule is known to be nontrivial. This article shows that the proof could be done easier by ensuring that conjunction rule is not used in a derivation, or by introducing *precise* assertions and invariants.

10.6 Logic and Reasoning

Separation Logic for Small-step Cminor [4] by Appel et. al. In this article, authors redesigned imperative programming language Cminor to make it suitable for Hoare Logic reasoning. The main contribution is a separation logic that has been designed for this language. In this work, classical Hoare triples have been extended to sextuples. This approach emerged from the need of dealing with nonlocal control constructs.

Java Program Verification via a Hoare Logic with Abrupt Termination [43] by Jacobs et. al. This article discusses some limitations of Hoare logic. Furthermore, the notion of Hoare logic is extended to deal with abrupt termination and side effects. Despite break, return and continue, the exception also causes an abrupt termination.

Both **Hoare Logic for Java in Isabelle/HOL** [100] and **A Hoare Logic for the Coinductive Trace-Based Big-Step Semantics of While** [72] articles mention various limitations of Hoare logic and then extend basic Hoare logic to handle side-effecting expressions, exceptions, and other nontrivial features.

Modular reasoning for deterministic parallelism [26] by Mike Dodds, Suresh Jagannathan, and Matthew J. Parkinson. According to the authors, *deterministic parallelism* can facilitate the addition of concurrency control protocol into the programs. However, *deterministic parallelism* approach requires automatically injected control constructs to ensure consistency of observable behaviour with the original program' [26]. In

other words, a sequential program is annotated to indicate the sections that can execute concurrently.

The problem definition is as follows: it is hard to create efficient concurrent programs, because it is necessary to coordinate the access of parallel threads to shared data. Authors noted, that effective reasoning about concurrent programs requires *modular* abstractions (reasoning in terms of abstract behaviour) [26]. Also, this paper is focused on verification of *barriers* (concurrency construct), as according to the authors, deterministic parallelism could be achieved by using compiler-injected barriers [26]. Authors highlighted most important aspects concerning *barriers* and decided to use *concurrent abstract predicates*, based on separation logic, to reason in a modular way about implementation of barriers.

The whole section was devoted to specification for deterministic parallelism. One important assumption has been made by authors, they expect that code sections suitable for parallelization are known, and distributed into threads wisely. According to definition, "barriers are associated with resources that are shared between concurrent program segments" [26]. Also it is defined that there are two types of barrier: *grant* barrier that notifies and *wait* barrier that blocks. Another assumption was about compiler, it is expected that it will inject all barriers correctly without modifying original meaning of the program. During explanation of barriers, a notion of *channel* has been introduced.

Authors criticized the opportunity to reason about program behaviour using the operational semantics of the barrier implementation, as any changes to the implementation may require reproving the correctness of the parallelization analysis [26]. Therefore, authors reason about program behaviour in terms of abstract specifications for `grant`, `wait` and `newchan`. Authors used O'Hearn's `Par` rule of concurrent separation logic to reason about the parallel composition of threads. As parallel rule requires preconditions, two predicates have been introduces: `fut` and `req`. As we understand, they are quite similar to the usual R and G predicates from rely-guarantee logic.

Finally, authors introduced a *chain* of channels which allows many thread to access the same resource in a sequence [26].

A Semantic Basis for Local Reasoning [106] by Hongseok Yang and Peter O'Hearn. In this article, authors continue their work on semantics of an approach for reasoning about mutable data structures. One of the key points in this work is that it is possible to avoid frame axioms when certain assumptions are satisfied.

10.7 Soundness, Completeness and Verification

Java Program Verification at Nijmegen: Developments and Perspective [52] by Jacobs et. al. This article gives historical overview of the works dedicated to the Java program verification. This work drew our attention because its aim was to reason about reasonably complex and powerful languages with side-effect in expressions, exceptions and other forms of abrupt control flow.

A Sound and Complete Program Logic for Eiffel[65] and **A Soundness and Completeness of a Program Logic for Eiffel** [66] by Nordio et. al. These articles focused on the program logic for Eiffel, which has a different from other languages (e.g., Java-like languages) exception handling mechanism. Furthermore, interesting observations have been presented on Exception Handling and Once Routines.

Concurrency Verification Introduction to Compositional and Noncompositional Methods [23] by de Roever et.al. and **Tentative steps toward a development method for interfering programs** [55] by Jones. A key contribution of rely/guarantee logic [55, 23] is to introduce *binary* relations on states, in addition to the unary predicates on states known from Hoare logic. Using such relations, we can express that a process *relies* on the interleaved state changes being contained in the relation, that is, (σ_i, σ_{i+1}).

A Structural Proof of the Soundness of Rely/guarantee Rules [17] is a longer version of the [18]. In this work, it is assumed that components of a rely/guarantee specification satisfy certain constraints with respect to each other. Author provides axioms, which we consider as constraints, that show interaction between states and interference, which comes from the environment. The notion of independent expressions is presented

in this work. It has the next form: b **indep** R. What is quite strong restriction as it requires that evaluation of the expression b, which can be used as a precondition, is completely unaffected by interference constrained by R.

25 Years of Formal Proof Cultures [41] by Furio Honsell provides a great retrospective overview of the Formal Proofs. Authors tried to clarify some controversial issues that appear in the theory and practice of Logical Frameworks, including issues that possibly have been the main cause of a diverse specifications.

In **Reasoning about concurrent programs: Refining rely-guarantee thinking** [47], rely-guarantee is embedded into a refinement calculus for concurrent programs, in which programs are developed in steps from an abstract specification. Authors extended the implementation language with specification constructs by adding two new commands to the existing pre and postconditions. These commands are `guar(g)(c)` and `rely(r)(c)`.

A few relevant to our research articles in the concurrency series were found at ACM Queue magazine. They are **Proving the Correctness of Nonblocking Data Structures** [24], **Nonblocking Algorithms and Scalable Multicore Programming** [3], and **The Balancing Act of Choosing Nonblocking Features** [67]. The first article covers nonblocking synchronization and its correctness proof. The second article explores and examines available alternatives to lock-based synchronization. And the last one, addresses design requirements of nonblocking systems.

10.8 Signals and Technical Documentation

As a first step, to understand the nature and current implementation of the signals in Unix like systems, we referred to the following literature: Understanding the Linux Kernel [9], UNIX Systems Programming: Communication, Concurrency and Threads [87], Advanced Programming in the UNIX(R) Environment [91], and The GNU C Library Reference Manual [88].

The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities [27] by Dowd et. al. Attacks on software security published by Zalewski using malicious signal handling were the initial motivation for logic and semantics we have designed and presented in this thesis. A critical ingredient in Zalewski's exploits is the idea to cause the same handler to run twice and thereby corrupt a resource. Signal implementation make it possible to specify that a handler should run at most once, becoming uninstalled after running for the first (and only) time. This is the main justification for making a distinction between one-shot and persistent signal handlers.

INTERNATIONAL STANDARD ISO/IEC 14882 [51], Information technology - Programming languages - C++ . This document contains important notes about signal handlers and exceptions. According to the clause 18.10, paragraph 9, the use of exception throw in signal handlers is restricted. Moreover, the signal handler may only consist of POF ("plain old function"). These are restriction that we overcome in our idealised language with signals.

DWARF Debugging Information Format [19], UNIX International. DWARF format is well known for debugging purposes. However, this format is also used to create special stack frames that are used by stack unwinding process to restore the state and registers. To design unwind function in our signal abstract machine, we referred to this format and that made our language closer to the real life implementations. In our language, the unwind function is called when the control flow jumps via raised exception. To locate corresponding exception handler and clean up the stack from signal and exception handlers that belong to the scope the control flow has left, the unwind function unwinds the stack according to the defined rules.

C++ exception handling for IA-64 [22] by Christophe de Dinechin. This article explains exception handling from the implementation perspective, based on IA-64 architecture. There are many exception-handling solutions and all of them have different impact on performance. Therefore, author addressed some of them (Portable Exception

Handling, and Table-Driven Exception Handling) and highlighted problems that occur during implementation. The most important concepts explained in this article are unwind tables, stack unwinding routine, and landing pads.

Understanding the Linux Kernel [10], Second Edition and **The Linux Programming Interface** [56]. Both books give detailed explanation of signal handling mechanism in Linux. The author of the latter, highlights the interesting differences between various Unix and Linux implementations. The second book also contains many examples and code snippets that make it easier to understand the implementation aspects of signals.

The needs of the formal verification for the embedded systems is discussed in **Synchronous Models for Embedded Software** [12]. Every design flow may need a different kind of formal verification methods. Compared to the traditional software design, the design of embedded software is more challenging, because in addition to the correct implementation of the systems, one has to consider non-functional constraints such as real-time energy consumption, behaviour and reliability [12].

In **A Verified Compiler for Relaxed-Memory Concurrency** [102], authors consider the semantic design and verified compilation of a C-like programming language for concurrent shared-memory computation on x86 multiprocessors. According to the authors, the design of such a language is subtle by several factors. Some of them are: the effects of compiler optimization on concurrent code, the need to support high-performance concurrent algorithms, and the desire for a reasonably simple programming model.

As an introduction in **Investigating Time Properties of Interrupt-Driven Programs** [42], authors discuss interrupt mechanism as a technique to support multi threads, where interrupts are implemented in terms of asynchronous signals and synchronous events. One of the key ideas is to address interrupts as threads. Then the time-operational semantics is presented, where interrupts have time properties; that allows to segregate interrupts which violates some time restrictions. The defined model of the interrupts is rather close to the real implementations, and even includes interrupt requesting, which is

reminiscent of signal sending. The crucial difference to our signals' model is that in this paper, interrupt handlers does not modify the data states of the interrupted programs. In another words, the handler and the program are executed in a separate memory, which eliminates the concurrency problem.

10.9 Continuations

Continuations, functions and jumps [95] by Hayo Thielecke. This paper explains an idea of continuations and how they are connected to the functions and jumps. Jumping is an example of control structure, and if extend the jump with an extra argument it will model the continuation. Moreover, from the compiler perspective, any function call consist of two jumps: from caller to the callee and back. Thus, the ideas of continuation are already inside of modern techniques but we could gain much more by extracting continuations explicitly.

Author presents the notion of continuation passing style, and explains it with comprehensive examples. A short retrospective literature review has been given to show how continuation passing style has been developed. A particular attention was paid to the technique and process of transformation from simple C program into the program with non-returning functions. In general, it consists of two main steps: CPS transformation from functions to jumps with arguments and transformation from jumps (eliminating `goto`) into functions. During the discussion, the `callcc` control operator has been introduced. It has been shown by author, how continuations can improve and enrich programming language (e.g.: power of `callcc`). And finally, continuation ideas have been represented in λ calculus as more general and complete version.

In **Continuations from Generalized Stack Inspection** [78], a question of continuations in modern platforms and languages is discussed. Authors presented a translation from Scheme to the language which deals with continuations in more close to standard stack evaluation model. The crash course to *continuation marks* has been provided.

Stimulus for this work was the fact that implementation of continuations in modern platforms, VMs (Sun JVM, Microsoft CLR) and modern languages meets the problem regarding run-time stack. To be more precise, there are no instructions for installing and saving the run-time stack in these VMs [78]. The general solution in situation when continuations are desired is allocating control stack in the heap. What in turn leads to other disadvantages concerning debugging and security management [78].

Proposed solution contains translation of Scheme programs with `call/cc` into a language with a generalized stack inspection mechanism. To study theory part of this question and develop prototype, the mechanism of "continuation mark" (Scheme) has been used. In another prototype, authors used *exception* `handlers` and exception `throws` to emulate continuation mark mechanism.

Elimination of `call/cc` has been shown by authors in a few steps. First of all, the source language with `call/cc` has been defined. Then, `call/cc` has been replaced by continuation marks what finally result in target language. And as the last bit, defunctionalization is performed what simply means replacing functions with *records*. This transformations require deeper knowledge of Scheme, familiarity with λ calculus and defunctionalization in general.

This paper highlights the following points: complexity needed for implementation of continuations, the way of possible prototyping continuations using exception handlers and throws, projection of continuations theory to practical usage in Web Technologies, different concepts and mechanisms related to continuations.

Monads and composable continuations [103] by Philip Wadler. This paper expects that readers are already familiar with monads and composable continuations, but still it presentes a short summary of monads and continuations. It is explained, how translations from λ calculus to monads could be performed. Author presents continuations as a special case of monad translation. The source language used in this work has been extended with various operators such as escape, shift, and reset. By evaluating monad, author resulted in desired type systems.

A Mathematical Semantics for Handling Full Jumps [92] by Christopher Strachey and Christopher P. Wadsworth. This paper focuses on developing of small programming language with basic continuation-jump support. Syntax and semantic equations are fully and clearly described in an informal way.

Checkpoints and Continuations Instead of Nested Transactions [57] by Eric Koskinen and Maurice Herlihy. The core idea of this paper is representing usefulness of usage continuations together with checkpoints to enforce partial commits and rollbacks. This scheme is compared with nested transactions that may be used to reach similar effect. Concept consists of two main parts: checkpoints which are stored in a runtime computation log and continuations themselves. Authors specified that continuation mechanism can be implemented differently in various languages.

In **Effect for Cooperable and Serializable Threads** [107], authors focus on race freedom and atomicity. This paper contains references to the original Lipton's [62] theory. Authors built an effect system which enables cooperative reasoning. The thing which differs this paper from other is `yield` annotations. It is known that atomic annotations are used to indicate atomic blocks. In turn, `yield` annotations are used to specify program points where interference may occur. Finally, the notion of cooperative traces is introduced and discussed in this paper.

CHAPTER 11

CONCLUSIONS

This chapter summarises the thesis in terms of what has been achieved, surveys previous research in the related domains, and outlines directions for future work. We idealized signal handling in combination with the more familiar exception handling to focus on some of their semantic and logical features. So let us zoom out for a wider view and reflect on what has been achieved in this thesis.

We defined an operational semantics for a language with both exception and signal handling. Signal handlers have persistent and one-shot control flow semantics. Moreover, signal handlers are not restricted to block all other signals during the execution. This makes our operational semantics close to the real life implementations. For example, in Unix like systems signal handlers may interrupt each other leaving a system with a set of nested interrupts. However, the most interesting feature of the operational semantics is the multiplicative way that one-shot signals are propagated. One-shot signal binding is split into two disjoint bindings when a semantics rule has two premises.

To compare how the idealized stack machine models features of real signal implementations, we designed and presented an abstract stack machine for signal handlers. We showed the challenges one may encounter if he decides to implement a language with block structured exception and signal handling. We defined two operational semantics to show their differences and informally discuss a relation between them. In big-step it is easy to describe and reason about block structured constructs. Whereas the small-step operational semantics, in our case abstract machine, is much easier to relate to the real

life implementations.

We defined a program logic with specifications for signal handlers and exception context to address concurrency explicitly. Specifications of the signal handlers limit how the handler interferes with the commands of the program body and other handlers. In contrast to signal binding splitting in semantics, the signal context, which contains specifications of installed handlers, is used additively. Moreover, the signal context is shared rather than split in the logic rules. We adopted the notion of stability from the Rely/Guarantee logic to address how exception and signal handlers with commands influence each other. Some of the capabilities of the designed logic have been shown via examining of the idioms of signal usage. For example, how the logic deals with invariant preserving and signal masking.

No doubt, with unsound logic one may derive properties of a program that do not hold. Therefore, one of the main contributions of this thesis is a soundness proof of the logic with respect to the big-step operational semantics. A path towards soundness proof was not straightforward and resulted in a set of supporting lemmas and conditions. To relate signal handling in operational semantics with logic, we imposed the condition that all signal bindings respect the specification given by the whole signal context. As besides supporting signal handling, our language supports exceptions, we introduced a form of stability condition between signal and exception contexts in the logic.

Understanding of the reentrancy is crucial in concurrent environment. Therefore, the related literature has been carefully analysed and the most important and relevant to our research findings has been summarized in the thesis. We provided a glossary of the reentrancy related terms, to show that they are extremely domain dependent. In particular, to show how diverse the definitions of reentrancy could be, we compared the notion of reentrancy in Object Oriented and Procedural paradigms. Reentrancy is important because it is tightly related to the thread-safety, asynchronous signal safety, locking, stability and etc.

Finally, we extended the logic with functions and local variables to address reentrancy,

and defined the Reentrancy Linear Type System. To make functions closer to the common implementations, we imitate argument passing and return values of the functions with global variables. We defined a classification for the functions, so they could be reentrant or non-reentrant. The designed Reentrancy Linear Type System ensures that non-reentrant functions are used at most once or not used at all in the environment with signals. We also raised some open-ended questions regarding variations of the logic rules due to availability of the Reentrancy Linear Type System.

11.1 Related Work

We are not aware of previous operational semantics and corresponding logic for signals, although Feng, Shao, Guo and Dong [32] presents a program logic for assembly language with interrupts, which are analogous to signals at the hardware level. In their semantics, blocking interrupts transfers ownership of parts of the heap among interrupt handlers, in the style of concurrent separation logic.

Hutton and Wright [46] study interruptions as asynchronous exceptions. By contrast, signals are a software alternative to hardware interrupts, where signal handlers could be addressed as asynchronous subroutine calls.

The use of binary predicates on program states in rely-guarantee [55] is a key technique from the concurrency literature that we have borrowed and adapted to signals. In recent years, rely-guarantee logic has received a boost due to its "marriage" with separation logic [98].

The tension between control flow and resource management has long been apparent in programming language design. In Java, the cleanup of resources when an exception leaves control of a block has been the cause of nests of `finally` clauses, and Java 7 adds a new `try-with-resources` block [76] for automatically closing resources like streams. The uninstalling of signal handlers in our semantics serves an analogous purpose.

The contrast between persistent and one-shot signal handlers is reminiscent of the

distinction between first-class and one-shot, linearly used continuations [8] and the resource usage in separation logic [86]. The way we have treated signal bindings in the big-step semantics borrows ideas from linear logic. Recall that we write

$$S; O \Vdash s_1, c \Downarrow s_2$$

for a judgement involving a persistent signal binding S and a one-shot signal binding O. As we have illustrated with the examples in Section 3.6, the signal binding S can be shared between two commands c_1 and c_2 in a sequential composition, whereas O has to be split into disjoint parts O_1 and O_2. This splitting prevents a one-shot signal handler from being re-used and makes it a linear resource just like the contexts in a linear logic. In fact, Dual Intuitionistic Linear Logic [6] has two zones Γ and Δ in the context, one which allows sharing and one which does not, as in the following rule that shares Δ and splits Γ:

$$\frac{\Gamma_1; \Delta \vdash M : A \multimap B \qquad \Gamma_2; \Delta \vdash N : A}{\Gamma_1, \Gamma_2; \Delta \vdash M\, N : B}$$

Recent work from the Flint group [37] presents a two-layer framework that is used to verify a concurrent thread management with a machine model including registers, instruction pointer, etc.

Huang et. al. [42] address interrupt-driven programs, but their main focus is on time semantics and time properties. Moreover, in their language, the interrupt handler does not modify the state of the interrupted programs.

Plotkin [79] models exception handlers with monads. Developing a general operational semantics for handling mechanisms and combining signals with other control structures in the same language are mentioned as further work. It was also noted that addressing the notion of "recursion" is important but requires extra work. Before addressing handling mechanism in general, author also started his work from a notion of exception handling.

A paper by [53] addresses a combination of exceptions and copyless messaging mech-

anism, where only pointers to messages are exchanged between two processes and the messages themselves are stored in an exchange heap. Authors assume exceptions are in general unpredictable, where in our work exceptions are triggered at the specific point of the code, but the signals' arrival is assumed to be unpredictable.

11.2 Directions for Future Work

In this section we outline what we believe to be most important and promising directions for future research. Work has already started on some of the them, but the details are out of scope of this thesis.

11.2.1 Separation Logic for Resource and Ownership

A natural extension of our program logic and operational semantics is the integration with separation logic [86] to address issues like deallocation of pointers. In recent years, rely-guarantee logic has received a boost due to its "marriage" with separation logic [98]. Our motivating examples from software security [27, 21] involve double free errors, which separation logic has successfully addressed in the absence of signals.

Resource separation between the handler and the main body of the program would greatly simplify the stability conditions that need to be checked. The ownership transfer described in the work Feng, Shao, Guo and Dong [32] for interrupts should also occur when signals are temporarily blocked.

11.2.2 Correctness of Signal Machine with Respect to Big-step Semantics

The formal connection between the big-step operational semantics and the signals abstract machine remains to be established. We conjecture that they are observationally equivalent and that this may be proved by way of a simulation relation.

11.2.3 Implementation

Signals have been part of the long evolution of Unix, and are correspondingly complex. To implement block-structured signal handling and integrate it with exceptions, the present signal mechanism may have to be revisited. The present implementations pose severe restrictions on programmers, for instance on using non-local control in a handler. Removing such implementation restrictions would enable natural programming idioms. In this thesis, we built on the operational semantics presented earlier for proving soundness of a Hoare logic for signals.

11.2.4 Signals in a Concurrent Setting

Many of the difficulties with "re-entrant" signal handlers are closely related to concurrent programming. Similarly, rely-guarantee logic is designed for shared-variable concurrency. Hence we would like to have a common logic for signals and parallel processes. One difficulty is that the assumptions about signals are so weak that it is not easy to see what pre- and postcondition we could specify for a kill command, since the signal may be handled much later or even ignored entirely. A more technical difficulty comes from the tension between the conjunction rule and trace (and also continuation) semantics. Defining a trace semantics (like the Aczel trace one for rely-guarantee [23]) and adding exceptions to it (in the double barrelled CPS style [96]) seems straightforward. However, proving the conjunction rule is known to be thorny in such a scenario. By contrast, a big-step operational semantics is technically convenient for block-structured control, which led us to prefer it to a trace semantics in the present thesis.

11.2.5 Reentrancy and Safety

There is no unified definition for the reentrancy and it is often confused with thread safety or async safety. Moreover, it is highly implementation dependant, what leads to a discrepancy in definitions. We are working towards addressing the reentrancy with

our logic and Reentrancy Linear Type System. The easiest way to achieve the safety is to block all signals (that prevents both safe and unsafe reentrancy), which could be addressed with a blocking rule in our logic. We aim to proof soundness of the updated logic with Reentrancy Linear Type System. Another potential branch of work is to address incomplete calls (aka slow library calls) with our logic, where interrupted and killed processes should be restarted until the desired outcome is reached.

11.2.6 Application to Software Security

The problem Zalewski's "Sending Signals for Fun and Profit" attack [27] is that is run twice, causing a double free and hence memory corruption. The signal constructs defined in the present thesis could prevent such vulnerabilities by installing the handler as a one-shot handler, or by jumping out of the handler using an exception.

Bibliography

[1] Mads Sig Ager. From natural semantics to abstract machines. In Sandro Etalle, editor, *Logic Based Program Synthesis and Transformation*, volume 3573 of *Lecture Notes in Computer Science*, pages 245–261. Springer Berlin Heidelberg, 2005.

[2] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science*, 342:04–28, 2005.

[3] Samy Al Bahra. Nonblocking algorithms and scalable multicore programming. *Commun. ACM*, 56(7):50–61, July 2013.

[4] Andrew W. Appel and Sandrine Blazy. Separation logic for small-step cminor. In *Proceedings of the 20th international conference on Theorem proving in higher order logics*, TPHOLs'07, pages 5–21, Berlin, Heidelberg, 2007. Springer-Verlag.

[5] Anindya Banerjee and David A. Naumann. A simple semantics and static analysis for stack inspection. *CoRR*, abs/1309.5144, 2013.

[6] Andrew Barber and Gordon Plotkin. Dual intuitionistic linear logic. *LFCS Report Series - Laboratory for Foundations of Computer Science*, 1996.

[7] Nick Benton and Andrew Kennedy. Exceptional syntax. *J. Funct. Program.*, 11:395–410, July 2001.

[8] Josh Berdine, Peter W. O'Hearn, Uday Reddy, and Hayo Thielecke. Linear continuation passing. *Higher-order and Symbolic Computation*, 15(2/3):181–208, 2002.

[9] Daniel Bovet and Marco Cesati. *Understanding the Linux Kernel, First Edition.* O'Reilly, 1 edition, 2000.

[10] Daniel Bovet and Marco Cesati. *Understanding the Linux Kernel, Second Edition.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2 edition, 2002.

[11] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel.* Oreilly & Associates Inc, 2005.

[12] J. Brandt. Synchronous models for embedded software. Master's thesis, Department of Computer Science, University of Kaiserslautern, July 2013. Habilitation.

[13] Stephen Brookes. A semantics for concurrent separation logic. In *Theoretical Computer Science*, pages 16–34. Springer, 2004.

[14] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science*, pages 366–378, Washington, DC, USA, 2007. IEEE Computer Society.

[15] Arthur Charguéraud. Pretty-Big-Step Semantics. In Matthias Felleisen and Philippa Gardner, editors, *22nd European Symposium on Programming (ESOP)*, Rome, Italie, March 2013. Springer.

[16] David Chase. Implementation of exception handling, Part I. *j-JCLT*, 5(4):229–240, jun 1994.

[17] Joey W. Coleman and Cliff B. Jones. A structural proof of the soundness of rely/guarantee rules. *J. Log. Comput.*, 17(4):807–841, 2007.

[18] Joey W. Coleman and Cliff B. Jones. A structural proof of the soundness of rely/guarantee rules (revised). Technical report, School of Computing Science, University of Newcastle, 2007.

[19] DWARF Standards Committee. DWARF Debugging Information Format Industry Review Draft, July 1993. Programming Languages SIG, Revision: 2.0.0 (July 27, 1993).

[20] Richard C. H. Connor, Alfred L. Brown, Raymund Carrick, Alan Dearle, and Ronald Morrison. The persistent abstract machine. In John Rosenberg and David Koch, editors, *POS*, Workshops in Computing, pages 353–366. Springer, 1989.

[21] Horia V Corcalciuc. A taxonomy of time and state attacks. In *International Workshop on Secure Software Engineering*, 2012.

[22] Christophe de Dinechin. C++ exception handling for IA-64. In *Proceedings of the 1st conference on Industrial Experiences with Systems Software - Volume 1*, pages 8–8, Berkeley, CA, USA, 2000. USENIX Association.

[23] Willem-Paul de Roever, Frank de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Number 54 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK, November 2001.

[24] Mathieu Desnoyers. Proving the correctness of nonblocking data structures. *Queue*, 11(5):30:30–30:43, May 2013.

[25] S. Diehl, P. H. Hartel, and P. Sestoft. Abstract machines for programming language implementation. *Future Generation Computer Systems*, 16(7):739–51, May 2000.

[26] Mike Dodds, Suresh Jagannathan, and Matthew J. Parkinson. Modular reasoning for deterministic parallelism. *SIGPLAN Not.*, 46:259–270, January 2011.

[27] Mark Dowd, John McDonald, and Justin Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley Professional, 2006.

[28] Itanium C++ ABI: Exception Handling, March 2001. (Revision: 1.22).

[29] Manuel Fähndrich, Diego Garbervetsky, and Wolfram Schulte. A static analysis to detect re-entrancy in object oriented programs. *Journal of Object Technology*, 7(5):5–23, 2008.

[30] Xinyu Feng, Zhong Shao, Yuan Dong, and Yu Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 170–182, New York, NY, USA, 2008. ACM.

[31] Xinyu Feng, Zhong Shao, Yu Guo, and Yuan Dong. Combining domain-specific and foundational logics to verify complete software systems. In *Proceedings of the 2nd international conference on Verified Software: Theories, Tools, Experiments*, VSTTE '08, pages 54–69, Berlin, Heidelberg, 2008. Springer-Verlag.

[32] Xinyu Feng, Zhong Shao, Yu Guo, and Yuan Dong. Certifying low-level programs with hardware interrupts and preemptive threads. *J. Autom. Reason.*, 42:301–347, April 2009.

[33] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The Spirit of Ghost Code. http://hal.inria.fr/hal-00873187/PDF/main.pdf.

[34] Inc Free Software Foundation. The GNU C Library Reference Manual. `http://www.gnu.org/software/libc/manual/html_node/Blocking-for-Handler.html`, January 2013. for Version 2.17 of the GNU C Library. Accessed: 2013-03-19.

[35] Colin S. Gordon, Michael D. Ernst, and Dan Grossman. Rely-guarantee references for refinement types over aliased mutable data. In *PLDI 2013, Proceedings of the ACM SIGPLAN 2013 Conference on Programming Language Design and Implementation*, Seattle, WA, USA, June 17–19, 2013.

[36] Alexey Gotsman, Josh Berdine, and Byron Cook. Precision and the conjunction rule in concurrent separation logic. *27th Conference on the Mathematical Foundations of Programming Semantics, MFPS 27, Pittsburgh, PA, USA, May 25–28, 2011. Electr. Notes Theor. Comput. Sci.*, 2011.

[37] Yu Guo, Xinyu Feng, Zhong Shao, and Peizhi Shi. Modular verification of concurrent thread management. In *Proc. 10th Asian Symposium on Programming Languages and Systems (APLAS'12), Kyoto, Japan*, volume 7705 of *Lecture Notes in Computer Science*. Springer-Verlag, 2012.

[38] Kyong-Hoon Kim Guy Martin Tchamgoue and Yong-Kee Jun. Testing and debugging concurrency bugs in event-driven programs. *International Journal of Advanced Science and Technology*, 40:55–68, 2012.

[39] Christian Haack, Marieke Huisman, and Clément Hurlin. Reasoning about java's reentrant locks. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 171–187, Berlin, Heidelberg, 2008. Springer-Verlag.

[40] Martin Hofmann and Mariela Pavlova. Elimination of ghost variables in program logics. In *Proceedings of the 3rd conference on Trustworthy global computing*, TGC'07, pages 1–20, Berlin, Heidelberg, 2008. Springer-Verlag.

[41] Furio Honsell. 25 years of formal proof cultures: some problems, some philosophy, bright future. In *Proceedings of the Eighth ACM SIGPLAN international workshop on Logical frameworks & meta-languages: theory & practice*, LFMTP '13, pages 37–42, New York, NY, USA, 2013. ACM.

[42] Yanhong Huang, Yongxin Zhao, Jianqi Shi, Huibiao Zhu, and Shengchao Qin. Investigating time properties of interrupt-driven programs. In *SBMF*, pages 131–146, 2012.

[43] Marieke Huisman and Bart Jacobs. Java program verification via a hoare logic with abrupt termination. In *Proceedings of the Third Internationsl Conference on Fundamental Approaches to Software Engineering: Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000*, FASE '00, pages 284–303, London, UK, 2000. Springer-Verlag.

[44] Graham Hutton and Joel Wright. Compiling exceptions correctly. In Dexter Kozen and Carron Shankland, editors, *MPC*, volume 3125 of *Lecture Notes in Computer Science*, pages 211–227. Springer, 2004.

[45] Graham Hutton and Joel Wright. Calculating an Exceptional Machine. In Hans-Wolfgang Loidl, editor, *Trends in Functional Programming volume 5*. Intellect, feb 2006. Selected papers from the Fifth Symposium on Trends in Functional Programming, Munich, November 2004.

[46] Graham Hutton and Joel Wright. What is the meaning of these constant interruptions? *J. Funct. Program.*, 17(6):777–792, 2007.

[47] Hayes IJ, Cliff B. Jones, and Colvin RJ. Reasoning about concurrent programs: Refining rely-guarantee thinking. Technical report, Newcastle University, September 2013. No. CS-TR-1395.

[48] Apple Inc. OS X Glossary. `http://developer.apple.com/library/mac/#documentation/General/Conceptual/SLGlobalGlossary/Glossary/Glossary.html`, July 2010. Accessed: 2013-03-14.

[49] Apple Inc. Concurrency Programming Guide - Glossary. `http://developer.apple.com/library/mac/#documentation/General/Conceptual/ConcurrencyProgrammingGuide/Glossary/Glossary.html`, December 2012. Accessed: 2013-03-14.

[50] ISO/IEC. 14882:1999 Programming languages - C++, 1999.

[51] ISO/IEC. 14882:2011 Information technology - Programming languages - C++, 2011.

[52] Bart Jacobs and Erik Poll. Java program verification at nijmegen: Developments and perspective. In *Nijmegen Institute of Computing and Information Sciences*, pages 134–153. Springer, 2003.

[53] Svetlana Jakšić and Luca Padovani. Exception handling for copyless messaging. In *Proceedings of the 14th symposium on Principles and practice of declarative programming*, PPDP '12, pages 151–162, New York, NY, USA, 2012. ACM.

[54] Dipak Jha. Use reentrant functions for safer signal handling. `http://www.ibm.com/developerworks/library/l-reent/index.html`, January 2005. IBM. Accessed: 2013-03-19.

[55] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5:596–619, October 1983.

[56] Michael Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press, San Francisco, CA, USA, 1st edition, 2010.

[57] Eric Koskinen and Maurice Herlihy. Checkpoints and continuations instead of nested transactions. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 160–168, New York, NY, USA, 2008. ACM.

[58] Jaroslaw D. M. Kusmierek and Viviana Bono. Big-step operational semantics revisited. *Fundam. Inform.*, 103(1-4):137–172, 2010.

[59] Leslie Lamport and Fred B. Schneider. The "hoare logic" of csp, and all that. *ACM Trans. Program. Lang. Syst.*, 6(2):281–296, April 1984.

[60] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207(2):284–304, February 2009.

[61] Bil Lewis and Daniel J. Berg. *Threads Primer: A Guide to Multithreaded Programming*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1995.

[62] Richard J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[63] Lutz Schröder and Till Mossakowski. Generic exception handling and the java monad. In *Algebraic Methodology and Software Technology, Lecture Notes in Computer Science*, volume 3116, pages 443–459. Springer, 2004.

[64] Simon Marlow, Simon L. Peyton Jones, Andrew Moran, and John H. Reppy. Asynchronous Exceptions in Haskell. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, pages 274–285, 2001.

[65] Martin Nordio and Cristiano Calcagno and Peter Müller and Bertrand Meyer. A sound and complete program logic for Eiffel. In M. Oriol and B. Meyer, editors, *TOOLS-EUROPE*, volume 33 of *Lecture Notes in Business and Information Processing*, pages 195–214, 2009.

[66] Martin Nordio and Cristiano Calcagno and Peter Müller and Bertrand Meyer. Soundness and completeness of a program logic for Eiffel. Technical Report 617, ETH Zurich, 2009.

[67] Maged M. Michael. The balancing act of choosing nonblocking features. *Queue*, 11(7):50:50–50:61, July 2013.

[68] Ana Milanova and Wei Huang. Static object race detection. In *APLAS*, pages 255–271, 2011.

[69] Robin Milner, Mads Tofte, Robert Harper, and David Macqueen. *The Definition of Standard ML - Revised*. The MIT Press, rev sub edition, May 1997.

[70] Eugenio Moggi. Computational lambda calculus and monads. In *Proceedings, Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, 1989.

[71] Manuel Montenegro, Ricardo Peña-Marí, and Clara Segura. A resource-aware semantics and abstract machine for a functional language with explicit deallocation. *Electr. Notes Theor. Comput. Sci.*, 246:167–182, 2009.

[72] Keiko Nakata and Tarmo Uustalu. A hoare logic for the coinductive trace-based big-step semantics of while. In Andrew D. Gordon, editor, *ESOP*, volume 6012 of *Lecture Notes in Computer Science*, pages 488–506. Springer, 2010.

[73] Aleksandar Nanevski, Paul Govereau, and Greg Morrisett. Towards type-theoretic semantics for transactional concurrency. In *Proceedings of the 4th international workshop on Types in language design and implementation*, TLDI '09, pages 79–90, New York, NY, USA, 2009. ACM.

[74] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in hoare type theory. In *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, ICFP '06, pages 62–73, New York, NY, USA, 2006. ACM.

[75] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Proceedings of the 15th International Workshop on Computer Science Logic*, CSL '01, pages 1–19, London, UK, 2001. Springer-Verlag.

[76] Oracle. The try-with-resources Statement, 2011.

[77] Matthew J. Parkinson, Richard Bornat, and Cristiano Calcagno. Variables as resource in hoare logics. In *LICS*, pages 137–146. IEEE Computer Society, 2006.

[78] Greg Pettyjohn, John Clements, Joe Marshall, Shriram Krishnamurthi, and Matthias Felleisen. Continuations from generalized stack inspection. *SIGPLAN Not.*, 40(9):216–227, 2005.

[79] Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In Giuseppe Castagna, editor, *ESOP*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2009.

[80] Qt Project and Digia. Qt 4.8 Documentation - Reentrancy and Thread-Safety. `http://qt-project.org/doc/qt-4.8/threads-reentrancy.html`. Accessed: 2013-03-19.

[81] John Regehr. *Safe and Structured Use of Interrupts in Real-Time and Embedded Software*, chapter 16, pages 1–12. Chapman and Hall/CRC, 2012/06/17 2007.

[82] Yann Régis-Gianas and François Pottier. A hoare logic for call-by-value functional programs. In *Proceedings of the 9th international conference on Mathematics of Program Construction*, MPC '08, pages 305–335, Berlin, Heidelberg, 2008. Springer-Verlag.

[83] Bernhard Reus and Thomas Streicher. About Hoare logic for higher-order store. In *ICALP'05*, volume 3580 of *LNCS*, pages 1337–1348. Springer, 2005.

[84] John C. Reynolds. *The craft of programming*. Prentice Hall International series in computer science. Prentice Hall, 1981.

[85] John C. Reynolds. *Theories of programming languages*. Cambridge University Press, 1998.

[86] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS)*, pages 55–74. IEEE, 2002.

[87] Kay Robbins and Steve Robbins. *UNIX Systems Programming: Communication, Concurrency and Threads (2nd Edition)*. Prentice Hall PTR, June 2003.

[88] Sandra Loosemore and Richard M. Stallman and Roland McGrath and Andrew Oram and Ulrich Drepper. The GNU C Library Reference Manual, nov 2007. Edition 0.12, last updated 2007-10-27, for version 2.8.

[89] Rui Shi, Dengping Zhu, and Hongwei Xi. A modality for safe resource sharing and code reentrancy. In *Proceedings of the 7th International colloquium conference on Theoretical aspects of computing*, ICTAC'10, pages 382–396, Berlin, Heidelberg, 2010. Springer-Verlag.

[90] William Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall Press, Upper Saddle River, NJ, USA, 6th edition, 2008.

[91] Richard W. Stevens and Stephen A. Rago. *Advanced Programming in the UNIX(R) Environment (2nd Edition)*. Addison-Wesley Professional, 2005.

[92] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling fulljumps. *Higher Order Symbol. Comput.*, 13(1-2):135–152, 2000.

[93] Maxim Strygin and Hayo Thielecke. Operational semantics for signal handling. In *SOS 2012*, volume 89 of *EPTCS*, pages 149–163, 2012.

[94] Éric Tanter. Controlling aspect reentrancy. *J. UCS*, 14(21):3498–3516, 2008.

[95] Hayo Thielecke. Continuations, functions and jumps. *SIGACT News*, 30(2):33–42, 1999.

[96] Hayo Thielecke. Comparing control constructs by double-barrelled CPS. *Higher-order and Symbolic Computation*, 15(2/3):141–160, 2002.

[97] Viktor Vafeiadis. Modular fine-grained concurrency verification. Technical Report UCAM-CL-TR-726, University of Cambridge, Computer Laboratory, July 2008.

[98] Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *In 18th CONCUR*, pages 256–271. Springer, 2007.

[99] Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. Technical Report UCAM-CL-TR-687, University of Cambridge, Computer Laboratory, June 2007.

[100] David von Oheimb. Hoare logic for Java in Isabelle/HOL. In *In Isabelle/HOL. Concurrency and Computation: Practice and Experience 13*, pages 1173–1214, 2001.

[101] Jérôme Vouillon. Order Theory for Big-Step Semantics. `http://hal.archives-ouvertes.fr/hal-00782145`, 2011.

[102] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. Compcerttso: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22:1–22:50, June 2013.

[103] Philip Wadler. Monads and composable continuations. *Lisp Symb. Comput.*, 7(1):39–56, 1994.

[104] Jan Wloka, Manu Sridharan, and Frank Tip. Refactoring for reentrancy. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 173–182, New York, NY, USA, 2009. ACM.

[105] Qiwen Xu, Willem P. de Roever, and Jifeng He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Asp. Comput.*, 9(2):149–174, 1997.

[106] Hongseok Yang and Peter W. O'Hearn. A semantic basis for local reasoning. In *Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures*, FoSSaCS '02, pages 402–416, London, UK, UK, 2002. Springer-Verlag.

[107] Jaeheon Yi and Cormac Flanagan. Effects for cooperable and serializable threads. In *TLDI '10: Proceedings of the 5th ACM SIGPLAN workshop on Types in language design and implementation*, pages 3–14, New York, NY, USA, 2010. ACM.