A UNIFYING THEORY OF MULTI-EXIT PROGRAMS

A UNIFYING THEORY OF MULTI-EXIT PROGRAMS

By

TIAN ZHANG, M.Eng., B.Eng.

A Thesis

Submitted to the Department of Computing and Software

and the School of Graduate Studies

of McMaster University

in Partial Fulfilment of the Requirements

for the Degree of

Doctor of Philosophy

Doctor of Philosophy (2013) McMaster University

(Computing and Software) Hamilton, Ontario, Canada

TITLE: A Unifying Theory of Multi-Exit Programs

AUTHOR: Tian Zhang

M.Eng. (Computer Software and Theory)

Central South University, Changsha, China

B.Eng. (Computer Science and Technology)

Central South University, Changsha, China

SUPERVISOR: Dr. Emil Sekerinski

NUMBER OF PAGES: x, 121

To my beloved family

Abstract

Programs have multiple exits in the presence of certain control structures, *e.g.*, exception handling and coroutines. These control structures offer flexible manipulations of control flow. However, their formalizations are overall specialized, which hinders reasoning about combinations of these control structures.

In this thesis, we propose a unifying theory of multi-exit programs. We mechanically formalize the semantics of multi-exit programs as indexed predicate transformers, a generalization of predicate transformers by taking the tuple of postconditions on all exits as the parameter. We explore their algebraic properties and verification rules, then define a normal form for monotonic and for conjunctive indexed predicate transformers. We also propose a new notion of fail-safe correctness to model the category of programs that always maintain certain safe conditions when they fail, and a new notion of fail-safe refinement to express partiality in software development.

For the indexed predicate transformer formalization, we illustrate its applications in three models of multi-exit control structures: the termination model of exception handling, the retry model of exception handling, and a coroutine model. Additionally, for the fail-safe correctness notion and the fail-safe refinement notion, we illustrate their applications in the termination model. Six design patterns in the termination model and one in the retry model are studied. All the verification rules and design patterns in the thesis have been formally checked by a verification tool.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Dr. Emil Sekerinski. Without his firm support and patient guidance, this thesis could not have been completed. I am also forever indebted to his generosity of funding me for Summer School Marktoberdorf 2011 as well as international conferences, which offered me great chances of discussing with researchers from all over the world.

I would also like to express my great appreciation to my supervisory committee members. In particular, my numerous enlightening discussion with Dr. William M. Farmer deepened my understanding of the underlying logics in formal methods. His detailed remarks of this thesis were also tremendously helpful. Dr. Ryszard Janicki's critical suggestion on the choice of my thesis topic, as well as his insightful comments on this thesis, are of great influence to my research.

To my external examiner Dr. Micheal Winter, I am tremendously grateful for his positive review on this thesis and his helpful indication of the parts that required further explanation. My special thanks to my fellow graduate students Ronald Eden Burton, Bojan Nokovic, and Shucai Yao, for their constructive feedback in our group meetings.

Last but not least, I want to thank my father Qiwei Zhang and my mother Hong Xiang, for their unconditional love and never-ending patience.

Contents

List of Figures

Declaration of Academic Achievement

Sekerinski, E. and T. Zhang (2009, September). Modelling finitary fairness in Event-B (extended abstract). In J.-R. Abrial, M. Butler, R. Joshi, E. Troubitsyna, and J. C. P. Woodcock (Eds.), *Dagstuhl Seminar on Refinement Based Methods for the Construction of Dependable Systems*, 5 pages, pp. 152–156. Leibniz-Zentrum fuer Informatik, Germany.

Sekerinski, E. and T. Zhang (2011). A new notion of partial correctness for exception handling. In B. Bonakdarpour and T. Maibaum (Eds.), *2nd International Workshop on Logical Aspects of Fault-Tolerance*, pp. 116–132.

Sekerinski, E. and T. Zhang (2012). Verification rules for exception handling in Eiffel. In R. Gheyi and D. Naumann (Eds.), *Formal Methods: Foundations and Applications*, Volume 7498 of *Lecture Notes in Computer Science*, pp. 179–193. Springer Berlin / Heidelberg. *(Awarded the best paper of Brazilian Symposium on Formal Methods 2012.)*

Sekerinski, E. and T. Zhang (2013, June). On a new notion of partial refinement. In J. Derrick, E. Boiten, and S. Reeves (Eds.), *Proceedings 16th International Refinement Workshop*, Volume 115 of *Electronic Proceedings in*

Theoretical Computer Science, pp. 1–14. Open Publishing Association.

Sekerinski, E. and T. Zhang (2013). Finitary fairness in action systems. In Z. Liu, J. Woodcock, and H. Zhu (Eds.), *Theoretical Aspects of Computing ICTAC 2013*, Volume 8049 of *Lecture Notes in Computer Science*, pp. 319–336. Springer Berlin Heidelberg.

Chapter 1

Introduction

Formal correctness has been originally studied for single-entry, single-exit imperative programs [Hoare, 1969; Tennent, 1976; Plotkin, 1981]. Certain control structures introduce new challenges for correctness reasoning since they violate the single-entry, single-exit abstraction of program blocks. Two examples are exception handling and coroutines[1]:

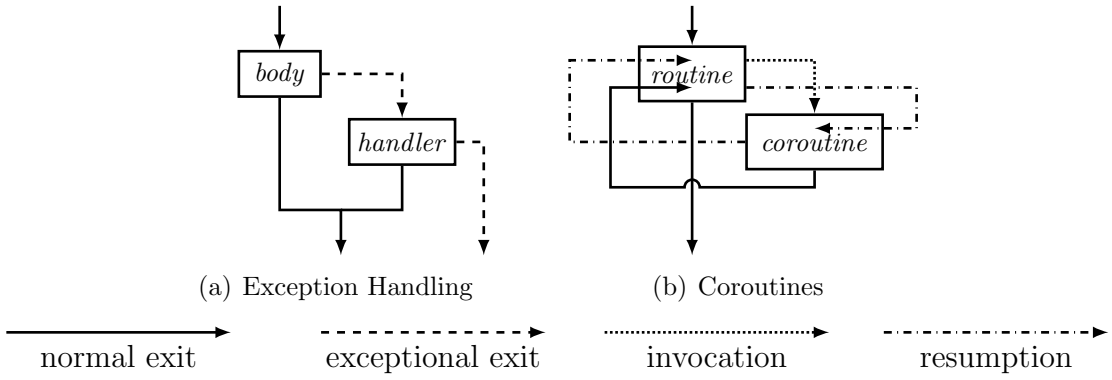


Figure 1.1: Flowcharts of Exception Handling and Coroutines

An *exception handling* mechanism is a dedicated control structure for detecting

[1]The two flowcharts do not represent all implementations of exception handling and coroutines.

and recovering errors [Garcia et al., 2001]. A *coroutine* generalizes a subroutine by allowing multiple entry points for suspending and resuming execution at certain locations [Conway, 1963]. Both of them allow redirecting control flow away from normal termination of a program block (two models are shown in Figure 1.1), which makes them multi-exit control structures. In the above flowchart of coroutines, the arrows going inside the program blocks indicate that the execution continues at the middle of the program block instead of at the beginning.

The implementation of control structures employs control flow redirection. Assembly languages offer *branch instructions* to redirect control flow [AMD, 2012; Intel, 2013]. Some high-level programming languages, *e.g.*, Fortran [ANSI, 1966], C [ISO, 1999], and C++ [ISO, 1998], provide goto as an unconditional branch statement. The structured programming theorem points out that every Turing machine can be reduced to a program in a language that only employs composition and iteration as formation rules [Böhm and Jacopini, 1966], implying that goto is dispensable with respect to theoretical expressivity. Dijkstra [1968] suggests removing goto statement from "higher-level" programming languages since it complicates analysis and verification of programs. This methodology is followed by languages like Java [Gosling et al., 2013] and Python [Python Software Foundation, 2013]. Knuth [1974] advocates another viewpoint: employing structured programming to reduce the reliance on goto, but still reserving this statement for efficiency. C [ISO, 2011] and C++ [ISO, 2012] follow this methodology. Fortran adds restriction on goto statements in later standards [ANSI, 1997]. Control structures are essentially usage of goto statements under various restrictions.

Formal reasoning about correctness requires *formal methods*, *i.e.*, mathematically based languages, techniques, and tools for specifying and verifying systems. Formal methods can dramatically enhance code quality and reduce (but not eliminate) the

reliance on testing [Clarke and Wing, 1996]. The *stepwise refinement* methodology constructs programs in a top-down style, starting from abstract specifications and working stepwise to concrete implementations [Back and von Wright, 1998]. Every refinement step reflects some design decisions towards the final solution [Wirth, 1971]. Formalizations and refinement calculi of single-exit programs have been studied [Hoare, 1969; Dijkstra, 1975; Back and von Wright, 1998]. However, current formalizations of multi-exit programs either are only applicable to a restricted range of languages and control structures or do not support stepwise refinement.

In this thesis, we propose a unifying theory of multi-exit programs, which not only gives a more abstract view on the manipulation of control flow, but also allows all multi-exit controls structures to be combined and reasoned about in a single framework. We restrict the work to imperative programs.

The thesis is organized according to the following structure.

- Chapter 2 summarizes related work;
- Chapter 3 gives basic formal definitions of indexed predicate transformers, then proposes a refinement calculus and verification rules for them;
- Chapter 4 studies the rules for recursion and iteration;
- Chapter 5 defines a normal form of monotonic indexed predicate transformers and a normal form of conjunctive ones;
- Chapter 6 first formalizes the termination model of exception handling, then proposes the new notions of fail-safe correctness and fail-safe refinement;
- Chapter 7 studies six formal design patterns which illustrate the usage of fail-safe correctness and fail-safe refinement;

- Chapters 8 formalizes the retry model of exception handling, and applies its verification rules to a design pattern;
- Chapters 9 defines the semantics of a coroutine language, as well as its verification rules;
- Chapter 10 concludes the thesis with a discussion.
- Appendix A gives an overview of the mechanical formalization.

All theorems and design patterns have been checked in theorem prover Isabelle [Nipkow and Paulson, 1992], so we use ✓ to mark them, and allow ourselves to give only proof sketches or leave out proofs completely[2].

[2]The complete Isabelle formalization is available at `http://www.cas.mcmaster.ca/~zhangt26/thesis/`

Chapter 2

Related Work

2.1 Towards Error-Free Programs

Software errors have caused loss of money, *e.g.*, Ariane 5 [Dowson, 1997], even precious lives, *e.g.*, Therac-25 [Leveson and Turner, 1993]. In 2002, the US Department of Commerce estimated that avoidable software errors cost the US economy 20 to 60 billion dollars each year [Reed et al., 2007].

One method to eliminate software errors is *software testing*, which executes a program to find errors. It typically accounts for approximately 50% of the elapsed time and more than 50% of the total cost [Myers, 2004]. Moreover, testing helps to find errors but can never guarantee the correctness of programs [Dijkstra, 1970]. An example is that the binary search implementation of arrays in Java standard class libraries was erroneous because of a potential integer overflow in computing the average of two integers. The error has remained undetected for about nine years because small sets of test cases are unlikely to include the large scale of inputs that is necessary to produce the overflow [Bloch, 2006].

Formal methods take a more mathematically based route towards error-free programs: they employ *specification* for describing a system with its desired properties, and *verification* for checking the faithfulness of an implementation *w.r.t.* its specification. Using them does not *a priori* guarantee correctness, but can reveal inconsistencies, ambiguities, and incompleteness that might otherwise go undetected [Clarke and Wing, 1996], so the testing workload required would be much less in their presence. Formal methods are essential in, but not restricted to, safety-critic systems, *e.g.*, railway and air traffic management systems [Bowen and Stavridou, 1993; Lecomte et al., 2007], or software that requires high reliability, *e.g.*, Microsoft Hyper-V hypervisor [Leinenbach and Santen, 2009].

Two well-established approaches to verification are *model checking* and *theorem proving*. Model checking builds a finite model of a system and performs an exhaustive state space search to check that the desired properties hold in that model. Theorem proving expresses both the system and its desired properties in some mathematical logic, which is given by defining a set of axioms and inference rules, then finds a proof for the properties from the axioms of the system [Clarke and Wing, 1996].

Each method has its own advantages: model checking is completely automatic and can provide counterexamples; theorem proving can directly deal with infinite state spaces and scales more easily [Clarke and Wing, 1996]. In this thesis, we follow the methodology of theorem proving, in particular for formalizing multi-exit programs, and exploring algebraic properties of the formalization.

Formalizations of program semantics fall into three major categories: *operational semantics*, *denotational semantics*, and *axiomatic semantics* [Schmidt, 1986]. Operational semantics [Plotkin, 1981] is defined by specifying an interpreter. Denotational semantics [Tennent, 1976] is defined through mapping a program to its meaning,

a.k.a. denotation. Axiomatic semantics [Hoare, 1969] is defined by specifying properties about language constructs, instead of explicitly formalizing the meanings of programs, then axioms and inference rules are employed for reasoning about these properties. The three methods have different areas of application; here we focus on axiomatic semantics in pursuit of a language-independent formalization of multi-exit programs.

2.2 Formalizations of Single-Exit Programs

Hoare logic uses $P\{Q\}R$ to represent "if assertion P is true before initiation of program Q, then assertion R will be true on its completion", but termination needs to be proved separately [Hoare, 1969]. Dijkstra [1975] extends Hoare logic by taking termination into consider: $wp(S, R)$ is used to denote the *weakest precondition* for the initial state of the system such that the execution of S guarantees proper termination in a final state that satisfies the postcondition R. The function *wp* is called a "*predicate transformer*" because it takes a postcondition as the parameter and calculates the weakest precondition.

Predicate transformer semantics is integral to the B-method [Abrial et al., 1991] and Event-B [Métayer et al., 2005], which are formal methods that support specification, design, proof and code generation. B-method is for sequential programs and Event-B is for concurrent programs. Cavalcanti and Woodcock [1998] proposes predicate transformer semantics for the Z notation. Lamport [1980, 1990] extends Hoare Logic and predicate transformers with concurrency.

2.3 Formalizations of Multi-Exit Programs

An *exit* is a possible way of transferring control out of a program block, corresponding to outgoing arrows of statements in flowcharts, as in Figure 1.1. We consider programs with more than one exit to be *multi-exit*, *e.g.*, programs written in languages with exception handling and coroutines. The formalizations mentioned in Section 2.2 are only for *single-exit* programs, thus insufficient to reason about multi-exit control structures.

Exceptions are rare circumstances that prevent a program from providing its specified standard service [Cristian, 1989], and exception handling deals with these cases in a structured way in which the original design remains visible [Sekerinski and Zhang, 2011]. Various programming languages provide structured exception handling constructs, *e.g.*, try-catch in C++ [ISO, 2012], try-catch-finally in Java [Gosling et al., 2013], and do-rescue-end in Eiffel [Meyer, 1988]. Consequently, programs in these languages have a *normal exit* and an *exceptional exit*. On normal termination without exception, the execution continues on the normal exit. When an exception is encountered, the control of execution would be passed to the proper handler on the exceptional exit [Cristian, 1982].

The models for exception handling vary in languages. C++ and Java use the *termination model*, *i.e.*, a try-catch(-finally) section terminates when its exception handler terminates. The language Eiffel uses the *retry model*, *i.e.*, the exception handler can choose to retry the task [Meyer, 1987], so there is a *retry exit* that immediately directs the execution to the beginning of the corresponding do section. The language Mesa uses the *resumption model*, *i.e.*, the exception handler can choose to resume the task by directing the execution to the parts following where the exception was encountered and raised [Mitchell et al., 1979].

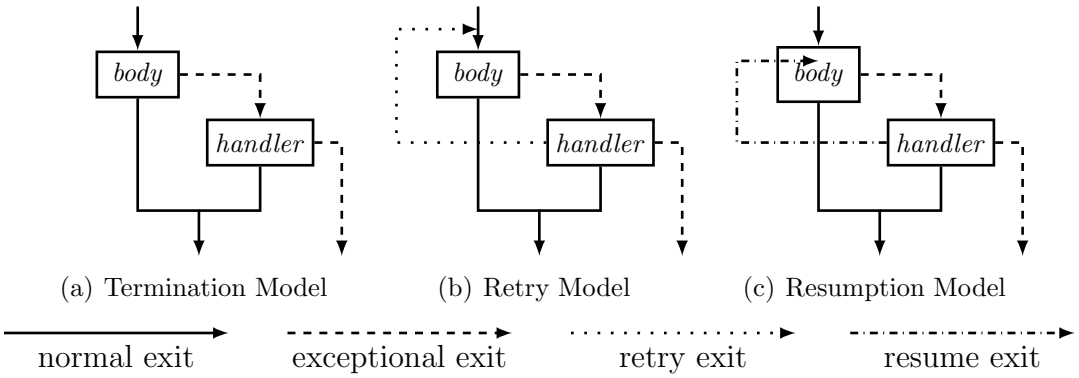


Figure 2.1: Exception Handling Mechanisms

Cristian [1984] specifies a statement with one entry, one normal exit and additional exceptional exits with one precondition and one postconditions for each exit, which is also advocated by Liskov and Guttag [2000]. It is followed by languages like Ada [Luckham and Polak, 1980], Spec# [Barnett et al., 2005], and JML [Leavens et al., 2006]. A different approach is advocated by Meyer [1997]: each method is specified by one precondition and one postcondition only. The normal exit is taken if the desired postcondition is established, and the exceptional exit is taken otherwise.

A coroutine communicates with adjacent peers as if they were input or output subroutines [Conway, 1963], which makes it well-suited for implementing cooperative tasks, iterators, and pipes. Coroutines are natively supported in Ruby [Flanagan and Matsumoto, 2008], Lua [Ierusalimschy et al., 2005], Python [van Rossum and Eby, 2011; Ewing, 2012], and Go [Google, 2009]. However, the implementations differ in semantics, the details of which is studied in [Moura and Ierusalimschy, 2009] and will be summarized in Chapter 9. A coroutine can be suspended and resumed. When a coroutine is suspended, it yields the control to resume the execution of the caller on the *resume exit*.

Clint [1973] proposes a rule for proving the correctness of coroutines, which requires

one condition to hold every time the coroutine is entered, and one condition to hold every time the coroutine exits. The two conditions serve as the bridges, linking up the alternating executions of the coroutine and its caller, in a similar way of how a loop invariant connects iterations of the loop. Clarke [1980] shows that the history variable in the histogram example in [Clint, 1973] is unnecessary. Their rules do not deal with termination.

2.4 Stepwise Refinement

The *refinement* relationship implies substitutability: if program S is refined by program T, then substituting T for S would always preserve correctness [Back and von Wright, 1998]. In other words, T maintains or reduces the nondeterminism in S by providing the same or more implementation details. Thus, due to the transitivity of the refinement relationship, stepwise refinement allows incremental development from abstract specifications to concrete implementations, with the correctness preserved through each step. It corresponds to the software development process of decomposing instructions of the given program into more detailed ones; the process terminates when all instructions are expressed in terms of an underlying computer or programming language [Wirth, 1971].

[Cristian, 1982] defines multi-exit programs by a set of predicate transformers, one for each exit. As pointed out by King and Morgan [1995], this approach disallows nondeterminism, an essential part of languages for specification and design. King and Morgan [1995] trace the use of *multi-argument predicate transformer*, as a formalization of multi-exit programs, to [Back and Karttunen, 1983]. However, according to [King and Morgan, 1995], Back and Karttunen [1983] do not deal with refinement and recursion. Double-exit predicate transformers are studied by King and Morgan

[1995], but are not extended to predicate transformers of arbitrary exits.

2.5 Formalizations in Higher-Order Logic

We propose indexed predicate transformer as a model for multi-exit programs, and formalize it in higher-order logic HOL. Dijkstra's guarded command language and refinement concepts have been formalized in HOL [Harrison, 1998; Back and Wright, 1990]. The semantics of C and Boogie (an intermediate verification language) have also been formalized in HOL [Cohen et al., 2009; Böhme et al., 2008]. Proof assistants HOL Light [Harrison, 2011] and Isabelle [Nipkow et al., 2002] support theorem proving in HOL. The semantics of Java and OCL have been formalized in Isabelle/HOL [von Oheimb, 2001; Brucker and Wolff, 2002].

2.6 Overview

Current formalizations of exception handling [Luckham and Polak, 1980; Barnett et al., 2005; Leavens et al., 2006; Tschannen et al., 2011] and coroutines [Clint, 1973; Clarke, 1980] are restricted to their own programming languages or control structures. Multi-argument predicate transformer [Back and Karttunen, 1983] is a higher-level abstraction, but verification rules about refinement and recursion are not provided. We propose a unifying theory based on indexed predicate transformers, providing a comprehensive solution for formal reasoning about multi-exit programs.

Chapter 3

Basic Definitions

3.1 Lattices and Monoids

In this section, we give the definitions of a lattice and a monoid, following [Birkhoff, 1967] and [Howie, 1995].

Definition 3.1. *(Section 1.1 of [Birkhoff, 1967]) A* partially-ordered set *(poset) P is a set in which a reflexive, antisymmetric, and transitive binary relation $\leq$ is defined,* i.e.*, for all $x, y, z \in P$:*

$$x \leq x \qquad \text{(reflexivity)}$$

$$x \leq y \wedge y \leq x \Rightarrow x = y \qquad \text{(antisymmetry)}$$

$$x \leq y \wedge y \leq z \Rightarrow x \leq z \qquad \text{(transitivity)}$$

Definition 3.2. *(Section 1.4 of [Birkhoff, 1967]) A* lattice *is a poset in which any two elements have a greatest lower bound (the* meet*, denoted by $\sqcap$) and a least upper*

bound (the join*, denoted by* $\sqcup$*), both of which are in* L*,* i.e.*, for all* $x, y, z \in L$*:*

$$x \sqcap y \in L \quad x \sqcup y \in L \qquad \text{(closure)}$$

$$x \sqcap y \leq x \quad x \sqcap y \leq y \quad z \leq x \wedge z \leq y \Rightarrow z \leq x \sqcap y \qquad \text{(greatest lower bound)}$$

$$x \leq x \sqcup y \quad y \leq x \sqcup y \quad x \leq z \wedge y \leq z \Rightarrow x \sqcup y \leq z \qquad \text{(least upper bound)}$$

A lattice L *is* complete *if and only if every subset of it has a join and a meet in* L*. A lattice* L *is* distributive *if and only if for all* $x, y, z \in L$*,* $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$*.*

Any nonvoid complete lattice L contains a *bottom element* $\bot$ that satisfies $\forall\, x \in L.\, \bot \leq x$ and a *top element* $\top$ that satisfies $\forall\, x \in L.\, x \leq \top$ [Birkhoff, 1967].

Definition 3.3. *(Section 1.1 of [Howie, 1995] A* monoid *is a set in which an associative binary operation "*$\cdot$*" is defined and an identity element* e *(the* unit*) exists,* i.e.*, for all* $x, y, z \in M$*:*

$$x \cdot y \in M \qquad \text{(closure)}$$

$$(x \cdot y) \cdot z = x \cdot (y \cdot z) \qquad \text{(associativity)}$$

$$x \cdot e = e \cdot x = x \qquad \text{(unit)}$$

For example, all integers form a monoid structure with $+$ *as the operation and* 0 *as the unit. Another monoid is the integers with operation* $\times$ *and unit* 1*.*

If $\forall\, x.\, e \cdot x = e$ we call e a *left zero*, and if $\forall\, x.\, x \cdot e = e$ we call e a *right zero*.

3.2 States, Predicates, and Relations

In this section, we follow the definitions of [Back and von Wright, 1998]. In higher-order logic, *types* are expressions that denote sets, and new types can be constructed from existing types, *e.g.*, $\Sigma \to \Gamma$ represents the type of functions with inputs of type Σ and outputs of type Γ. The Boolean type *Bool* consists of the truth values *False* and *True*. As a special case, $\Sigma \to \mathit{Bool}$ represents the type of sets on Σ, with the

value interpreted as membership. We write the equality in definitions as $\widehat{=}$ instead of as $=$, and we write the equality of truth values as $\equiv$ instead of as $=$. Arithmetic operators bind stronger than $=$, which itself binds stronger than Boolean operators, which themselves bind stronger than $\equiv$.

Theorem 3.1. ✓ *Truth values form a lattice with implication $\Rightarrow$ as the order relation, conjunction $\wedge$ as the meet, disjunction $\vee$ as the join, False as the bottom element, and True as the top element. The lattice is complete, with set conjunction $\wedge_s$ and set disjunction $\vee_s$ being the meet and the join of sets. The lattice is also distributive as $\wedge$ distributes over $\vee$ and vice versa. This extends to* infinite distributivity*, i.e., for all $c : Bool$ and $\mathfrak{b} : Bool \to Bool$, in which $Bool \to Bool$ is a set of Bool:*

$$c \vee (\wedge_s \mathfrak{b}) \equiv \wedge_s \{c \vee b \mid b \in \mathfrak{b}\}$$

$$c \wedge (\vee_s \mathfrak{b}) \equiv \vee_s \{c \wedge b \mid b \in \mathfrak{b}\}$$

We use lower case Greek letters (*e.g.*, σ, γ) for program states, and corresponding upper case Greek letters (*e.g.*, Σ, Γ) for corresponding program state spaces. For generality, we do not specify the inner structure of a program state, unless it is necessary (*e.g.*, for defining assignment statements).

A *state predicate* (abbreviated as *predicate*) of type $\mathcal{P}\Sigma$ is a function from Σ, the type of a state space, to *Bool*, *i.e.*,

$$\mathcal{P}\Sigma \mathrel{\widehat{=}} \Sigma \to Bool$$

The type of sets on Σ is also $\mathcal{P}\Sigma$. We use lower case letters (*e.g.*, p, q) for predicates.

On predicates, entailment $\leq$ is defined by pointwise extension:

$$p \leq q \mathrel{\widehat{=}} \forall \sigma.\, p\,\sigma \Rightarrow q\,\sigma$$

Conjunction $\wedge$, disjunction $\vee$, implication $\Rightarrow$, negation $\neg$, set conjunction $\wedge_s$, and set disjunction $\vee_s$ are defined by the corresponding operations on *Bool*. For predicates

$p, q : \mathcal{P}\Sigma$ and state $\sigma : \Sigma$ we have

$$(p \wedge q)\,\sigma \mathrel{\widehat{=}} p\,\sigma \wedge q\,\sigma$$

$$(p \vee q)\,\sigma \mathrel{\widehat{=}} p\,\sigma \vee q\,\sigma$$

$$(\neg p)\,\sigma \mathrel{\widehat{=}} \neg(p\,\sigma)$$

$$p \Rightarrow q \mathrel{\widehat{=}} (\neg p) \vee q$$

and for predicate set $\mathfrak{p} : \mathcal{P}(\mathcal{P}\Sigma)$ we have

$$(\wedge_s \mathfrak{p})\,\sigma \mathrel{\widehat{=}} \wedge_s \{p\,\sigma \mid p \in \mathfrak{p}\}$$

$$(\vee_s \mathfrak{p})\,\sigma \mathrel{\widehat{=}} \vee_s \{p\,\sigma \mid p \in \mathfrak{p}\}$$

The bottom and top predicates false and true represent the universally *False* and *True* truth values:

$$\mathsf{false}\ \sigma \mathrel{\widehat{=}} \mathit{False}$$

$$\mathsf{true}\ \sigma \mathrel{\widehat{=}} \mathit{True}$$

The pointwise extension of a (complete, distributive) lattice is again a (complete, distributive) lattice [Back and von Wright, 1998]. Therefore we have:

Theorem 3.2. ✓ *Predicates on $\mathcal{P}\Sigma$ with entailment $\leq$ as the order relation form a complete distributive lattice, with* false, true, $\wedge$, $\vee$, $\wedge_s$ *and* $\vee_s$ *being the bottom, top, meet, join, set meet, and set join operations, respectively.*

A *state relation* (abbreviated as *relation*) is a function from Σ, the initial state space, to a predicate on Γ, the final state space, *i.e.*, a function of type $\Sigma \to \mathcal{P}\Gamma$, which is isomorphic to $(\Sigma \times \Gamma) \to \mathit{Bool}$, but more natural to write in HOL. We allow the initial and final state spaces to be different. The *empty relation* $\bot$, the *universal relation* $\top$, and the *identity relation* id are defined by

$$\bot\,\sigma\,\gamma \mathrel{\widehat{=}} \mathit{False}$$

$$\top\,\sigma\,\gamma \mathrel{\widehat{=}} \mathit{True}$$

$$\mathsf{id}\ \sigma\,\sigma' \mathrel{\widehat{=}} \sigma = \sigma'$$

Intersection $\cap$, union $\cup$, complement $\overline{r}$, and composition $\circ$ can be defined straightforwardly, but are not needed in this thesis; *inclusion* is defined by

$$r \subseteq r' \mathrel{\widehat{=}} \forall \sigma.\, r\, \sigma \leq r'\, \sigma$$

3.3 Indexed Predicates and Indexed Relations

In this section we introduce indexed predicates and indexed relations, with the indices being the names of exits. For a tuple $t = (t_1, \ldots, t_n)$, we write $t\, i$ for selecting its i-th element. An *indexed predicate* is a tuple $(p_1, \ldots, p_n)$ of predicates p_i. We call $I = \{1, \ldots, n\}$ its *exit indices* and write the type $\mathcal{P}\Gamma_1 \times \cdots \times \mathcal{P}\Gamma_n$ of an indexed predicate more concisely as $\mathcal{P}\Gamma_I$[1]. The exit $\mathsf{nrl} = 1$ is the normal exit. Generally, every imperative program block contains a normal exit, indexed $\mathsf{nrl} = 1$. We specify other exit indices later as we study more specific program languages. An indexed predicate models a tuple of predicates on all exits, with $P\, i$ corresponding to the predicates on each exit i. We use upper case letters (*e.g.*, P, Q) for indexed predicates.

On indexed predicates, entailment $\leq$ is again defined by pointwise extension:

$$P \leq Q \mathrel{\widehat{=}} \forall\, i.\, P\, i \leq Q\, i$$

Conjunction $\wedge$, disjunction $\vee$, implication $\Rightarrow$, negation $\neg$, set conjunction $\wedge_s$, and set disjunction $\vee_s$ are defined by the corresponding operations on $\mathcal{P}\Gamma$. For indexed

[1]The proof assistant Isabelle does support dependent types, so in Isabelle we model both arbitrary number of exits with the same state space, and various state spaces on exits of a fixed number. The proofs in the thesis do not rely on the uniformity of the final state spaces, unless restricted by certain constructs. Please refer to Appendix A for detailed discussion.

predicates $P, Q : I \to \mathcal{P}\Gamma$ and exit index $i : I$ we have:

$$(P \wedge Q)\, i \mathrel{\widehat{=}} P\, i \wedge Q\, i$$

$$(P \vee Q)\, i \mathrel{\widehat{=}} P\, i \vee Q\, i$$

$$(\neg P)\, i \mathrel{\widehat{=}} \neg(P\, i)$$

$$P \Rightarrow Q \mathrel{\widehat{=}} (\neg P) \vee Q$$

and for $\mathfrak{P} : \mathcal{P}(I \to \mathcal{P}\Gamma)$ we have

$$(\wedge_s \mathfrak{P})\, i \mathrel{\widehat{=}} \wedge_s \{P\, i \mid P \in \mathfrak{P}\}$$

$$(\vee_s \mathfrak{P})\, i \mathrel{\widehat{=}} \vee_s \{P\, i \mid P \in \mathfrak{P}\}$$

The bottom and top elements of type $I \to \mathcal{P}\Gamma$ are $\overline{\mathsf{false}}$ and $\overline{\mathsf{true}}$ respectively, defined by

$$\overline{\mathsf{false}}\, i \mathrel{\widehat{=}} \mathsf{false}$$

$$\overline{\mathsf{true}}\, i \mathrel{\widehat{=}} \mathsf{true}$$

Again the result of pointwise extension is a (complete, distributive) lattice:

Theorem 3.3. ✓ *Indexed predicates on $I \to \mathcal{P}\Gamma$ with entailment $\leq$ as the order relation form a complete distributive lattice, with $\wedge$, $\vee$, $\wedge_s$ and $\vee_s$ being the meet, join, set meet, and set join operations, respectively.*

An *indexed relation* is a tuple $(r_1, \ldots, r_n)$ of relations with the same initial state space but possibly different final state spaces, *i.e.*, is of type $(\Sigma \to \mathcal{P}\Gamma_1) \times \cdots \times (\Sigma \to \mathcal{P}\Gamma_n)$. With exit indices $I = \{1, \ldots, n\}$ we write this more concisely as $\Sigma \xrightarrow{\mathcal{I}} \mathcal{P}\Gamma$. We write $\overline{\top}$, $\overline{\bot}$, and $\overline{\mathsf{id}}$ for the indexed relations that are universally $\top$, $\bot$, and id. We use upper case letters (*e.g.*, R) for indexed predicates.

3.4 Indexed Predicate Transformers

A indexed predicate transformer, which is a model of single-exit programs, is a function from $\mathcal{P}\Gamma$, the type of a postcondition, to $\mathcal{P}\Sigma$, the type of a precondition, *i.e.*,

of type $\mathcal{P}\Gamma \to \mathcal{P}\Sigma$. An indexed predicate transformer is an extension for modelling multi-exit programs. It is of type $\mathcal{P}\Gamma_I \to \mathcal{P}\Sigma$, a function from $\mathcal{P}\Sigma_I$, the type of indexed postcondition, to $\mathcal{P}\Sigma$, the type of a precondition. We use upper case letters (*e.g.*, S, T) for indexed predicate transformers.

3.5 Basic Multi-Exit Statements

First we define indexed predicate transformers for some basic multi-exit statements: abort *aborts* execution, meaning it may terminate at any exit in any state or may not terminate at all; stop, also known as magic, *blocks* execution, thus satisfies miraculously any postcondition on any exit; jump i does not change the state and terminates at exit i. With indexed predicate Q we define:

$$\mathsf{abort}\ Q \mathrel{\widehat{=}} \mathsf{false}$$

$$\mathsf{stop}\ Q \mathrel{\widehat{=}} \mathsf{true}$$

$$\mathsf{jump}\, i\ Q \mathrel{\widehat{=}} Q\, i$$

We use skip as the abbreviation of jump nrl, thus $\mathsf{skip}\ Q = \mathsf{jump}\,\mathsf{nrl}\ Q = Q\ \mathsf{nrl}$.

Let G be an indexed predicate. The *assumption* $[G]$ allows continuation at exit i if $G\, i = \mathsf{true}$. If continuation is possible at several exits, the choice is *demonic*, which means that every choice must establish the corresponding postcondition. If continuation is not possible, the assumption stops. The *assertion* $\{G\}$ also allows continuation at exit i if $G\, i = \mathsf{true}$. If continuation is possible at several exits, the choice is *angelic*, which means that at least one choice must establish the corresponding postcondition. If continuation is not possible, the assertion aborts:

$$[G]\ Q \mathrel{\widehat{=}} \wedge_s \{G\, i \Rightarrow Q\, i \mid i \in I\}$$

$$\{G\}\ Q \mathrel{\widehat{=}} \vee_s \{G\, i \wedge Q\, i \mid i \in I\}$$

3.6 Composite Multi-Exit Statements

Now we define indexed predicate transformers for composite multi-exit statements. Let S, T be multi-exit statements. The *sequential composition* $S \,;_i\, T$ executes first S and provided that S terminates at exit i, continues with T. The *demonic choice* $S \sqcap T$ establishes a postcondition at an exit if both S and T do. The *angelic choice* $S \sqcup T$ establishes a postcondition at an exit if either S or T does. We write $f[i \leftarrow v]$ for updating function f to evaluate to v at i:

$$(S \,;_i\, T)\; Q \mathrel{\widehat{=}} S\,(Q[i \leftarrow T\; Q])$$

$$(S \sqcap T)\; Q \mathrel{\widehat{=}} S\; Q \wedge T\; Q$$

$$(S \sqcup T)\; Q \mathrel{\widehat{=}} S\; Q \vee T\; Q$$

In sequential composition, $T\; Q$ is the weakest precondition for T to terminate with Q, and it is also the postcondition of S on the ith exit.

Binary choice generalizes to choice over arbitrary sets. Let $\mathfrak{S}$ be a set of predicate transformers:

$$(\sqcap_s \mathfrak{S})\; Q \mathrel{\widehat{=}} \wedge_s \{S\; Q \mid S \in \mathfrak{S}\}$$

$$(\sqcup_s \mathfrak{S})\; Q \mathrel{\widehat{=}} \vee_s \{S\; Q \mid S \in \mathfrak{S}\}$$

The demonic choice over the empty set blocks, $\sqcap_s \emptyset = \mathsf{stop}$, and the angelic choice over the empty set aborts, $\sqcup_s \emptyset = \mathsf{abort}$.

We have that $[\overline{\mathsf{false}}] = \mathsf{stop}$ and that $[\overline{\mathsf{true}}] = \sqcap_s\{\mathsf{jump}\, i \mid i\}$. Dually, we have that $\{\overline{\mathsf{false}}\} = \mathsf{abort}$ and that $\{\overline{\mathsf{true}}\} = \sqcup_s\{\mathsf{jump}\, i \mid i\}$. For predicates $p_1, \ldots, p_k$ and distinct $i_1, \ldots, i_k \in I$ we write $i_1 \mapsto p_1, \ldots, i_k \mapsto p_k$ for the indexed predicate that is p_1 at i_1, $\ldots$, p_k at i_k, and false everywhere else. As a special case, $i \mapsto \mathsf{true}$ is the indexed predicate that is true for $i \in I$ and false otherwise. In the case of assertions and assumptions with only one predicate being true and all other false, the continuation is deterministic, in the sense that $[i \mapsto \mathsf{true}] = \mathsf{jump}\, i = \{i \mapsto \mathsf{true}\}$. If

only a predicate for the normal exit is specified, we write $[p]$ for $[\mathsf{nrl} \mapsto p]$ and likewise $\{p\}$ for $\{\mathsf{nrl} \mapsto p\}$.

Theorem 3.4. ✓ *Let p be a predicate and Q be an indexed predicate:*

$$[p]\ Q = (p \Rightarrow Q\ \mathsf{nrl})$$
$$\{p\}\ Q = (p \wedge Q\ \mathsf{nrl})$$

An *update* specifies a state change by one relation for each exit, each relation relating the common initial state to the final state at each exit. Let R be an indexed relation. The *demonic update* $[R]$ allows continuation at exit i from initial state σ if $R\,i\,\sigma$ specifies some final states. The choice among all possible final states and the choice among all possible exits are demonic. If continuation is not possible, the demonic update stops. Dually, the *angelic update* $\{R\}$ allows continuation at exit i from initial state σ if $R\,i\,\sigma$ specifies some final states. The choice among all possible final states and the choice among all possible exits are angelic. If continuation is not possible, the angelic update aborts:

$$[R]\ Q\,\sigma \mathrel{\widehat{=}} (\forall\, i.\, \forall\, \gamma.\, R\,i\,\sigma\,\gamma \Rightarrow Q\,i\,\gamma)$$
$$\{R\}\ Q\,\sigma \mathrel{\widehat{=}} (\exists\, i.\, \exists\, \gamma.\, R\,i\,\sigma\,\gamma \wedge Q\,i\,\gamma)$$

We have that $[\overline{\bot}] = \mathsf{stop}$ and that $[\overline{\mathsf{id}}] = \sqcap_s\{\mathsf{jump}\,i \mid i\}$. Dually, we have that $\{\overline{\bot}\} = \mathsf{abort}$ and that $\{\overline{\mathsf{id}}\} = \sqcup_s\{\mathsf{jump}\,i \mid i\}$. Both updates $[\overline{\top}]$ and $\{\overline{\top}\}$ always terminate in some state at some exit, with $[\overline{\top}]$ making the choice among the exits and among the states demonic and $\{\overline{\top}\}$ making these choices angelic. In the case of updates with only one relation being the identity and all other relations being empty, the continuation is deterministic, in the sense that $[i \mapsto \mathsf{id}] = \mathsf{jump}\,i = \{i \mapsto \mathsf{id}\}$. For the case that only a relation for the normal exit is specified, we write $[r]$ for $[\mathsf{nrl} \mapsto r]$ and likewise $\{r\}$ for $\{\mathsf{nrl} \mapsto r\}$. We call them *normal demonic update* and *normal angelic update*, respectively.

Theorem 3.5. ✓ *Let r be a relation and Q be an indexed predicate:*

$$[r]\, Q\, \sigma = (\forall\, \gamma.\, r\, \sigma\, \gamma \Rightarrow Q\ \mathsf{nrl}\ \gamma)$$
$$\{r\}\, Q\, \sigma = (\exists\, \gamma.\, r\, \sigma\, \gamma \wedge Q\ \mathsf{nrl}\ \gamma)$$

For an indexed predicate $G : \mathcal{P}\Sigma_I$, the *lifting* $|G| : \Sigma \xrightarrow{\mathcal{I}} \mathcal{P}\Sigma$ is an indexed relation that, for each exit i, is a partial identity relation:

$$|G|\, i\, \sigma\, \sigma' \mathrel{\widehat{=}} G\, i\, \sigma \wedge \sigma = \sigma'$$

Assumptions and assertions can be expressed as demonic and angelic updates by lifting their argument to an indexed relation:

Theorem 3.6. ✓ *Let G be an indexed predicate:*

$$[G] = [|G|]$$
$$\{G\} = \{|G|\}$$

A *multi-exit statement* (or *statement* for short) S is an indexed predicate transformer that is constructed using only

1. *basic multi-exit statements* abort, stop, $\mathsf{jump}\, i$, $[G]$, $\{G\}$, $[R]$, $\{R\}$, where i is an exit index, G is an indexed predicate, R is an indexed relation, and

2. *composite multi-exit statements* $S\ ;_i\ T$, $\sqcap_s\, \mathfrak{S}$, $\sqcup_s\, \mathfrak{S}$, where S, T are statements and $\mathfrak{S}$ is a set of statements.

We write skip for the jump statement to the normal exit, and write ; for the *normal sequential composition*:

$$\mathsf{skip} \mathrel{\widehat{=}} \mathsf{jump}\, \mathsf{nrl}$$
$$S\ ;\ T \mathrel{\widehat{=}} S\ ;_{\mathsf{nrl}}\ T$$

Exception handling is expressed in terms of the statement $\mathsf{raise}\, i$ to raise exception i and the statement $\mathsf{try}\, S\, \mathsf{on}\, i\, \mathsf{do}\, T$ to start with S and on exception i to continue with

T, otherwise to continue normally. With exceptions being exits, these are defined as:

$$\mathsf{raise}\, i \mathrel{\widehat{=}} \mathsf{jump}\, i \qquad \text{if } i \neq \mathsf{nrl}$$

$$\mathsf{try}\, S\, \mathsf{on}\, i\, \mathsf{do}\, T \mathrel{\widehat{=}} S \mathbin{;_i} T \qquad \text{if } i \neq \mathsf{nrl}$$

3.7 Algebraic Properties

For exit index i, indexed predicate transformers with $;_i$ as composition on exit i and $\mathsf{jump}\, i$ as unit form a monoid. For distinct exit indices i, j, we only have that $\mathsf{jump}\, i$ is a left zero of $;_j$.

Theorem 3.7. ✓ *Let I be the exit indices and let S, T, U be indexed predicate transformers. For any $i, j \in I$ that satisfies $i \neq j$ we have:*

$$(S \mathbin{;_i} T) \mathbin{;_i} U = S \mathbin{;_i} (T \mathbin{;_i} U) \qquad \text{(associativity)}$$

$$S \mathbin{;_i} \mathsf{jump}\, i = S \qquad \text{(right zero of } ;_i\text{)}$$

$$\mathsf{jump}\, i \mathbin{;_i} S = S \qquad \text{(left unit of } ;_i\text{)}$$

$$\mathsf{jump}\, j \mathbin{;_i} S = \mathsf{jump}\, j \qquad \text{(left zero of } ;_i\text{)}$$

As the unit of a monoid is unique, we have that the unique unit of $;_i$ is $\mathsf{jump}\, i$. To see that $(S \mathbin{;_i} T) \mathbin{;_j} U \neq S \mathbin{;_i} (T \mathbin{;_j} U)$ in general, consider that $i = \mathsf{nrl}$ and $j \neq \mathsf{nrl}$. Rewriting using try statements, it is intuitive that $\mathsf{try}\, S \,;\, T\, \mathsf{on}\, j\, \mathsf{do}\, U$ is in general different from $S \,;\, \mathsf{try}\, T\, \mathsf{on}\, j\, \mathsf{do}\, U$. Assuming $i, j \neq \mathsf{nrl}$ and rewriting $S \mathbin{;_j} \mathsf{jump}\, i$ as $\mathsf{try}\, S\, \mathsf{on}\, j\, \mathsf{do}\, \mathsf{raise}\, i$ we obtain the idiom of *re-raising* an exception; in this case, $\mathsf{jump}\, i$ (or $\mathsf{raise}\, i$) is neither unit nor zero.

The refinement relation is defined again by pointwise extension from predicates. For indexed predicate transformers S, T we define:

$$S \sqsubseteq T \mathrel{\widehat{=}} \forall\, Q.\, S\, Q \leq T\, Q$$

Intuitively, refinement may reduce demonic choice and may increase angelic choice,

where the choice is between exits or states. For example, $\mathsf{jump}\, i \sqcap \mathsf{jump}\, j \sqsubseteq \mathsf{jump}\, i$. This is captured by the lattice structure of indexed predicate transformers.

Theorem 3.8. ✓ *Indexed predicate transformers with* $\sqsubseteq$ *as the order relation,* abort *as bottom,* stop *as top,* $\sqcap$ *as meet,* $\sqcup$ *as join,* $\sqcap_s$ *as set meet, and* $\sqcup_s$ *as set join form a complete distributive lattice.*

3.8 Program Expressions

A *value function* of type $\Sigma \to V$ is a total function from state space Σ to value type V. However, before embarking on modelling statements in concrete programming languages, we need to determine how to treat possible undefinedness in program expressions. We distinguish *terms* in the underlying logic, here HOL, from *program expressions*, here those of programming languages. A Boolean term, even one like $x/y > 0$ and $a[x] < k$ is always true or false. However, the program expressions $x/y > 0$ and $a[x] < k$ may not always yield a result. For program expression E its *definedness* ΔE and *value* 'E' are in part determined by the underlying machine; the result of ΔE and 'E' are terms. Since the value function can be complicated because it involves definedness, for readability we write 'E' $\cong v$ if $\Delta E \leq (\lambda\, \sigma.\, \text{'}E\text{'}\, \sigma = v\, \sigma)$, which means that when E is defined at σ, the value 'E'$\,\sigma$ equals $v\, \sigma$. Formally, a program expression is a partial function, with its domain being the state space Σ, and its range being V. In Isabelle, the value of a partial function is *None* where it is undefined.

We consider a subset of Java operators on Booleans and integers: assuming that c is a constant, x a variable, E and F program expressions, and $\approx$ is $=$, $<$ or another relational operator, $\circ$ is $+$, $-$, or $*$, and $|$ is div or mod (integer division and modulo),

we have

$$\begin{aligned}
\Delta c &= \mathsf{true} & \text{`}c\text{'} &\approxeq \lambda\,\sigma.\, c \\
\Delta x &= \mathsf{true} & \text{`}x\text{'} &\approxeq \lambda\,\sigma.\, x\,\sigma \\
\Delta(E \mathsf{\ and\ } F) &= \Delta E \wedge \Delta F & \text{`}E \mathsf{\ and\ } F\text{'} &\approxeq \text{`}E\text{'} \wedge \text{`}F\text{'} \\
\Delta(E \mathsf{\ or\ } F) &= \Delta E \wedge \Delta F & \text{`}E \mathsf{\ or\ } F\text{'} &\approxeq \text{`}E\text{'} \vee \text{`}F\text{'} \\
\Delta(E \mathsf{\ and\ then\ } F) &= \Delta E \wedge (\text{`}E\text{'} \Rightarrow \Delta F) & \text{`}E \mathsf{\ and\ then\ } F\text{'} &\approxeq \text{`}E\text{'} \wedge \text{`}F\text{'} \\
\Delta(E \mathsf{\ or\ else\ } F) &= \Delta E \wedge (\neg\text{`}E\text{'} \Rightarrow \Delta F) & \text{`}E \mathsf{\ or\ else\ } F\text{'} &\approxeq \text{`}E\text{'} \vee \text{`}F\text{'} \\
\Delta(E \approx F) &= \Delta E \wedge \Delta F & \text{`}E \approx F\text{'} &\approxeq \text{`}E\text{'} \approx \text{`}F\text{'} \\
\Delta(E \mid F) &= \Delta E \wedge \Delta F \wedge \text{`}F\text{'} \neq 0 & \text{`}E \mid F\text{'} &\approxeq \text{`}E\text{'} \mid \text{`}F\text{'} \\
\Delta(E \circ F) &= \Delta E \wedge \Delta F \wedge & \text{`}E \circ F\text{'} &\approxeq \text{`}E\text{'} \circ \text{`}F\text{'} \\
&\quad \mathsf{min} \leq \text{`}E \circ F\text{'} \leq \mathsf{max}
\end{aligned}$$

where min and max are the smallest and largest machine-representable integers, operators and and or evaluate both operands, and operators and then (*conditional conjunction*, or *conditional and*, also written as $\wedge_c$) and or else (*conditional disjunction*, or *conditional or*, also written as $\vee_c$) evaluate conditionally. For example, with the state being a pair of integers (l, u):

Theorem 3.9. ✓ *Assuming that* $\mathsf{min} \leq 0 \leq l \leq u \leq \mathsf{max}$*,*

$$\Delta((l+u)\,\mathsf{div}\,2) = \lambda\,(l,u).\, l + u \leq \mathsf{max}$$

$$\text{`}(l+u)\,\mathsf{div}\,2\text{'} \approxeq \lambda\,(l,u).\,(l+u)\,\mathsf{div}\,2$$

and:

$$\Delta(l + (u-l)\,\mathsf{div}\,2) = \mathsf{true}$$

$$\begin{aligned}
\text{`}l + (u-l)\,\mathsf{div}\,2\text{'} &\approxeq \lambda\,(l,u).\, l + (u-l)\,\mathsf{div}\,2 \\
&= \lambda\,(l,u).\,(l+u)\,\mathsf{div}\,2
\end{aligned}$$

That is, program expressions $(l+u)\,\mathsf{div}\,2$ and $l+(u-l)\,\mathsf{div}\,2$ have the same value when defined, namely the term $(l+u)\,\mathsf{div}\,2$, but the latter one is always defined under

above assumption, whereas the former is not.

The distinction between terms in the logic and program expressions keeps the logic simple, *e.g.*, all familiar laws of the Boolean algebra like the law of the excluded middle still hold, while allowing the capture of all restrictions of an underlying machine.

3.9 Monotonicity and Junctivity

We characterize different classes of statements. Indexed predicate transformer S is *monotonic* if and only if $Q \leq Q' \Rightarrow S\,Q \leq S\,Q'$ for arbitrary indexed predicates Q, Q'. An immediate consequence is:

Theorem 3.10. ✓ *All basic statements are monotonic and all composed statements preserve monotonicity.*

By monotonicity of S we have that for arbitrary i, $tr_i\,S \leq tr\,S$.

Positive "junctivity" properties are defined by distributivity over arbitrary, but non-empty sets of indexed predicates. Indexed predicate transformer S is *positively conjunctive* if and only if $S\,(\sqcap_s \mathfrak{Q}) = \wedge_s\{S\,Q \mid Q \in \mathfrak{Q}\}$ and *positively disjunctive* if and only if $S\,(\sqcup_s \mathfrak{Q}) = \vee_s\{S\,Q \mid Q \in \mathfrak{Q}\}$ for arbitrary set $\mathfrak{Q} \neq \emptyset$ of indexed predicates.

Theorem 3.11. ✓ *Statements* stop, $\mathsf{jump}\,i$, *assertion* $\{G\}$, *assumption* $[G]$, *and demonic update* $[R]$ *are positively conjunctive. Sequential composition* $;_i$ *and demonic choice* $\sqcap$ *preserve positive conjunctivity.*

Angelic choice does not preserve positive conjunctivity in general. A dual theorem holds for positively disjunctive statements.

Theorem 3.12. ✓ *Statements* abort *and* $\mathsf{jump}\,i$, *assumption* $[G]$, *assertion* $\{G\}$, *and angelic update* $\{R\}$ *are positively disjunctive. Sequential composition* $;_i$ *and angelic choice* $\sqcup$ *preserve positive disjunctivity.*

Universal "junctivity" properties are defined by distributivity over arbitrary sets of indexed predicates. Indexed predicate transformer S is *universally conjunctive* if $S\,(\sqcap_s \mathfrak{Q}) = \wedge_s\{S\ Q \mid Q \in \mathfrak{Q}\}$ and *universally disjunctive* if $S\,(\sqcup_s \mathfrak{Q}) = \vee_s\{S\ Q \mid Q \in \mathfrak{Q}\}$ for arbitrary set $\mathfrak{Q}$ of indexed predicates.

Theorem 3.13. ✓ *Any universally conjunctive indexed predicate transformer is positively conjunctive. Any positively conjunctive indexed predicate transformer is monotonic.*

It is easy to see that abort is not universally conjunctive by taking $\mathfrak{Q} = \emptyset$. As a consequence, assertion $\{G\}$ is in general not universally conjunctive either. Angelic choice does not preserve universal conjunctivity. The relationship between universal and positive "junctivity" can be stated more precisely by considering domains.

Theorem 3.14. ✓ *A indexed predicate transformer is universally conjunctive if and only if it is positively conjunctive and terminating.*

As an immediate consequence, stop, jump i, assumption $[G]$, and demonic update $[R]$ are universally conjunctive. Also, sequential composition $;_i$ and demonic choice $\sqcap$ preserve universal conjunctivity. Dually, we have:

Theorem 3.15. ✓ *A indexed predicate transformer is universally disjunctive if and only if it is positively disjunctive and enabled.*

The monoid and lattice structure are connected by following distributivity properties.

Theorem 3.16. ✓ *Let S be an indexed predicate transformer and $\mathfrak{T}$ be a non-empty*

set of indexed predicate transformers:

$$(\sqcap_s \mathfrak{T}) ;_i S = \sqcap_s \{ T ;_i S \mid T \in \mathfrak{T} \}$$

$$S ;_i (\sqcap_s \mathfrak{T}) \sqsubseteq \sqcap_s \{ S ;_i T \mid T \in \mathfrak{T} \} \qquad \textit{if } S \textit{ is monotonic}$$

$$S ;_i (\sqcap_s \mathfrak{T}) = \sqcap_s \{ S ;_i T \mid T \in \mathfrak{T} \} \qquad \textit{if } S \textit{ is positively conjunctive}$$

3.10 Domains

For multi-exit statements, the *termination domain*, written as $tr\, S$, identifies when statement S does not abort. The termination domain on exit i, written as $tr_i\, S$ identifies when statement S terminates on exit i. The *enabledness domain*, written as $en\, S$, identifies when the statement does not block. We say a indexed predicate transformer S is *terminating* if $tr\, S = \mathsf{true}$ and that S is *enabled* (or *strict*) if $en\, S = \mathsf{true}$.

$$tr\, S \mathrel{\widehat{=}} S\, \overline{\mathsf{true}}$$

$$tr_i\, S \mathrel{\widehat{=}} S(i \mapsto \mathsf{true})$$

$$en\, S \mathrel{\widehat{=}} \neg(S\, \overline{\mathsf{false}})$$

The next theorem summarizes the basic properties of the domain operations.

Theorem 3.17. ✓ *Let S, T be statements, G be an indexed predicate, R be a relation, i, j be two different exit indices:*

$$tr\ \mathsf{abort} = \mathsf{false} \qquad tr\ \mathsf{stop} = \mathsf{true} \qquad tr\, \mathsf{jump}\, i = \mathsf{true}$$

$$tr_i\ \mathsf{abort} = \mathsf{false} \qquad tr_i\ \mathsf{stop} = \mathsf{true} \qquad tr_i\, \mathsf{jump}\, i = \mathsf{true}$$

$$tr_j\, \mathsf{jump}\, i = \mathsf{false}$$

$$en\ \mathsf{abort} = \mathsf{true} \qquad en\ \mathsf{stop} = \mathsf{false} \qquad en\, \mathsf{jump}\, i = \mathsf{true}$$

$$tr\, [G] = \mathsf{true} \qquad tr\, \{G\} = \vee_s \{ G\, i \mid i \}$$

$$en\, [G] = \vee_s \{ G\, i \mid i \} \qquad en\, \{G\} = \mathsf{true}$$

$$tr\,(S\ ;_i\ T) \Rightarrow tr\,S \qquad en\,(S\ ;_i\ T) \Rightarrow en\,S$$
$$tr\,(S \sqcap T) = tr\,S \wedge tr\,T \qquad tr\,(S \sqcup T) = tr\,S \vee tr\,T$$
$$en\,(S \sqcap T) = en\,S \vee en\,T \qquad en\,(S \sqcup T) = en\,S \wedge en\,T$$
$$tr\,[R] = \mathsf{true} \qquad tr\,\{R\} = (\lambda\,\sigma.\,\exists\,i, \gamma.\,R\,i\,\sigma\,\gamma)$$
$$en\,[R] = (\lambda\,\sigma.\,\exists\,i, \gamma.\,R\,i\,\sigma\,\gamma)) \qquad en\,\{R\} = \mathsf{true}$$

Statement stop blocks execution, hence $en\ \mathsf{stop} = \mathsf{false}$, and stop "terminates orderly" by refusing execution, hence $tr\ \mathsf{stop} = \mathsf{true}$.

3.11 Total Correctness

Hoare's total correctness assertion $[\![p]\!]\,S\,[\![q]\!]$ states that under precondition p, statement S terminates with postcondition q. This is now generalized to indexed postconditions.

Definition 3.4. *Total correctness assertion $[\![p]\!]\,S\,[\![Q]\!]$ states that under precondition p, statement S terminates with postcondition Q,* i.e.*,*

$$[\![p]\!]\,S\,[\![Q]\!] \mathrel{\widehat{=}} p \leq S\,Q$$

The next theorem summarizes the basic properties of total correctness; they extend those in [Cristian, 1984; King and Morgan, 1995]:

Theorem 3.18. ✓ *Let p be any predicate, Q be any indexed predicate, G be any indexed predicate, R be an indexed relation, and S, T be statements:*

$$[\![p]\!]\ \mathsf{abort}\ [\![Q]\!] \equiv p = \mathsf{false}$$
$$[\![p]\!]\ \mathsf{stop}\ [\![Q]\!] \equiv \mathit{True}$$
$$[\![p]\!]\,\mathsf{jump}\,i\,[\![Q]\!] \equiv p \Rightarrow Q\,i$$
$$[\![p]\!]\,[G]\,[\![Q]\!] \equiv p \Rightarrow (\wedge_s\{G\,i \Rightarrow Q\,i \mid i \in I\})$$

$$
\begin{aligned}
[\![p]\!]\,\{G\}\,[\![Q]\!] &\equiv p \Rightarrow (\vee_s\{G\,i \wedge Q\,i \mid i \in I\}) \\
[\![p]\!]\,S\,;_i\,T\,[\![Q]\!] &\equiv \exists\, h.\, [\![p]\!]\,S\,[\![Q[i \leftarrow h]]\!] \wedge [\![h]\!]\,T\,[\![Q]\!] \\
[\![p]\!]\,S \sqcap T\,[\![Q]\!] &\equiv [\![p]\!]\,S\,[\![Q]\!] \wedge [\![p]\!]\,T\,[\![Q]\!] \\
[\![p]\!]\,[R]\,[\![Q]\!] &\equiv p \Rightarrow (\lambda\,\sigma.\, \forall\, i\,\gamma.\, R\,i\,\sigma\,\gamma \Rightarrow Q\,i\,\gamma) \\
[\![p]\!]\,\{R\}\,[\![Q]\!] &\equiv p \Rightarrow (\lambda\,\sigma.\, \exists\, i\,\gamma.\, R\,i\,\sigma\,\gamma \wedge Q\,i\,\gamma)
\end{aligned}
$$

Definition 3.5. *A precondition p and an indexed postcondition Q can specify a set of statements $\{S \mid [\![p]\!]\,S\,[\![Q]\!]\}$, in which any S is monotonic. We denote the set meet of this set as $\lfloor p, Q \rfloor$:*

$$\lfloor p, Q \rfloor\ Q' = \mathit{if}\ Q \leq Q'\ \mathit{then}\ p\ \mathit{else}\ \mathsf{false}$$

Apparently it is monotonic and it is refined by any statement in the set $\{S \mid [\![p]\!]\,S\,[\![Q]\!]\}$, so it is the least specified statement *of given p and Q.*

Theorem 3.19. ✓ *Let p be a predicate and Q be an indexed predicate,*

$$[\![p]\!]\,\lfloor p, Q \rfloor\,[\![Q]\!]$$
$$\forall\, S.\, [\![p]\!]\,S\,[\![Q]\!] \Rightarrow \lfloor p, Q \rfloor \sqsubseteq S$$

Chapter 4

Recursion and Iteration

Recursion is suitable for solving problems when each solution is dependent on the same problem of smaller instances [Graham et al., 1994], and iteration repeats a block of statements. Back and von Wright [1998] study recursion and iteration rules for single-exit programs. We extend them to multi-exit programs, starting from giving the definitions of ranked predicates and least fixed points.

4.1 Ranked Predicates and Least Fixed Points

Definition 4.1. *(Section 18.1 of [Back and von Wright, 1998]) A* well-founded *set is a poset* $(W, \leq)$ *in which every non-empty subset of* W *has a minimal element,* i.e.*, if*

$$\forall\, X \subseteq W.\, X \neq \emptyset \Rightarrow (\exists\, x \in X.\, \forall\, y \in X.\, y \not< x)$$

Definition 4.2. *Let* $\{p_w \mid w \in W\}$ *be a collection of predicates that are indexed by the well-founded non-empty set* W *such that* $w < w' \Rightarrow p_w \leq p_{w'}$*. A predicate in such a set is called a* ranked predicate*. A ranked predicate* p_w *is often written as the*

conjunction of an invariant *expression* p *and a* variant *expression* v*:*

$$p_w \mathrel{\widehat{=}} (\lambda\, \sigma.\, p\, \sigma \wedge v\, \sigma = w)$$

$$p_{<w} \mathrel{\widehat{=}} \vee_s \{p_{w'} \mid w' \in W \wedge w' < w\}$$

Thus, p *holds if some ranked predicate holds, while* $p_{<w}$ *holds if some ranked predicate with a "lower rank" than that of* p_w *holds.*

Definition 4.2 essentially follows Section 20.2 of [Back and von Wright, 1998], except that the well-founded set W is required to be non-empty here, since it no longer requires defining p as the disjunction of all p_w, and in practice the set is non-empty anyway.

A well-founded set does not have a chain that decreases infinitely, so ranked predicates can be used to guarantee the termination in recursion or iteration.

Definition 4.3. *Let* A *be a complete lattice and* f *a monotonic function on* A*,* i.e.*,* $f \in A \to_m A$*, then the* least fixed point *a in* A *is denoted as* $\mu\, f$*. The least element that satisfies* $f\ a = a$*,* i.e.*,* $\forall\, a' \in A.\, f\ a' = a' \Rightarrow a \sqsubseteq a'$*.*

Knaster-Tarski theorem guarantees the existence of the least fixed point, and it is given by $\sqcap_s\{x \mid x \in A \wedge f\ x \sqsubseteq x\}$ [Tarski, 1955].

4.2 Recursion

Theorem 4.1. ✓ *Assume that* W *is a well-founded set,* f *is a monotonic function on monotonic indexed predicate transformers, and* $\{p_w \mid w \in W\}$ *is a collection of ranked predicates with ranks of type* W*, then the following holds:*

$$(\forall\, w \in W.\, \{p_w\}\, ;\, S \sqsubseteq f\, (\{p_{<w}\}\, ;\, S)) \Rrightarrow \{p\}\, ;\, S \sqsubseteq \mu\, f$$

Proof. Assume that $\{p_w\}\, ;\, S \sqsubseteq f\, (\{p_w\}\, ;\, S)$ holds for all $w \in W$. First we show that

$$(\forall\, w \in W.\, \{p_w\}\, ;\, S \sqsubseteq \mu\, f) \qquad (*)$$

by well-founded induction, as follows. The induction assumption is $(\forall\, v.\, v < w \Rightarrow \{p_v\}\,;S \sqsubseteq \mu\, f)$:

$$
\begin{array}{ll}
 & \{p_w\}\,;S \\
\sqsubseteq & \langle\text{assumption}\rangle \\
 & f(\{\wedge_s\{p_v \mid v \in W \wedge v < w\}\}\,;S) \\
= & \langle\text{homomorphism property of assertion, distributivity}\rangle \\
 & f\,(\sqcap_s\{\{p_v\}\,;S \mid v \in W \wedge v < w\}) \\
\sqsubseteq & \langle\text{induction assumption, monotonicity}\rangle \\
 & f\,(\sqcap_s\{\mu\, f \mid v \in W \wedge v < w\}) \\
\sqsubseteq & \langle\text{empty join gives } \mathsf{abort}\text{, otherwise } \mu\, f\text{, monotonicity}\rangle \\
 & f\,(\mu\, f) \\
= & \langle\text{fold fixed point}\rangle \\
 & \mu\, f
\end{array}
$$

Thus,

$$
\begin{array}{ll}
 & \{p\}\,;S \\
= & \langle\text{definition of } p\rangle \\
 & \{\vee_s\{p_w \mid w \in W\}\}\,;S \\
= & \langle\text{homomorphism property of assertion, distributivity}\rangle \\
 & \sqcup_s\,\{\{p_w\}\,;S \mid w \in W\} \\
\sqsubseteq & \langle(*)\text{ above}\rangle \\
 & \mu\, f
\end{array}
$$

□

4.3 Iteration

Definition 4.4. *Let S be a monotonic indexed predicate transformer. We define* strong iteration *on the ith exit, through least fixed point:*

$$S^{\omega_i} \mathrel{\widehat{=}} (\mu X \cdot S \mathbin{;_i} X \sqcap \mathsf{jump}\, i)$$

and we write S^{ω} as the abbreviation of $S^{\omega_{\mathsf{nrl}}}$.

Theorem 4.2. ✓ *Let S be a statement and let p_w with $w \in W$ be a collection of ranked predicates with ranks of type W. Assume that $(W, \leq)$ is a well-founded set, then*

$$(\forall\, w \in W.\, [\![p_w]\!]\, S\, [\![Q[i \leftarrow p_{<w}]]\!]) \Rightarrow [\![p]\!]\, S^{\omega_i}\, [\![Q[i \leftarrow p]]\!]$$

Proof. For any $w \in W$, according to the assumption, $[\![p_w]\!]\, S\, [\![Q[i \leftarrow p_{<w}]]\!]$. For any Q', when $Q[i \leftarrow p] \leq Q'$,

$$\begin{aligned}
& (\{p_w\} \,;\, \lfloor p, Q[i \leftarrow p] \rfloor)\, Q' \\
= \;& \langle \text{sequential composition, least specified statement} \rangle \\
& \{p_w\}\, p \\
= \;& \langle \text{assertion, ranked predicates} \rangle \\
& p_w \\
\leq \;& \langle \text{assmption, total correctness} \rangle \\
& S\, (Q[i \leftarrow p_{<w}]) \wedge p_w \\
\leq \;& \langle \text{sequential composition, monotonicity} \rangle \\
& ((S \mathbin{;_i} (\{p_{<w}\} \,;\, \lfloor p, Q[i \leftarrow p] \rfloor))\, Q') \wedge (Q'\, i) \\
= \;& \langle \text{definition of } \mathsf{jump}\, i, \text{ demonic choice} \rangle \\
& ((S \mathbin{;_i} (\{p_{<w}\} \,;\, \lfloor p, Q[i \leftarrow p] \rfloor)) \sqcap \mathsf{jump}\, i)\, Q'
\end{aligned}$$

and when $Q[i \leftarrow p] \not\leq Q'$

$$\begin{aligned}
& (\{p_w\} \,;\, \lfloor p, Q[i \leftarrow p] \rfloor)\, Q' \\
= \;& \langle \text{sequential composition, least specified statement} \rangle
\end{aligned}$$

false

$\leq$ $\langle$definition of $\mathsf{false}\rangle$

$((S \;;_i (\{p_{<w}\} \,;\, \lfloor p, Q[i \leftarrow p]\rfloor) \sqcap \mathsf{jump}\, i)\; Q'$

Then $\forall\, w \in W.\, \{p_w\} \,;\, \lfloor p, Q[i \leftarrow p]\rfloor \sqsubseteq (S \;;_i (\{p_{<w}\} \,;\, \lfloor p, Q[i \leftarrow p]\rfloor)) \sqcap \mathsf{jump}\, i$, and

$\forall\, w \in W.\, \{p_w\} \,;\, \lfloor p, Q[i \leftarrow p]\rfloor \sqsubseteq (S \;;_i (\{p_{<w}\} \,;\, \lfloor p, Q[i \leftarrow p]\rfloor)) \sqcap \mathsf{jump}\, i$

$=$ $\langle$function application$\rangle$

$\forall\, w \in W.\, \{p_w\} \,;\, \lfloor p, Q[i \leftarrow p]\rfloor \sqsubseteq (\lambda\, X.\, S \;;_i X \sqcap \mathsf{jump}\, i)\, (\{p_{<w}\} \,;\, \lfloor p, Q[i \leftarrow p]\rfloor)$

$\Rightarrow$ $\langle$Theorem 4.2$\rangle$

$\{p\} \,;\, \lfloor p, Q[i \leftarrow p]\rfloor \sqsubseteq \mu\, (\lambda\, X.\, S \;;_i X \sqcap \mathsf{jump}\, i)$

$\Rightarrow$ $\langle$least specified statement, iteration$\rangle$

$[\![p]\!]\, S^{\omega_i}\, [\![Q[i \leftarrow p]]\!]$

□

4.4 Discussion

Recursion allows reasoning about recursive programs. In Theorem 4.1, w is the size of an instance, and f is how the solutions to smaller instances can be combined into a solution of larger instance (the recursive function body). Theorem 4.2 will be used implicitly for proofs of theorems in Chapters 6 to 9.

Chapter 5

Normal Forms of Multi-Exit Statements

Normal forms facilitate construction of correct programs in a variety of aspects. Hoare et al. [1987] gives a normal form to show the completeness of algebraic laws of straight-line programs with bounded nondeterminism; it also implies the completeness of refinement laws, since refinement can be expressed in terms of equality and nondeterministic choice, *i.e.*, $S \sqsubseteq T \equiv S = S \sqcap T$. Hoare et al. [1993] expresses compilation of Dijkstra's guarded command programs as a reduction to a normal form. Von Wright [1994] proposes a normal form of programs with (unbounded) demonic and angelic nondeterminism; the normal form relies on a sequential composition of (abstract) angelic and demonic update statements. Abrial [1996] gives a normal form for programs with demonic nondeterminism to justify the specification constructs of the B method. Kozen [1997] proves that all while programs can be transformed into a normal form with only a single while-loop, using the Kleene algebra with tests and commutativity conditions. Back and von Wright [1998] study systematically normal forms for various classes of programs, including those with demonic and angelic nondeterminism. Ying

[2003] gives a normal form of probabilistic programs in the framework of [Back and von Wright, 1998]. Borba et al. [2004] proposes a normal form for object-oriented programs, which is illustrated with provably-correct refactoring.

The normal forms given in this chapter use sequential composition on the normal exit only. According to the symmetry of exits, a normal forms exists using sequential composition on any other exits.

5.1 Normal Form of Monotonic Indexed Predicate Transformers

Every multi-exit statement is a monotonic indexed predicate transformer by definition. We show that the converse also holds, that every monotonic indexed predicate transformer can be equivalently expressed as a multi-exit statement. It turns out that only normal angelic update, normal sequential composition, and demonic update are needed to express any monotonic indexed predicate transformer. The following theorem and its proof generalize those of [Back and von Wright, 1998] for single-exit statements.

Theorem 5.1. ✓ *Let S be a monotonic indexed predicate transformer. Then there exists a relation r and an indexed relation R such that $S = \{r\} \,;\, [R]$.*

Proof. Assume that $S : \mathcal{P}\Gamma_I \to \mathcal{P}\Sigma$ is a monotonic indexed predicate transformer. Define relation $r : \Sigma \to \mathcal{P}(\mathcal{P}\Gamma_I)$ and indexed relation $R : \mathcal{P}\Gamma_I \xrightarrow{\mathcal{I}} \mathcal{P}\Gamma$ as follows, where P is any indexed predicate of type $\mathcal{P}\Gamma_I$:

$$r\,\sigma\,P \mathrel{\widehat{=}} S\,P\,\sigma$$

$$R\,i\,P\,\gamma \mathrel{\widehat{=}} P\,i\,\gamma$$

Then, for arbitrary indexed predicate Q and arbitrary state σ_0 we have:

$$
\begin{aligned}
& (\{r\} \,;\, [R])\; Q\; \sigma_0 \\
\equiv \quad & \langle \text{definition of sequential composition} \rangle \\
& \{r\}\; (Q[\mathsf{nrl} \leftarrow [R]\; Q])\; \sigma_0 \\
\equiv \quad & \langle \text{definition of demonic update} \rangle \\
& \{r\}\; (Q[\mathsf{nrl} \leftarrow (\lambda\, \sigma.\, \forall\, i\; \gamma.\; R\; i\; \sigma\; \gamma \Rightarrow Q\; i\; \gamma)])\; \sigma_0 \\
\equiv \quad & \langle \text{Theorem 3.5, definition of } R \rangle \\
& (\exists\, P.\; r\; \sigma_0\; P \wedge (\forall\, i\; \gamma.\; P\; i\; \gamma \Rightarrow Q\; i\; \gamma)) \\
\equiv \quad & \langle \text{definition of } r, \text{ definition of } \leq \rangle \\
& (\exists\, P.\; S\; P\; \sigma_0 \wedge P \leq Q) \\
\equiv \quad & \langle (**) \rangle \\
& S\; Q\; \sigma_0
\end{aligned}
$$

The step (**) is shown by mutual implication:

$$
\begin{aligned}
& (\exists\, P.\; S\; P\; \sigma_0 \wedge P \leq Q) \\
\Rightarrow \quad & \langle S \text{ is monotonic, context says } P \leq Q \rangle \\
& (\exists\, P.\; S\; Q\; \sigma_0 \wedge P \leq Q) \\
\Rightarrow \quad & \langle \text{weakening} \rangle \\
& (\exists\, P.\; S\; Q\; \sigma_0) \\
\Rightarrow \quad & \langle \text{quantifier rule} \rangle \\
& S\; Q\; \sigma_0
\end{aligned}
$$

For the reverse implication we have:

$$
\begin{aligned}
& (\exists\, P.\; S\; P\; \sigma_0 \wedge P \leq Q) \\
\Leftarrow \quad & \langle \text{witness } P = Q \rangle \\
& S\; Q\; \sigma_0 \wedge Q \leq Q \\
\equiv \quad & \langle \text{reflexitivity} \rangle \\
& S\; Q\; \sigma_0
\end{aligned}
$$

Together it is shown that $S = \{r\} ; [R]$. □

As a consequence, any monotonic indexed predicate transformer can be expressed as a statement. For example, the least statement $\lfloor p, Q \rfloor$ specified by p and Q can be written in normal form as

$$\lfloor p, Q \rfloor = \{p\} ; [(\lambda\, i\, \sigma\, \gamma.\, p\, \sigma \wedge Q\, i\, \gamma)]$$

The normal form $\{r\} ; [R]$ can be interpreted as a simple two-player game [Back and von Wright, 1998]: given an initial state, first the angel chooses an intermediate exit and an intermediate state. Then, for any possible intermediate exit and intermediate state, the demon choses the final exit and final state. An interesting observation is how the intermediate state space is constructed in the proof. The precondition is of type $\mathcal{P}\Sigma$, the indexed postcondition is of type $\mathcal{P}\Gamma_I$, and the intermediate condition is of type $\mathcal{P}(\mathcal{P}\Gamma_I)$. That is, an intermediate state is a set of final states. Given initial state σ, the angel first picks an indexed predicate P of type $\mathcal{P}\Gamma_I$ such that $S\, P\, \sigma$ holds and then the demon picks at each exit i a final state γ such that $P\, i\, \gamma$ holds.

To illustrate this with a concrete example, consider statement S with two exits, a normal and an exceptional exit, $I = \{\mathsf{nrl}, \mathsf{exc}\}$, and we write (q_1, q_2) as an abbreviation of $\{\mathsf{nrl} \mapsto q_1, \mathsf{exc} \mapsto q_2\}$. Recalling that $\mathsf{skip} = \mathsf{jump}\,\mathsf{nrl}$, we define $\mathsf{raise} = \mathsf{jump}\,\mathsf{exc}$ and $S = \mathsf{skip} \sqcup \mathsf{raise}$. The weakest precondition of S is $S\,(q_1, q_2) = (q_1 \vee q_2)$. The normal form of S is $\{r\}; [R]$ such that $r\, p\, (q_1, q_2) \equiv p = (q_1 \vee q_2)$ and $R = (r_1, r_2)$ where $r_1\,(q_1, q_2)\,\gamma_1 \equiv q_1\,\gamma_1$ and $r_2\,(q_1, q_2)\,\gamma_2 \equiv q_2\,\gamma_2$. From any precondition p, the normal angelic update $\{r\} = \{(r, \bot)\}$ always exits normally. However, intuitively it can choose to go to the intermediate state (p, false), which represents either exiting normally with postcondition p or exiting exceptionally with false, it can also choose to go to the intermediate state (false, p), which represents either exiting normally with postcondition false or exiting exceptionally with p, or it can choose to establish any

(q_1, q_2) where $p = (q_1 \vee q_2)$. In case of (p, false), the next statement, $[(r_1, r_2)]$, forces r_1 to be chosen (r_2 is $\perp$ due to false, according to the definition) and S exits normally with p, as skip does. In case of (false, p), the next statement, $[(r_1, r_2)]$, forces r_2 to be chosen (r_1 is $\perp$ due to false, according to the definition) and S exits exceptionally with p, as raise does.

As a second example, consider statement S defined by $S = \mathsf{skip} \sqcap \mathsf{raise}$. The weakest precondition of S is $S\,(q_1, q_2) = (q_1 \wedge q_2)$. The normal form of S is $\{r\}; [R]$ such that $r\,p\,(q_1, q_2) \equiv p = (q_1 \wedge q_2)$, and $R = (r_1, r_2)$ where $r_1\,(q_1, q_2)\,\gamma_1 \equiv q_1\,\gamma_1$, $r_2\,(q_1, q_2)\,\gamma_2 \equiv q_2\,\gamma_2$. From any precondition p, the normal angelic update $\{r\} = \{(r, \perp)\}$ exits normally. It can choose to go in the state (p, p) or it can choose to establish (q_1, q_2) where $p = (q_1 \wedge q_2)$. In case of (p, p) in the intermediate state, the next statement, $[(r_1, r_2)]$ can choose r_1 and exit normally with p, as skip does, or choose r_2 and exit exceptionally with p, as raise does.

In principle, sequential composition at exit other than nrl, angelic update, demonic choice, and angelic choice are not necessary to express an arbitrary monotonic indexed predicate transformer. Still, we consider them for symmetry reasons and to be able to define concrete programming constructs in terms of those.

5.2 Normal Form of Conjunctive Indexed Predicate Transformers

Statements stop, $\mathsf{jump}\,i$, assumption $[P]$, and demonic update $[R]$ are universally conjunctive, and sequential composition $;_i$ and demonic choice $\sqcap$ preserve universal conjunctivity. We show that in turn every universally conjunctive indexed predicate transformer can be expressed by a statement consisting only of normal demonic

update, with one relation for each exit.

Lemma 5.1. ✓ *If S is a universally conjunctive indexed predicate transformer, then for all indexed predicates Q and all states σ:*

$$S\ Q\ \sigma \equiv \wedge_s\{P \mid P \wedge S\ P\ \sigma\} \leq Q$$

Proof. When $S\ Q\ \sigma = \mathit{True}$:

$$\begin{aligned}
&\wedge_s\ \{P \mid P \wedge S\ P\ \sigma\} \leq Q \\
\equiv\ &\langle \text{splitting } \wedge_s \rangle \\
&Q \wedge (\wedge_s\{P \mid P \wedge S\ P\ \sigma \wedge P \neq Q\}) \leq Q \\
\equiv\ &\langle \wedge\text{-elimination} \rangle \\
&\mathit{True}
\end{aligned}$$

When $S\ Q\ \sigma = \mathit{False}$:

$$\begin{aligned}
&(\wedge_s\{P \mid P \wedge S\ P\ \sigma\} \leq Q \\
\Rightarrow\ &\langle S \text{ is monotonic, Theorem 3.13} \rangle \\
&S\ (\wedge_s\{P \mid P \wedge S\ P\ \sigma\} \leq S\ Q \\
\equiv\ &\langle \text{definition of conjunctivity} \rangle \\
&(\wedge_s\{S\ P \mid P \wedge S\ P\ \sigma\} \leq S\ Q \\
\Rightarrow\ &\langle \text{definition of } \leq \rangle \\
&(\wedge_s\{S\ P \mid P \wedge S\ P\ \sigma\}\ \sigma) \Rightarrow S\ Q\ \sigma \\
\equiv\ &\langle \text{definition of } \wedge_s \rangle \\
&(\wedge_s\{S\ P\ \sigma \mid P \wedge S\ P\ \sigma\}) \Rightarrow S\ Q\ \sigma \\
\equiv\ &\langle \text{definition of } \wedge_s, \text{ assumption } S\ Q\ \sigma = \mathsf{false} \rangle \\
&\mathsf{true} \Rightarrow \mathsf{false} \\
\equiv\ &\langle \text{lattice property} \rangle \\
&\mathit{False}
\end{aligned}$$

Thus we know that $S\ Q\ \sigma \equiv \{P \mid P \wedge S\ P\ \sigma\} \leq Q$ □

Theorem 5.2. ✓ *Let S be an universally conjunctive indexed predicate transformer. There exists a unique indexed relation R such that $S = [R]$.*

Proof. Let I be the exit indices of S. For arbitrary $i \in I$ we define:

$$R\, i\, \sigma = \wedge_s \{P\, i \mid P \wedge S\, P\, \sigma\}$$

Then we calculate for any indexed predicate Q:

$$\begin{array}{ll} & [R]\, Q\, \sigma \\ \equiv & \langle \text{definition of } R, \text{ definition of demonic update} \rangle \\ & \forall\, i.\, (\forall\, \gamma.\, (\wedge_s \{P\, i \mid P \wedge S\, P\, \sigma\})\, \gamma \Rightarrow Q\, i\, \gamma) \\ \equiv & \langle \text{definition of } \leq \text{ on predicates} \rangle \\ & \forall\, i.\, \wedge_s \{P\, i \mid P \wedge S\, P\, \sigma\} \leq Q\, i \\ \equiv & \langle \text{definition of } \leq \text{ on indexed predicates} \rangle \\ & \wedge_s \{P \mid P \wedge S\, P\, \sigma\} \leq Q \\ \equiv & \langle \text{Lemma 5.1} \rangle \\ & S\, Q\, \sigma \end{array}$$

This shows that $[R] = S$. For uniqueness we have:

$$\begin{array}{ll} & [R] = [R'] \\ \equiv & \langle \text{antisymmetry} \rangle \\ & [R] \sqsubseteq [R'] \wedge [R] \sqsupseteq [R'] \\ \equiv & \langle \text{definition of demonic update, definition of } \leq, \text{ definition of } \sqsubseteq \rangle \\ & \forall\, i.\, R\, i \supseteq R'\, i \wedge R\, i \sqsubseteq R'\, i \\ \equiv & \langle \text{antisymmetry} \rangle \\ & \forall\, i.\, R\, i = R'\, i \\ \equiv & \langle \text{equality of tuples} \rangle \\ & R = R' \end{array}$$

□

Lemma 5.2. ✓ *If S is conjunctive, then $[tr\ S]\ ;\ S$ is universally conjunctive.*

Proof. For arbitrary predicate S, we show that $[tr\ S]\ ;\ S$ is terminating:

$$
\begin{array}{ll}
& tr\ ([tr\ S]\ ;\ S) \\
= & \langle \text{definition of } tr \rangle \\
& ([S\ \overline{\mathsf{true}}]\ ;\ S)\ \overline{\mathsf{true}} \\
= & \langle \text{definition of sequential composition} \rangle \\
& [S\ \overline{\mathsf{true}}]\ (\overline{\mathsf{true}}[\mathsf{nrl} \leftarrow S\ \overline{\mathsf{true}}]) \\
= & \langle \text{definition of assumption} \rangle \\
& S\ \overline{\mathsf{true}} \Rightarrow (\overline{\mathsf{true}}[\mathsf{nrl} \leftarrow S\ \overline{\mathsf{true}}])\ \mathsf{nrl} \\
= & \langle \text{simplification of function update} \rangle \\
& S\ \overline{\mathsf{true}} \Rightarrow S\ \overline{\mathsf{true}} \\
= & \langle \text{lattice property} \rangle \\
& \mathsf{true}
\end{array}
$$

Thus with Theorem 3.11, $[tr\ S]\ ;\ S$ is universally conjunctive by definition. □

Theorem 5.3. ✓ *Let S be an arbitrary positively conjunctive indexed predicate transformer. Then there exists a unique predicate p and a unique indexed relation R such that $S = \{p\}\ ;\ [R]$.*

Proof. It is easily seen that we must choose $p = tr\ S$:

$$
\begin{array}{ll}
& S = \{p\}\ ;\ [R] \\
\Rightarrow & \langle \text{congruence} \rangle \\
& tr\ S = tr\ (\{p\}\ ;\ [R]) \\
\equiv & \langle \text{from definitions} \rangle \\
& tr\ S = (p \wedge tr\ [R]) \\
\equiv & \langle \text{from definitions} \rangle \\
& tr\ S = p
\end{array}
$$

By Lemma 5.2 we have that $\{p\}\,;[R]$ is universally conjunctive. Then we can conclude by Theorem 5.2 that some indexed relation R exists such that $[tr\ S]\,;S = [R]$, and by the proof of that theorem, $R\,i\,\sigma = \wedge_s\{P\ i \mid P \wedge S\ P\ \sigma\}$. Then we have for arbitrary indexed predicate Q:

$$
\begin{array}{ll}
 & (\{tr\ S\}\ ;\ S)\ Q \\
\equiv & \langle \text{from definitions} \rangle \\
 & (\{tr\ S\}\ ;\ [tr\ S]\ ;\ S)\ Q \\
\equiv & \langle \text{definition of ;, assertion, assumption} \rangle \\
 & tr\ S \wedge (tr\ S \Rightarrow S\ Q) \\
\equiv & \langle \text{lattice property} \rangle \\
 & tr\ S \wedge S\ Q \\
\equiv & \langle \text{definition of } tr\ S \rangle \\
 & S\ \overline{\mathsf{true}} \wedge S\ Q \\
\equiv & \langle S \text{ is monotonic by Theorem 3.13} \rangle \\
 & S\ Q
\end{array}
$$

Thus $S = \{p\}\,;[R]$ for p and R as above. □

5.3 Discussion

The normal forms not only give a uniform specification for certain categories of statements, but also show the equivalence between all statements and monotonic indexed predicate transformers by mutual inclusion (Theorems 3.10 and 5.1). From the next chapter on these two terms will be used interchangeably.

Chapter 6

Fail-Safe Correctness and Fail-Safe Refinement

In this section we study programs that use exception handling with termination model semantics, as followed by C++ and Java [ISO, 2012; Gosling et al., 2013]. We model statements with termination model semantics as double-exit predicate transformers with a normal exit and a single exceptional exit, *i.e.*, the exit indices being $I = \{\mathsf{nrl}, \mathsf{exc}\}$. Correspondingly, we write (q_1, q_2) as an abbreviation of $\{\mathsf{nrl} \mapsto q_1, \mathsf{exc} \mapsto q_2\}$. Thus, the meaning of total correctness becomes

$[\![p]\!]\, S\, [\![q_1, q_2]\!] \equiv$ Under precondition p, statement S terminates and

– on normal termination q_1 holds finally,

– on exceptional termination q_2 holds finally.

In addition to the statements defined in Chapter 3, we define a few more statements that are commonly used in programming languages. We define ;; as the abbreviation of $;_{\mathsf{exc}}$, the *exceptional composition*, and raise as the abbreviation of $\mathsf{jump}\,\mathsf{exc}$, which terminates exceptionally without changing the state. The common $\mathsf{try}\, S\, \mathsf{catch}\, T$

statement with *body* S and *handler* T is just a different notation for exceptional composition. The statement $\mathsf{try}\ S\ \mathsf{catch}\ T\ \mathsf{finally}\ U$ with *finalization* U can be defined in terms of sequential and exceptional composition. The finalization U is executed either after S succeeds, after S fails and T succeeds, or after S fails and T fails, in which case the whole statement fails whether U succeeds or fails; see Figure 6.1.

$$\mathsf{try}\ S\ \mathsf{catch}\ T \mathrel{\widehat{=}} S\ ;;\ T$$

$$\mathsf{try}\ S\ \mathsf{catch}\ T\ \mathsf{finally}\ U \mathrel{\widehat{=}} (S\ ;;\ (T\ ;;\ (U\ ;\ \mathsf{raise})))\ ;\ U$$

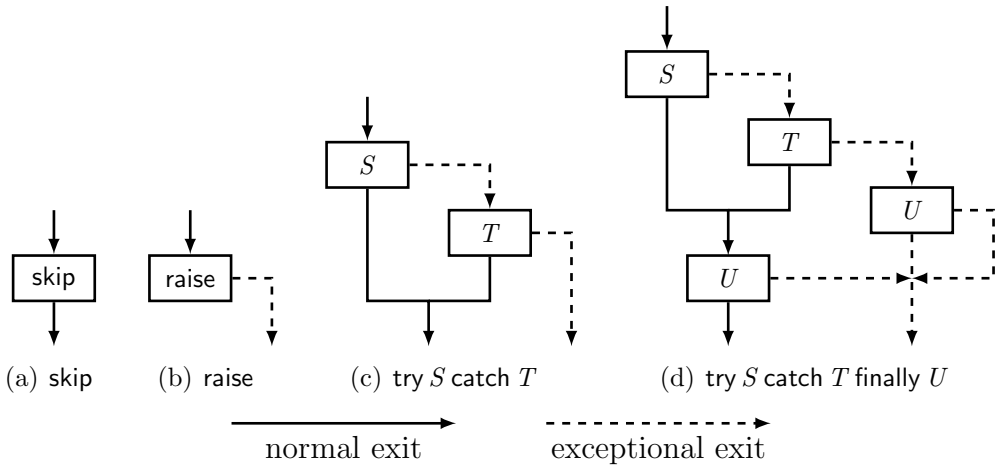


Figure 6.1: Symmetry on Two Exits

The assignment statement $x := E$ is defined in terms of an update statement that affects only component x of the state space. For this we assume that the state is a tuple and variables select elements of the tuple.

With x being a variable and e being a relation, the *relational assignment* $x := e$ modifies component x of the state space to be e and leaves all other components of the state space unchanged; the initial and final state spaces are the same. The *nondeterministic relational assignment* $x :\in \mathfrak{e}$ modifies component x of the state space

to be any element of the set $\mathfrak{e}$. Provided that the state space consists of variables x, y we define:

$$x := e \mathrel{\widehat{=}} \lambda\,(x, y).\, \lambda\,(x', y').\, x' = e\,(x, y) \wedge y' = y$$

$$x :\in \mathfrak{e} \mathrel{\widehat{=}} \lambda\,(x, y).\, \lambda\,(x', y').\, x' \in \mathfrak{e}\,(x, y) \wedge y' = y$$

The *(deterministic) assignment* $x := E$ fails if program expression E is undefined, otherwise it succeeds and assigns the value of E to x. The *nondeterministic assignment* $x :\in \mathfrak{E}$ fails if $\mathfrak{E}$ is undefined, otherwise it succeeds and assigns any element of the set $\mathfrak{E}$ to x, the choice being demonic:

$$x := E \mathrel{\widehat{=}} \{\Delta E, \neg\Delta E\}\,;\, [x := \text{`}E\text{'}]$$

$$x :\in \mathfrak{E} \mathrel{\widehat{=}} \{\Delta \mathfrak{E}, \neg\Delta \mathfrak{E}\}\,;\, [x :\in \text{`}\mathfrak{E}\text{'}]$$

The *check statement* $\mathsf{check}\, B$ terminates normally if Boolean expression B is defined and evaluates to *True*, and terminates exceptionally if B is undefined or evaluates to *False*.

$$\mathsf{check}\, B \mathrel{\widehat{=}} \{\Delta B \wedge \text{`}B\text{'}, \neg\Delta B \vee \neg\text{`}B\text{'}\}$$

The *conditional statement* $\mathsf{if}\, B\, \mathsf{then}\, S\, \mathsf{else}\, T$ fails if B is undefined, otherwise continues with either S or T, depending on the value of B:

$$\mathsf{if}\, B\, \mathsf{then}\, S\, \mathsf{else}\, T \mathrel{\widehat{=}} \{\Delta B, \neg\Delta B\}\,;\, ((\text{[`}B\text{']}\,;\, S) \sqcap (\text{[}\neg\text{`}B\text{']}\,;\, T))$$

$$\mathsf{if}\, B\, \mathsf{then}\, S \mathrel{\widehat{=}} \mathsf{if}\, B\, \mathsf{then}\, S\, \mathsf{else}\, \mathsf{skip}$$

The *while loop* $\mathsf{while}\, B\, \mathsf{do}\, S$, in which B is a Boolean expression and S is a statement, is defined as the least fixed point of $\lambda\, X.\ \mathsf{if}\, B\, \mathsf{then}(S;X)$ with respect to the refinement ordering, *i.e.*,

$$\mathsf{while}\, B\, \mathsf{do}\, S \mathrel{\widehat{=}} (\mu X \cdot \mathsf{if}\, B\, \mathsf{then}(S\,;\,X))$$

In general we have the following *sub-conjunctivity* property:

$$S\,(q_1, q_2) \wedge S\,(q_1', q_2') \Leftarrow S\,(q_1 \wedge q_1', q_2 \wedge q_2')$$

As a direct consequence of sub-conjuctivity we get following *sub-separation* property:

$$S\,(q_1, \mathsf{true}) \wedge S\,(\mathsf{true}, q_2) \Leftarrow S\,(q_1, q_2)$$

Unfortunately the direction of the implication does not allow the reasoning to be separated in general: we would like from $S\,(q_1, \mathsf{true})$ (succeeding with q_1 or failing in any state) and $S\,(\mathsf{true}, q_2)$ (succeeding in any state or failing with q_2) to deduce $S\,(q_1, q_2)$. For any universal conjunctive statement S *separation* holds:

$$S\,(q_1, \mathsf{true}) \wedge S\,(\mathsf{true}, q_2) = S\,(q_1, q_2)$$

Statements abort, stop, skip, raise, assumption $[q_1, q_2]$, and demonic update $[r_1, r_2]$ are positively conjunctive. Sequential composition, exceptional composition, and demonic choice preserve positive conjunctivity. The assertion $\{g_1, g_2\}$ is conjunctive only if q_1 excludes q_2, *i.e.*, $q_1 \wedge q_2 = \mathsf{false}$. Angelic choice and angelic update are in general not conjunctive either. Since separation is a desirable property, in this section we mainly consider conjunctive statements; all of the statements considered in the following theorems are conjunctive.

6.1 Total Correctness

Suppose the state space of p is a tuple with x as an element, *e.g.*, $(x, \ldots)$, we write $p[x \backslash v]$ for the predicate resulted from replacing free variable x by the value of expression v whenever evaluating predicate p, *i.e.*, $\lambda\,(x, \ldots).\, p\,(v\,(x, \ldots), \ldots)$. The next theorem states the basic properties of total correctness, most part of it as the instantiation of Theorem 3.18 for double-exit programs.

Theorem 6.1. ✓ *Let p, q_1, q_2, g_1, g_2 be predicates, r_1, r_2 be relations, B, E, $\mathfrak{E}$ be a program expressions, and S, T be statements:*

$$[\![p]\!]\ \mathsf{abort}\ [\![q_1, q_2]\!] \equiv p = \mathsf{false}$$

$$[\![p]\!]\ \mathsf{stop}\ [\![q_1, q_2]\!] \equiv \mathit{True}$$

$$
\begin{aligned}
[\![p]\!]\ \mathsf{skip}\ [\![q_1, q_2]\!] &\equiv p \Rightarrow q_1 \\
[\![p]\!]\ \mathsf{raise}\ [\![q_1, q_2]\!] &\equiv p \Rightarrow q_2 \\
[\![p]\!]\ [g_1, g_2]\ [\![q_1, q_2]\!] &\equiv p \leq (g_1 \Rightarrow q_1) \wedge (g_2 \Rightarrow q_2) \\
[\![p]\!]\ [r_1, r_2]\ [\![q_1, q_2]\!] &\equiv (\forall\, \sigma.\, p\, \sigma \Rightarrow (r_1\, \sigma \leq q_1) \wedge (r_2\, \sigma \leq q_2)) \\
[\![p]\!]\ x := E\ [\![q_1, q_2]\!] &\equiv (\Delta E \wedge p \leq q_1[x \backslash \text{`}E\text{'}]) \wedge (\neg \Delta E \Rightarrow q_2) \\
[\![p]\!]\ x :\in \mathfrak{E}\ [\![q_1, q_2]\!] &\equiv (\Delta \mathfrak{E} \wedge p \Rightarrow \forall\, x' \in \text{`}\mathfrak{E}\text{'}.\, q_1[x \backslash x']) \wedge (\neg \Delta \mathfrak{E} \Rightarrow q_2) \\
[\![p]\!]\ S\ ;\ T\ [\![q_1, q_2]\!] &\equiv \exists\, h.\, [\![p]\!]\ S\ [\![h, q_2]\!] \wedge [\![h]\!]\ T\ [\![q_1, q_2]\!] \\
[\![p]\!]\ S\ ;;\ T\ [\![q_1, q_2]\!] &\equiv \exists\, h.\, [\![p]\!]\ S\ [\![q_1, h]\!] \wedge [\![h]\!]\ T\ [\![q_1, q_2]\!] \\
[\![p]\!]\ S \sqcap T\ [\![q_1, q_2]\!] &\equiv [\![p]\!]\ S\ [\![q_1, q_2]\!] \wedge [\![p]\!]\ T\ [\![q_1, q_2]\!] \\
[\![p]\!]\ \mathsf{check}\ B\ [\![q_1, q_2]\!] &\equiv (\Delta B \wedge \text{`}B\text{'} \wedge p \Rightarrow q_1) \wedge \\
&\qquad (\Delta B \wedge \neg \text{`}B\text{'} \wedge p \Rightarrow q_2) \wedge \\
&\qquad (\neg \Delta B \wedge p \Rightarrow q_2) \\
[\![p]\!]\ \mathsf{if}\ B\ \mathsf{then}\ S\ \mathsf{else}\ T\ [\![q_1, q_2]\!] &\equiv [\![\Delta B \wedge \text{`}B\text{'} \wedge p]\!]\ S\ [\![q_1, q_2]\!] \wedge \\
&\qquad [\![\Delta B \wedge \neg \text{`}B\text{'} \wedge p]\!]\ T\ [\![q_1, q_2]\!] \wedge \\
&\qquad (\neg \Delta B \wedge p \Rightarrow q_2)
\end{aligned}
$$

We immediately get following consequence rule for any statement S:

$$(p' \Rightarrow p) \wedge [\![p]\!]\ S\ [\![q_1, q_2]\!] \wedge (q_1 \Rightarrow q_1') \wedge (q_2 \Rightarrow q_2') \Rightarrow [\![p']\!]\ S\ [\![q_1', q_2']\!]$$

For conjunctive statement S we have also:

$$[\![p]\!]\ S\ [\![q_1, q_2]\!] \wedge [\![p']\!]\ S\ [\![q_1', q_2']\!] \Rightarrow [\![p \wedge p']\!]\ S\ [\![q_1 \wedge q_1', q_2 \wedge q_2']\!]$$

Separation arises as a special case:

$$[\![p]\!]\ S\ [\![q_1, \mathsf{true}]\!] \wedge [\![p]\!]\ S\ [\![\mathsf{true}, q_2]\!] \Rightarrow [\![p]\!]\ S\ [\![q_1, q_2]\!]$$

6.2 Domains

For statements with two exits, the termination domain includes the *normal termination domain*, written as $nr\,S$, and the *exceptional termination domain*, written as $ex\,S$:

$$tr\,S \mathrel{\widehat{=}} S\,(\mathsf{true}, \mathsf{true})$$
$$nr\,S \mathrel{\widehat{=}} tr_{\mathsf{nrl}}\,S = S\,(\mathsf{true}, \mathsf{false})$$
$$ex\,S \mathrel{\widehat{=}} tr_{\mathsf{exc}}\,S = S\,(\mathsf{false}, \mathsf{true})$$
$$en\,S \mathrel{\widehat{=}} \neg S\,(\mathsf{false}, \mathsf{false})$$

As a corollary of sub-conjunctivity, we have $nr\,S \wedge ex\,S \Leftarrow \neg en\,S$ for any statement S. This can be strengthen to $nr\,S \wedge ex\,S \equiv \neg en\,S$ if S is conjunctive. The next theorem summarizes the basic properties of the domain operations, part of which being an instantiation of Theorem 3.17.

Theorem 6.2. ✓ *Let S, T be double-exit predicate transformers, q_1, q_2 be predicates, r_1, r_2 be relations, x be a variable, and E, B be program expressions, we have the following theorem as instantiation of Theorem 3.17 for double-exit statements:*

$$tr\ \mathsf{abort} = \mathsf{false} \qquad tr\ \mathsf{stop} = \mathsf{true} \qquad tr\ \mathsf{skip} = \mathsf{true} \qquad tr\ \mathsf{raise} = \mathsf{true}$$
$$nr\ \mathsf{abort} = \mathsf{false} \qquad nr\ \mathsf{stop} = \mathsf{true} \qquad nr\ \mathsf{skip} = \mathsf{true} \qquad nr\ \mathsf{raise} = \mathsf{false}$$
$$ex\ \mathsf{abort} = \mathsf{false} \qquad ex\ \mathsf{stop} = \mathsf{true} \qquad ex\ \mathsf{skip} = \mathsf{false} \qquad ex\ \mathsf{raise} = \mathsf{true}$$
$$en\ \mathsf{abort} = \mathsf{true} \qquad en\ \mathsf{stop} = \mathsf{false} \qquad en\ \mathsf{skip} = \mathsf{true} \qquad en\ \mathsf{raise} = \mathsf{true}$$
$$tr\,[g_1, g_2] = \mathsf{true} \qquad nr\,[g_1, g_2] = \neg g_2$$
$$ex\,[g_1, g_2] = \neg g_1 \qquad en\,[g_1, g_2] = g_1 \vee g_2$$
$$tr\,(x := E) = \mathsf{true} \qquad tr\,(x :\in \mathfrak{E}) = \mathsf{true}$$
$$nr\,(x := E) = \Delta E \qquad nr\,(x :\in \mathfrak{E}) = \Delta\mathfrak{E}$$
$$ex\,(x := E) = \neg\Delta E \qquad ex\,(x :\in \mathfrak{E}) = \Delta\mathfrak{E} \Rightarrow (\lambda\,\sigma.\,`\mathfrak{E}'\,\sigma = \emptyset)$$
$$en\,(x := E) = \mathsf{true} \qquad en\,(x :\in \mathfrak{E}) = \Delta\mathfrak{E} \Rightarrow (\lambda\,\sigma.\,`\mathfrak{E}'\,\sigma \neq \emptyset)$$

$$tr\,(S\ ;\ T) \Rightarrow tr\ S \qquad tr\,(S\ ;;\ T) \Rightarrow tr\ S$$

$$nr\,(S\ ;\ T) \Rightarrow nr\ S \qquad nr\,(S\ ;;\ T) \Leftarrow nr\ S$$

$$ex\,(S\ ;\ T) \Leftarrow ex\ S \qquad ex\,(S\ ;;\ T) \Rightarrow ex\ S$$

$$en\,(S\ ;\ T) \Rightarrow en\ S \qquad en\,(S\ ;;\ T) \Rightarrow en\ S$$

$$tr\,(S \sqcap T) = tr\ S \wedge tr\ T \qquad nr\,(S \sqcap T) = nr\ S \wedge nr\ T$$

$$ex\,(S \sqcap T) = ex\ S \wedge ex\ T \qquad en\,(S \sqcap T) = en\ S \vee en\ T$$

$$tr\,[r_1, r_2] = \mathsf{true} \qquad nr\,[r_1, r_2] = (\lambda\,\sigma.\,(\forall\,\gamma_2.\,\neg r_2\,\sigma\,\gamma_2))$$

$$ex\,[r_1, r_2] = (\lambda\,\sigma.\,(\forall\,\gamma_1.\,\neg r_1\,\sigma\,\gamma_1)) \qquad en\,[r_1, r_2] = (\lambda\,\sigma.\,(\exists\,\gamma_1.\,r_1\,\sigma\,\gamma_1) \vee (\exists\,\gamma_2.\,r_2\,\sigma\,\gamma_2))$$

$$tr\,(\mathsf{check}\ B) = \mathsf{true} \qquad nr\,(\mathsf{check}\ B) = \Delta B \wedge \text{`}B\text{'}$$

$$ex\,(\mathsf{check}\ B) = \neg\Delta B \vee \neg\text{`}B\text{'} \qquad en\,(\mathsf{check}\ B) = \mathsf{true}$$

$$tr\,(\mathsf{if}\ B\ \mathsf{then}\ S\ \mathsf{else}\ T) = (\Delta B \wedge \text{`}B\text{'} \Rightarrow tr\ S) \wedge (\Delta B \wedge \neg\text{`}B\text{'} \Rightarrow tr\ T)$$

$$nr\,(\mathsf{if}\ B\ \mathsf{then}\ S\ \mathsf{else}\ T) = \Delta B \wedge (\text{`}B\text{'} \Rightarrow nr\ S) \wedge (\neg\text{`}B\text{'} \Rightarrow nr\ T)$$

$$ex\,(\mathsf{if}\ B\ \mathsf{then}\ S\ \mathsf{else}\ T) = (\Delta B \wedge \text{`}B\text{'} \Rightarrow ex\ S) \wedge (\Delta B \wedge \neg\text{`}B\text{'} \Rightarrow ex\ T)$$

$$en\,(\mathsf{if}\ B\ \mathsf{then}\ S\ \mathsf{else}\ T) = \Delta B \Rightarrow (\text{`}B\text{'} \wedge en\ S \vee \neg\text{`}B\text{'} \Rightarrow en\ T)$$

6.3 Fail-Safe Correctness

The notion of total correctness assumes that any possible failure has to be anticipated in the specification; the outcome in case of failure is specified by the exceptional postcondition, an implementation may not fail in any other way: a rather optimistic point of view. We propose a new notion of *fail-safe correctness*, which weakens the notion of total correctness by allowing also "true" exceptions. For orderly continuation after an unanticipated exception, the restriction is that in that case, the state must not change. Partial correctness does not guarantee termination [Back and von Wright, 1998]. In Sekerinski and Zhang [2011] we use "partial correctness" instead of "fail-safe

correctness" for lack of a better name.

For example, database transactions can be explained in terms of fail-safe correctness: either a transaction succeeds, establishing the desired postcondition, or it fails and the original state is restored. Another example for fail-safe correctness is the recovery block for software fault tolerance [Horning et al., 1974; Randell, 1975]: a list of alternative implementations is attempted in given order. If one alternative fails, the original state is restored and the next attempted, until either one succeeds or all fail. The design of class methods in robust object-oriented programs also follows the principles of fail-safe correctness: if a method fails, it must at least establish the object invariant as the alternative postcondition, such that program execution can continue and methods of the object may still be called.

We introduce following notation:

$\langle\!|p|\!\rangle\ S\ \langle\!|\, q_1, q_2 |\!\rangle \equiv$ Under precondition p, statement S terminates and

– on normal termination q_1 holds finally,

– on exceptional termination p or q_2 holds finally.

Both total correctness and fail-safe correctness guarantee termination when the precondition holds. If $\langle\!|p|\!\rangle\ S\ \langle\!|\, q_1, \mathsf{false} |\!\rangle$ holds, then statement S does not modify the state when terminating exceptionally, and we write this more concisely as $\langle\!|p|\!\rangle\ S\ \langle\!|\, q_1 |\!\rangle$. Let p, q_1, q_2 be predicates:

$$\langle\!|p|\!\rangle\ S\ \langle\!|\, q_1, q_2 |\!\rangle \mathrel{\widehat{=}} p \Rightarrow S\,(q_1, p \vee q_2)$$

$$\langle\!|p|\!\rangle\ S\ \langle\!|\, q_1 |\!\rangle \mathrel{\widehat{=}} p \Rightarrow S\,(q_1, p)$$

Total correctness implies fail-safe correctness, but not vice versa. The very definition of fail-safe correctness breaks the duality between normal and exceptional postconditions that total correctness enjoys. This leads to some curious consequences that we will explore.

Theorem 6.3. ✓ *Let p, q_1, q_2, g_1, g_2 be predicates, r_1, r_2 be relations, B, E, $\mathfrak{E}$ be program expressions, and S, T be statements:*

$$\begin{aligned}
\langle\!|p|\!\rangle\ \mathsf{abort}\ \langle\!|q_1, q_2|\!\rangle &\equiv p = \mathsf{false} \\
\langle\!|p|\!\rangle\ \mathsf{stop}\ \langle\!|q_1, q_2|\!\rangle &\equiv \mathit{True} \\
\langle\!|p|\!\rangle\ \mathsf{skip}\ \langle\!|q_1, q_2|\!\rangle &\equiv p \Rightarrow q_1 \\
\langle\!|p|\!\rangle\ \mathsf{raise}\ \langle\!|q_1, q_2|\!\rangle &\equiv \mathit{True} \\
\langle\!|p|\!\rangle\ [g_1, g_2]\ \langle\!|q_1, q_2|\!\rangle &\equiv (p \wedge g_1 \leq q_1) \\
\langle\!|p|\!\rangle\ [r_1, r_2]\ \langle\!|q_1, q_2|\!\rangle &\equiv (\forall\, \sigma.\, p\, \sigma \Rightarrow (r_1\, \sigma \leq q_1) \wedge (r_2\, \sigma \leq p \vee q_2)) \\
\langle\!|p|\!\rangle\ x := E\ \langle\!|q_1, q_2|\!\rangle &\equiv \Delta E \wedge p \Rightarrow q_1[x \backslash \text{`}E\text{'}] \\
\langle\!|p|\!\rangle\ x :\in \mathfrak{E}\ \langle\!|q_1, q_2|\!\rangle &\equiv \Delta \mathfrak{E} \wedge p \Rightarrow \forall\, x' \in \text{`}\mathfrak{E}\text{'}.\, q_1[x \backslash x'] \\
\langle\!|p|\!\rangle\ S\ ;\ T\ \langle\!|q_1, q_2|\!\rangle &\equiv \exists\, h.\, \langle\!|p|\!\rangle\ S\ \langle\!|h, q_2|\!\rangle \wedge [\![h]\!]\ T\ [\![q_1, p \vee q_2]\!] \\
\langle\!|p|\!\rangle\ S\ ;;\ T\ \langle\!|q_1, q_2|\!\rangle &\equiv \exists\, h.\, [\![p]\!]\ S\ [\![q_1, h]\!] \wedge [\![h]\!]\ T\ [\![q_1, p \vee q_2]\!] \\
\langle\!|p|\!\rangle\ S \sqcap T\ \langle\!|q_1, q_2|\!\rangle &\equiv \langle\!|p|\!\rangle\ S\ \langle\!|q_1, q_2|\!\rangle \wedge \langle\!|p|\!\rangle\ T\ \langle\!|q_1, q_2|\!\rangle \\
\langle\!|p|\!\rangle\ \mathsf{check}\ B\ \langle\!|q_1, q_2|\!\rangle &\equiv \Delta B \wedge \text{`}B\text{'} \wedge p \Rightarrow q_1 \\
\langle\!|p|\!\rangle\ \mathsf{if}\ B\ \mathsf{then}\ S\ \mathsf{else}\ T\ \langle\!|q_1, q_2|\!\rangle &\equiv [\![\Delta B \wedge \text{`}B\text{'} \wedge p]\!]\ S\ [\![q_1, p \vee q_2]\!] \wedge \\
&\qquad [\![\Delta B \wedge \neg\text{`}B\text{'} \wedge p]\!]\ T\ [\![q_1, p \vee q_2]\!]
\end{aligned}$$

The raise statement miraculously satisfies any fail-safe correctness specification by failing and leaving the state unchanged. The rules for assignment and nondeterministic assignment have only conditions in case the expression is defined; in case the expression is undefined, the assignment fails without changing the state, thus satisfies the fail-safe correctness specification automatically. Likewise, the check statement and the conditional have only conditions in case B is defined. If B is undefined, the statement fails without changing the state. We immediately get the following consequence rule for any statement S:

$$\langle\!|p|\!\rangle\ S\ \langle\!|q_1, q_2|\!\rangle \wedge (q_1 \Rightarrow q_1') \wedge (q_2 \Rightarrow q_2') \Rightarrow \langle\!|p|\!\rangle\ S\ \langle\!|q_1', q_2'|\!\rangle$$

Like for total correctness, this rule allows the postconditions to be weakened; however, it does not allow the precondition to be weakened.

For $S\,;\,T$ and $\mathsf{try}\,S\,\mathsf{catch}\,T$ let us consider the special case when $q_2 \equiv \mathsf{false}$:

$$\langle\!|p|\!\rangle\; S\,;\,T\;\langle\!|q_1|\!\rangle \equiv \exists\, h.\, \langle\!|p|\!\rangle\; S\;\langle\!|h|\!\rangle \wedge [\![h]\!]\; T\;[\![q_1, p]\!]$$

$$\langle\!|p|\!\rangle\;\mathsf{try}\,S\,\mathsf{catch}\,T\;\langle\!|q_1|\!\rangle \equiv \exists\, h.\, [\![p]\!]\; S\;[\![q_1, h]\!] \wedge [\![h]\!]\; T\;[\![q_1, p]\!]$$

The fail-safe correctness assertion for $S\,;\,T$ is satisfied if S fails without changing the state, but if S succeeds with h, then T must either succeed with the specified postcondition q_1, or fail with the original precondition p. For the fail-safe correctness assertion of $\mathsf{try}\,S\,\mathsf{catch}\,T$ to hold, either S must succeed with q_1, or fail with h, from which T either succeeds with q_1 or fails with the original precondition p.

6.4 Loop Theorems

Let p_w be ranked predicates in which $w \in W$ is a well-founded set. The fundamental rules for total correctness and fail-safe correctness of loops are as follows:

Theorem 6.4. ✓ *Assume that B is a Boolean program expression, q_2 is a predicate and S is a statement. Assume that p_w for $w \in W$ is a ranked collection of predicates. Then*

$$\begin{aligned}&\forall\, w \in W.\, [\![p_w \wedge \Delta B \wedge \text{`}B\text{'}]\!]\; S\;[\![p_{<w}, q_2]\!]\\ \Rightarrow &[\![p]\!]\;\mathsf{while}\,B\,\mathsf{do}\,S\;[\![p \wedge \Delta B \wedge \neg\text{`}B\text{'}, (p \wedge \neg\Delta B) \vee q_2]\!]\end{aligned}$$

and

$$\begin{aligned}&\forall\, w \in W.\, \langle\!|p_w \wedge \Delta B \wedge \text{`}B\text{'}|\!\rangle\; S\;\langle\!|p_{<w}, q_2|\!\rangle\\ \Rightarrow &\langle\!|p|\!\rangle\;\mathsf{while}\,B\,\mathsf{do}\,S\;\langle\!|p \wedge \Delta B \wedge \neg\text{`}B\text{'}, q_2|\!\rangle\end{aligned}$$

These theorems separate the concerns of the two exits: on the normal exit, both rules are similar as with one exit, except that the postconditions include the definedness of the guard B. On the exceptional exit, the rule for total correctness is the

same as its counterpart with one exit [Back and von Wright, 1998]; on the exceptional exit, it states that if the loop body S exits exceptionally with postcondition q_2, then the loop exits exceptionally with postcondition $p \wedge \neg\Delta B$ (failure of the guard) or q_2 (failure of the loop body).

6.5 Fail-Safe Refinement

Formal specification techniques allow expressing idealized specifications, which abstract from restrictions that may arise in implementations. However, partial implementations are universal in software development due to practical limitations. In software development, specifications are meant to be concise by stating abstractly only the intention of a program rather than elaborating on a possible implementation. However, practical restrictions can prevent idealized specifications from being fully implemented. In general, there are three sources of partiality in implementations: there may be inherent limitations of the implementation, some features may intentionally not (yet) be implemented, or there may be a genuine fault.

As an example of inherent limitations of an implementation, consider a class for the analysis of a collection of integers. The operations are initialization, inserting an integer, and summing all its elements. Assume that int is a type for machine-representable integers, bounded by min and max, and machine arithmetic is bounded, *i.e.*, an overflow caused by arithmetic operations on int is detected and raises an exception, as available in x86 assembly language [Intel, 2013] and .NET [Microsoft, 2013b]. We define:

class *IntCollection*
  bag(int) b
  invariant $inv : \forall\, x \in b.\ \mathsf{min} \leq x \leq \mathsf{max}$
  method $init()$

$b := []$
method $insert(n : \mathsf{int})$
$b := b + [n]$
method $sum() : \mathsf{int}$
result $:= \sum x \in b \cdot x$

This specification allows an unbounded number of machine-representable integers to be stored in the abstract bag b, which is an unordered collection that, unlike a set, allows duplication of elements, The empty bag is written as [], the bag consisting only of a single n as $[n]$, union of bags b, b' as $b + b'$, and the sum of all elements of bag b as $\sum x \in b \cdot x$. (A model of bags is a function from elements to their number of occurrences in the bag.) However, in an implementation, method *init* (object initialization) and *insert* may fail due to memory exhaustion at heap allocation (and raise an exception), and method *sum* may fail due to overflow of the result (and raise an exception). Even if the result of *sum* is machine-representable, the iterative computation of the sum in a particular order may still overflow. Hence no realistic implementation of this class can be faithful. Still, it is a useful specification because of its clarity and brevity. Obviously, using mathematical integers instead of machine-representable integers in the specification would make implementations even "more partial".

The second source of partiality is intentionally missing features. The evolutionary development of software consists of anticipating and performing *extensions* and *contractions* [Parnas, 1978]. Anticipated extensions can be expressed by features that are present but not yet implemented. Contractions lead to obsolete features that will eventually be removed. In both cases, implementations of features may be missing. A common practice is to raise an exception if a feature is not yet implemented. Here is a recommendation from the Microsoft Developer Documentation for .NET [Microsoft,

2013a]:

```
static void FutureFeature()
{
    // Not developed yet.
    throw new NotImplementedException();
}
```

The third source of partiality is genuine faults. These may arise from the use of software layers that are themselves faulty (operating system, compilers, libraries), from faults in the hardware (transient or permanent), or from errors in the design (errors in the correctness argument, incorrect hypothesis about the abstract machine).

Our goal is to contribute to a method of program refinement that allows for partial implementations that guarantee "safe" failure if the desired outcome cannot be computed. For program statements S (the specification) and T (the implementation), the total refinement of S by T means that either T terminates normally and establishes a postcondition that S may also establish on normal termination, or T terminates exceptionally and establishes a postcondition that S may also establish on exceptional termination. As a relaxation, *fail-safe refinement* allows T additionally to terminate exceptionally provided the initial state is preserved. The intention is that the implementation T tries to meet specification S, but if it cannot do so, T can fail safely by not changing the state and terminating exceptionally. When applying fail-safe refinement to data refinement, an implementation that cannot meet the specification and fails may still change the state as long as the change is not visible in the specification.

The exception handling of the Eiffel programming language provided the inspiration for fail-safe refinement [Meyer, 1997]: in Eiffel, each method has one entry and two exits, a normal and an exceptional exit, but is specified by a single precondition and single postcondition only. The normal exit is taken if the desired postcondition

is established and the exceptional exit is taken if the desired postcondition cannot be established, thus allowing partial implementations. In Eiffel, the postcondition must be evaluated at run-time to determine if the normal or exceptional exit is taken (and hence must be efficiently computable). Fail-safe refinement is more general in the sense that it does not require that a postcondition to be evaluated at run-time, as long as faults are detected at run-time in some way. Fail-safe refinement is related to this notion of fail-safe correctness in the same sense as total refinement is related to total correctness.

Retrenchment also addresses the issue of partial implementations, but with statements with one entry and one exit [Banach et al., 2007]: the refinement of each operation of a data type requires a *within* and a *concedes* relation that restrict the initial states and widen the possible final states of an implementation. Compared to retrenchment, fail-safe refinement does not require additional relations to be specified. Fail-safe refinement is more restrictive in the sense that initial states cannot be restricted and the final state cannot be widened on normal termination. In fail-safe refinement, the caller is notified through an exception of the failure; retrenchment is a design technique that is independent of exception handling.

In Sekerinski and Zhang [2012] we use "partial refinement" for fail-safe refinement for lack of a better name. The term "partial refinement" has been introduced in [Back, 1981] for statements with one exit. There, partial refinement allows the domain of termination to be reduced. In this thesis, fail-safe refinement requires termination, either on the normal or exceptional exit. Partial refinement in [Jeffords et al., 2009] refers to transition system refinement with a partial refinement relation; the approach is to construct fault-tolerant systems in two phases, first with an idealized specification and then adding fault-tolerant behaviour. Here we use fail-safe refinement specifically for statements with two exits.

We relax total refinement to fail-safe refinement and study its application. Fail-safe refinement of double-exit predicate transformers S, T is defined by:

$$S \sqsubseteq\!\!\!\!\sim\; T \mathrel{\widehat{=}} S \sqcap \mathsf{raise} \sqsubseteq T$$

This implies that on normal exit, T can only do what S does. However, T may fail when S does not, but then has to preserve the initial state. For example, in such a case T would still maintain an invariant and signal to the caller the failure. Note that the types of S and T require the initial state space to be the same as the final state space on exceptional exit. Partial refinement is related to fail-safe correctness in the same way as total refinement is related to total correctness:

Theorem 6.5. ✓ *For double-exit predicate transformers S, T:*

$$S \sqsubseteq T \equiv \forall\, p, q_1, q_2.\, [\![p]\!]\, S\, [\![q_1, q_2]\!] \Rightarrow [\![p]\!]\, T\, [\![q_1, q_2]\!]$$
$$S \sqsubseteq\!\!\!\!\sim\; T \equiv \forall\, p, q_1, q_2.\, \langle\!|p|\!\rangle\, S\, \langle\!|q_1, q_2|\!\rangle \Rightarrow \langle\!|p|\!\rangle\, T\, \langle\!|q_1, q_2|\!\rangle$$

Proof. Here we give proof only for the second one. Unfolding the definitions yields:

$$\begin{aligned} &\forall\, q_1, q_2.\, (S\,(q_1, q_2) \wedge q_2) \leq T\,(q_1, q_2) \\ \equiv\; &\forall\, p', q_1', q_2'.\, p' \leq S\,(q_1', p' \vee q_2') \Rightarrow p' \leq T\,(q_1', p' \vee q_2') \end{aligned}$$

This is shown by mutual implication. For any p', q_1' and q_2', by letting q_1 and q_2 to be q_1' and $p' \vee q_2'$ respectively, it is straightforward that the left side implies the right side. Implication of the other direction is more involved: for any p and q_1, letting p', q_1' and q_2' to be $S\,(q_1, q_2) \wedge q_2$, q_1 and q_2 respectively gives us $(S\,(q_1, q_2) \wedge q_2) \leq S\,(q_1, S\,(q_1, q_2) \wedge q_2 \vee q_2) \Rightarrow (S\,(q_1, q_2) \wedge q_2) \leq T\,(q_1, S\,(q_1, q_2) \wedge q_2 \vee q_2)$. Since $S\,(q_1, q_2) \wedge q_2 \vee q_2 = q_2$, we have $(S\,(q_1, q_2) \wedge q_2) \leq S\,(q_1, q_2) \Rightarrow (S\,(q_1, q_2) \wedge q_2) \leq T\,(q_1, q_2)$, which reduces to $S\,(q_1, q_2) \wedge q_2 \leq T\,(q_1, q_2)$, thus the left side is implied. □

Total refinement implies fail-safe refinement (in the same way as total correctness implies fail-safe correctness):

Theorem 6.6. ✓ *For predicates p, q_1, q_2 and double-exit predicate transformers S, T:*

$$[\![p]\!]\, S\, [\![q_1, q_2]\!] \Rightarrow \langle\!|p|\!\rangle\, S\, \langle\!|q_1, q_2|\!\rangle$$

$$S \sqsubseteq T \Rightarrow S \mathrel{\sqsubseteq\!\!\!\sim} T$$

In addition to stop as top element, fail-safe refinement has also raise as top element:

Theorem 6.7. ✓ *For double-exit predicate transformers S, T:*

$$S \mathrel{\sqsubseteq\!\!\!\sim} \mathsf{raise}$$

$$S \mathrel{\sqsubseteq\!\!\!\sim} \mathsf{stop}$$

The fact that $S \mathrel{\sqsubseteq\!\!\!\sim} \mathsf{raise}$ may be surprising, but it does allow for intentionally missing features, the second source of partiality discussed earlier on. Like total refinement, fail-safe refinement is a preorder, as it is reflexive and transitive:

Theorem 6.8. ✓ *For double-exit predicate transformers S, T, U:*

$$S \mathrel{\sqsubseteq\!\!\!\sim} S$$

$$S \mathrel{\sqsubseteq\!\!\!\sim} T \wedge T \mathrel{\sqsubseteq\!\!\!\sim} U \Rightarrow S \mathrel{\sqsubseteq\!\!\!\sim} U$$

Fail-safe refinement is not antisymmetric, for example $\mathsf{stop} \mathrel{\sqsubseteq\!\!\!\sim} \mathsf{raise}$ and $\mathsf{raise} \mathrel{\sqsubseteq\!\!\!\sim} \mathsf{stop}$, but $\mathsf{stop} \neq \mathsf{raise}$. With respect to fail-safe refinement, sequential composition is monotonic only in its first operand, while demonic choice and conditional statement are monotonic in both operands:

Theorem 6.9. ✓ *For statements S, S', T:*

$$S \mathrel{\sqsubseteq\!\!\!\sim} S' \Rightarrow S\,;\, T \mathrel{\sqsubseteq\!\!\!\sim} S'\,;\, T$$

$$S \mathrel{\sqsubseteq\!\!\!\sim} S' \wedge T \mathrel{\sqsubseteq\!\!\!\sim} T' \Rightarrow S \sqcap T \mathrel{\sqsubseteq\!\!\!\sim} S' \sqcap T'$$

$$\wedge\, \mathsf{if}\ B\ \mathsf{then}\ S\ \mathsf{else}\ T \mathrel{\sqsubseteq\!\!\!\sim} \mathsf{if}\ B\ \mathsf{then}\ S'\ \mathsf{else}\ T'$$

However, $S \sqsubseteq\!\!\!\!\sim\ S'$ does not imply $T\ ;\ S \sqsubseteq\!\!\!\!\sim\ T\ ;\ S'$ in general, since T might modify the initial state on normal exit. Similarly, $S \sqsubseteq\!\!\!\!\sim\ S'$ implies neither $S\ ;;\ T \sqsubseteq\!\!\!\!\sim\ S'\ ;;\ T$ nor $T\ ;;\ S \sqsubseteq\!\!\!\!\sim\ T\ ;;\ S'$.

For *data refinement*, which makes the data structure more concrete while the correctness is maintained in all circumstances and for all purposes [He et al., 1986], we extend the refinement relationships to allow two double-exit predicate transformers on possibly different state spaces. We extend the definition of data refinement with double-exit predicate transformer, *e.g.*, [Gardiner and Morgan, 1991; von Wright, 1994], which uses an representation operation to link concrete and abstract spaces, to two postconditions. Suppose $S : \mathcal{P}\Sigma \times \mathcal{P}\Sigma \to \mathcal{P}\Sigma$ and $T : \mathcal{P}\Gamma \times \mathcal{P}\Gamma \to \mathcal{P}\Gamma$ are double-exit predicate transformers on Σ and Γ respectively, and $r : \Sigma \to \mathcal{P}\Gamma$ is the abstraction relation from Σ to Γ. Then we define:

$$T_r \mathrel{\widehat{=}} [r]\ ;\ ((T\ ;\ \{r^{-1}\})\ ;;\ \{\bot, r^{-1}\})$$

The composition $(T;\{r^{-1}\});;\{\bot, r^{-1}\}$ applies the angelic update $\{r^{-1}\}$ to the normal outcome of T, terminating normally, and applies $\{r^{-1}\}$ to the exceptional outcome of T, terminating exceptionally. This can be equivalently expressed as $(T\ ;;\ \{\bot, r^{-1}\})\ ;\ \{r^{-1}\}$. The demonic update $[r]$ maps the states in Σ to states in Γ. Then, after executing T, $\{r^{-1}\}$ and $\{\bot, r^{-1}\}$ map the states back to Σ from states in Γ, on each exit. In other words, T_r is the projection of T on $\mathcal{P}\Sigma \times \mathcal{P}\Sigma \to \mathcal{P}\Sigma$ through relation r. Now we can define *total data refinement* and *partial data refinement* between S and T through r as:

$$S \sqsubseteq_r T \mathrel{\widehat{=}} S \sqsubseteq T_r$$

$$S \sqsubseteq\!\!\!\!\sim_r T \mathrel{\widehat{=}} S \sqsubseteq\!\!\!\!\sim T_r$$

Note that here we could not define $S \sqsubseteq\!\!\!\!\sim_r T$ as $(S\ ;\ [r])\ ;;\ [r] \sqsubseteq\!\!\!\!\sim [r]\ ;\ T$, due to the restriction of $\sqsubseteq\!\!\!\!\sim$ that on both sides the initial state space must be the same as the exceptional state space, which $[r]\ ;\ T$ obviously does not satisfy.

Example: Limitation in Class Implementation

Now let us revisit the introductory example of the class *IntCollection*. We consider an implementation using a fixed-size array. Using dynamic arrays or a heap-allocated linked list would be treated similarly, as an extension of a dynamic array and heap allocation may fail in the same way as a fixed-sized array may overflow. Since representing dynamic arrays or heaps complicates the model, we illustrate the refinement step using fixed-sized arrays. Let *SIZE* be a constant of type int:

class $IntCollection1$
  int l, int[] a
  invariant $inv1 : 0 \leq l \leq SIZE \wedge len(a) = SIZE \wedge$
    $(\forall\, x.\, 0 \leq x < l \Rightarrow \mathsf{min} \leq a[x] \leq \mathsf{max})$
  method $init1()$
    $l := 0$; $a :=$ new int$[SIZE]$;
  method $insert1(n : \mathsf{int})$
    $l, a[l] := l + 1, n$;
  method $sum1() : \mathsf{int}$
    int $s, i := 0, 0$;
    {loop invariant $inv : 0 \leq i \leq l \wedge$
      $s = \sum x \in [0..i) \cdot a[x] \wedge \mathsf{min} \leq s \leq \mathsf{max}$)}
    while $i < l$ do
      $s, i := s + a[i], i + 1$;
    $\{s = \sum x \in 0..l - 1 \cdot a[x] \wedge \mathsf{min} \leq s \leq \mathsf{max}, inv1\}$
    result $:= s$

The state spaces of classes *IntCollection* and $IntCollection1$ are bag(int) and int $\times$ int[] respectively. The invariant that links the state spaces is $b = bagof(a[0..l-1])$, where *bagof* converts an array to a bag with the same elements. The statement $a :=$ new int$[SIZE]$ might fail due to heap allocation failure. Writing s^n for n times repeating sequence s, allocation of an integer array is define as:

$x :=$ new int$[n] \mathrel{\widehat{=}} [\mathsf{true}, \mathsf{true}]$; $x := [0]^n$

where [true, true] might succeed or fail. Here the three methods refine those in class *IntCollection* respectively, formally $init \sqsubseteq_{rel} init1$, $insert \sqsubseteq_{rel} insert1$, and $sum \sqsubseteq_{rel1} sum1$, in which the relations are given as $rel\, b\, (l, a) \equiv b = bagof\,(a[0..l-1])$ and $rel1\, (b, \mathsf{result})\, (l, a, \mathsf{result}') \equiv b = bagof\,(a[0..l-1]) \wedge \mathsf{result} = \mathsf{result}'$, since result is part of the state space in $sum1$. We only give the proof sketch of the third one. Using *body* and *loop* as the abbreviations of $s, i := s + a[i], i + 1$ and $\mathsf{while}\, i < l\, \mathsf{do}\, s, i := s + a[i], i + 1$ respectively, and defining $B = (\lambda(s, i, l, a, \mathsf{result}) \cdot i < l)$, $linv_w = (linv \wedge (\lambda(s, i, l, a, \mathsf{result}) \cdot w = l - i))$, we have $\Delta B = \mathsf{true}$ and:

$$[\![linv \wedge \Delta B \wedge B]\!]\ body\ [\![linv_{<w}, linv]\!]$$

According to Theorem 6.4 we have

$$[\![linv \wedge (\lambda(s, i, l, a, \mathsf{result}) \cdot s = 0 \wedge i = 0)]\!]\ loop\ [\![linv \wedge \Delta B \wedge \neg B, linv]\!]$$

and $linv \wedge \Delta B \wedge \neg B \Rightarrow s = \sum x \in 0..l-1 \cdot a[x]$. We require that the exceptional postconditions must be independent of result since no value will be returned in failures. With the correctness rule for sequential composition we know that for arbitrary q_1, q_2:

$$[\![(\lambda(l, a, \mathsf{result}) \cdot q_1\, (l, a, \sum x \in 0..l-1 \cdot a[x])) \wedge q_2]\!]\ sum1\ [\![q_1, q_2]\!]$$

Since for arbitrary q_1', q_2',

$$sum\, (q_1', q_2') = (\lambda b, \mathsf{result} \cdot q_1'\, (\sum x \in b \cdot x, b) \wedge \mathsf{min} \leq \sum x \in b \cdot x \leq \mathsf{max}) \wedge q_2'$$

by definition we know that $sum \sqsubseteq_{rel1} sum1$.

Furthermore, $inv1\, (l, a) \wedge rel\, b\, (l, a) \Rightarrow inv\, b$, which means that the original invariant *inv* is preserved by the new invariant $inv1$ through relation *rel*. In $sum1$, local variables s and i would be erased on both exits, thus in exceptional termination caused by arithmetic overflow inside the loop, the original state (l, a) remains the same, maintaining the invariant on exceptional exit.

Example: Incremental Development

Another use of fail-safe refinement is to express incremental development. When each of two programs handles some cases of the same specification, then their combination can handle no fewer cases, while remaining a fail-safe refinement of the specification:

$$\begin{aligned} & S \sqsubseteq_{\approx} \mathsf{if}\ B\ \mathsf{then}\ T\ \mathsf{else}\ \mathsf{raise} \wedge S \sqsubseteq_{\approx} \mathsf{if}\ B'\ \mathsf{then}\ T'\ \mathsf{else}\ \mathsf{raise} \\ \Rightarrow & S \sqsubseteq_{\approx} \mathsf{if}\ B\ \mathsf{then}\ T\ \mathsf{else}\ (\mathsf{if}\ B'\ \mathsf{then}\ T'\ \mathsf{else}\ \mathsf{raise}) \end{aligned}$$

Moreover, for conditional disjunction $\vee_c$, we have

$$\begin{aligned} & \mathsf{if}\ B\ \mathsf{then}\ T\ \mathsf{else}\ (\mathsf{if}\ B'\ \mathsf{then}\ T'\ \mathsf{else}\ \mathsf{raise}) \\ = & \mathsf{if}\ B \vee_c B'\ \mathsf{then}(\mathsf{if}\ B\ \mathsf{then}\ T\ \mathsf{else}\ T')\ \mathsf{else}\ \mathsf{raise} \end{aligned}$$

which allows the combination of more than two such programs in a switch-case style, since the right-hand side is again in the "$\mathsf{if} \ldots \mathsf{then} \ldots \mathsf{else}\ \mathsf{raise}$" form. With more and more such partial implementations combined this way, ideally more cases can be handled incrementally.

6.6 Discussion

We introduced fail-safe correctness for modelling programs that can fail "safely", and fail-safe refinement for the description of programs that are unable to fully implement their specifications. Using double-exit predicate transformers for the semantics of statements allows us to specify the behaviour on both normal and exceptional exits. King and Morgan [1995] as well as Watson [2002] present a number of rules for total refinement, although using different programming constructs: $S > T$, pronounced "S else T", can be defined here as $S \sqcap (T ; \mathsf{raise})$, the *specification statement* $x : [p, q_1 > q_2]$ with *frame* x can be defined here as $\{p\} ; [\overline{q_1}, \overline{q_2}]$, where $\overline{q_1}, \overline{q_2}$ lifts predicates q_1, q_2 to relations over x, the *exception block* $[\![S]\!]$ with $\mathsf{raise}\, H$ in its body, where H is the exception handler, can be defined here as $S \,;; H$. This allows their total refinement

rules to be used here. However, the purpose of fail-safe refinement is different and we would expect different kinds of rules needed.

For two sources of partiality, inherent limitations of an implementation and intentionally missing implementation, the two examples in Section 6.5 illustrate how fail-safe refinement can be used to help in reasoning. For the third source of partiality, genuine faults, the approach needs further elaboration. In next chapter, we apply the notions of fail-safe correctness to three design patterns for fault tolerance, rollback, degraded service, and recovery block. These ideas can be carried over to fail-safe refinement.

Implementation restrictions are also addressed by IO-refinement, which allows the type of method parameters to be changed in refinement steps [Boiten and Derrick, 1998; Banach et al., 2007; Derrick and Boiten, 2001; Mikhajlova and Sekerinski, 1997; Stepney et al., 1998]. Fail-safe refinement and IO-refinement are independent of each other and can be combined.

Chapter 7

Design Patterns for The Termination Model

In this section we illustrate the use of total correctness and fail-safe correctness with six design patterns, and three of them are with loops. These patterns are studied in [Sekerinski, 2011], providing fault-tolerant solutions for different problems.

7.1 Design Patterns without Loops

Rollback

When a statement fails, it may leave the program in an inconsistent state, for example in one in which an invariant does not hold and from which another failure is likely, or in an undesirable state, for example one in which the only course of action is termination of the program. We give a pattern for *rolling back* to the original state such that the failure is *masked*, meaning that is it not visible to the outside. Rolling back relies on a procedure *backup*, which makes a copy of the state, and a procedure *restore*, which restores the saved state. We formalize this by requiring that *backup*

establishes a predicate k, which *restore* requires to roll back, and which the attempted statement, called S, has to preserve in case of failure. The backup may consist of a copy of all variables in main memory or secondary storage, or a partial or compressed copy, as long as a state satisfying k can be established. The attempted statement S does not need to preserve k in case of success, *e.g.*, can overwrite the backup of the variables. We let statement T do some "cleanup" after restoring to achieve the desired postcondition.

Theorem 7.1. ✓ *Let k, p, q be predicates and let backup, restore, S, T be statements. If*

$\langle\!|p|\!\rangle$ *backup* $\langle\!|p \wedge k|\!\rangle$ (establishes k from p or fails with p)

$[\![k]\!]$ *restore* $[\![p]\!]$ (restores p from k)

$[\![p \wedge k]\!]\ S\ [\![q, k]\!]$ (establishes q or fails with k)

$\langle\!|p|\!\rangle\ T\ \langle\!|q|\!\rangle$ (establishes q or fails with p)

then:

$\langle\!|p|\!\rangle$ *backup* ; try S catch (*restore* ; T) $\langle\!|q|\!\rangle$

Procedure *backup* may either fail with p or succeed with $p \wedge k$, but *restore* must always succeed. The cleanup T must either succeed with q or fail with its precondition p. Thus it can be implemented by raise, which would be an example of re-raising an exception.

Degraded Service

Suppose that two or more statements achieve the same goal, but some statements are preferred over others—the preferred one may be more efficient, may achieve a higher precision of numeric results, may transmit faster over the network, may achieve a higher sound quality. If the most preferred one fails, we may *fall back* to one that is

less desirable, but more likely to succeed, and if that fails, fall back to a third one, and so forth. The least preferred one may simply inform the user of the failure. We call this the pattern of *degraded service*. In the formulation below degraded service is combined with rollback such that each attempt starts in the original state, rather than in the state that was left from the previous attempt. Hence, all alternatives have to adhere to the same specification, but try to satisfy that by different means. In case all attempts fail, the failure is propagated to the user.

Theorem 7.2. ✓ *Let k, p, q be predicates and let backup, restore, S_1, S_2 be statements. If*

$\langle\!|p|\!\rangle$ *backup* $\langle\!|p \wedge k|\!\rangle$ (establishes k from p or fails with p)

$[\![k]\!]$ *restore* $[\![p \wedge k]\!]$ (restores p from k)

$[\![p \wedge k]\!]\ S_1\ [\![q, k]\!]$ (establishes q or fails with k)

$[\![p \wedge k]\!]\ S_2\ [\![q, k]\!]$ (establishes q or fails with k)

then:

$\langle\!|p|\!\rangle$ *backup* ; try S_1 catch (*restore* ; try S_2 catch (*restore* ; raise)) $\langle\!|q|\!\rangle$

The theorem readily generalizes to more than two attempts.

Recovery Block

The *recovery block* specifies n alternatives together with an *acceptance test* [Horning et al., 1974]. The alternatives are executed in the specified order. If the acceptance test at the end of an alternative fails or an exception is raised within an alternative, the original state is restored and the next alternative attempted. If an acceptance test passes, the recovery block terminates. If the acceptance test fails for all alternatives, the recovery block fails, possibly leading to alternatives taken at an outer level. Here is the originally suggested syntax of [Randell, 1975] and a formulation with try-catch

statements [Sekerinski, 2011]; B is the acceptance test:

ensure B	*backup* ;
by S_1	try (S_1 ; check B)
else by S_2	catch
else by S_3	*restore* ;
else raise	try (S_2 ; check B)
	catch
	restore ;
	try (S_3 ; check B)
	catch (*restore* ; raise)

The acceptance test does not have to be the complete postcondition; that would be rather impractical in general. However, suppose that we know that alternative S_i terminates with q_i, if it succeeds. If we can devise a predicate B_i such that $q_i \wedge$ 'B_i' implies the desired postcondition q, then B_i is an adequate *acceptance test* for S_i; for this each alternative has to have its own acceptance test, a possibility already mentioned in [Randell, 1975]:

Theorem 7.3. ✓ *Let k, p, q, q_1, q_2, q_3 be predicates, let B_1, B_2, B_3 be program expressions, and let backup, restore, S_1, S_2, S_3 be statements. If*

$\langle\!\vert p \vert\!\rangle$ *backup* $\langle\!\vert p \wedge k \vert\!\rangle$	(establishes k from p or fails with p)
$[\![k]\!]$ *restore* $[\![p \wedge k]\!]$	(restores p from k)
$[\![p \wedge k]\!]\ S_1\ [\![q_1 \wedge k, k]\!]$	(establishes q_1 or fails with k)
$[\![p \wedge k]\!]\ S_2\ [\![q_2 \wedge k, k]\!]$	(establishes q_2 or fails with k)
$[\![p \wedge k]\!]\ S_3\ [\![q_3 \wedge k, k]\!]$	(establishes q_3 or fails with k)

$$q_1 \Rightarrow \Delta B_1 \qquad q_1 \wedge \text{`}B_1\text{'} \Rightarrow q$$

$$q_2 \Rightarrow \Delta B_2 \qquad q_2 \wedge \text{`}B_2\text{'} \Rightarrow q$$

$$q_3 \Rightarrow \Delta B_3 \qquad q_3 \wedge \text{`}B_3\text{'} \Rightarrow q$$

then:

$\langle\!| p |\!\rangle$
backup ;
try $(S_1$; check $B_1)$
catch
  restore ;
  try $(S_2$; check $B_2)$
  catch
    restore ;
    try $(S_3$; check $B_3)$
    catch $(restore$; raise$)$
$\langle\!| q |\!\rangle$

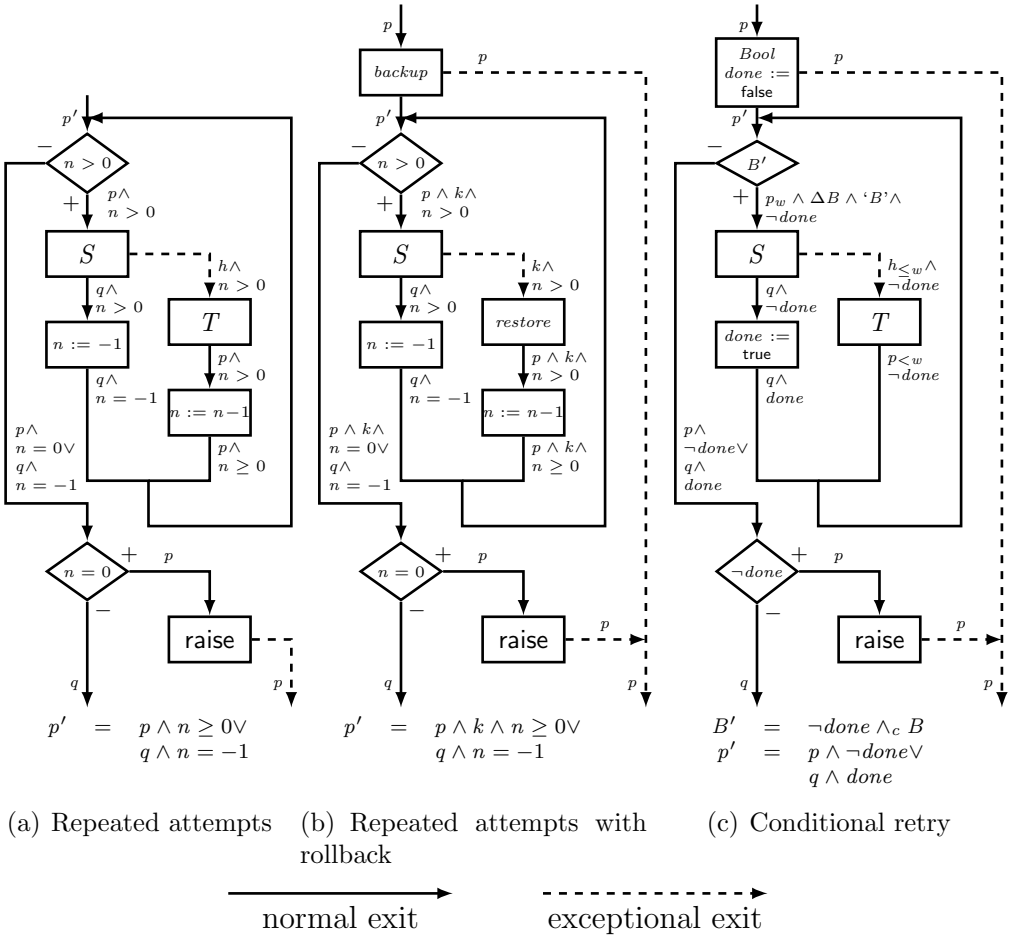
More generally, partial acceptance tests in form of additional check statements can be carried out anywhere within an alternative, rather than only at the end; failure should be detected early such that resources are not wasted.

The theorem is a consequence of degraded service with rollback, generalized to three attempts, by replacing S_1 by S_1 ; check $B_1, \ldots$. The conclusion follows immediately provided that $[\![p \wedge k]\!]\, S_1$; check $B_1\, [\![q, k]\!], \ldots$ hold. Given $[\![p \wedge k]\!]\, S_1\, [\![q_1 \wedge k, k]\!]$, $q_1 \Rightarrow \Delta B_1$, and $q_1 \wedge \text{`}B_1\text{'} \Rightarrow q, \ldots$ this follows by the rules for total correctness assertions.

7.2 Design Patterns with Loops

Repeated Attempts

Failures may be transient, *e.g.*, because environmental influences, unreliable hardware, or temporary usage of resources by other programs. In such cases, a strategy is to repeat the failing statement, perhaps after a delay. In the pattern of repeated attempts, statement S is attempted at most n times, $n \geq 0$. When S succeeds, the



(a) Repeated attempts (b) Repeated attempts with rollback (c) Conditional retry

Figure 7.1: Flowcharts of Three Design Patterns with Loops

whole statement succeeds, if S fails n times, the whole pattern fails, see Figure 7.1.

Theorem 7.4. ✓ *Let p, q be predicates that are independent of integer variable n, let S, T be statements that do not assign to n, and let ra be defined by:*

$$
\begin{aligned}
ra = \;& \mathsf{while}\; n > 0 \;\mathsf{do} \\
& \quad \mathsf{try}\; (S\,;\, n := -1) \\
& \quad \mathsf{catch}\; (T\,;\, n := n-1)\,; \\
& \mathsf{if}\; n = 0 \;\mathsf{then}\; \mathsf{raise}
\end{aligned}
$$

If

$[\![p]\!]\ S\ [\![q, h]\!]$ (establishes q from p or fails with h)

$[\![h]\!]\ T\ [\![p]\!]$ (establishes p from h)

then:

$\langle\!|p|\!\rangle\ ra\ \langle\!|q|\!\rangle$

Proof. Let

$$loop = \mathsf{while}\ n > 0\ \mathsf{do}\ \mathsf{try}\ (S\ ;\ n := -1)\ \mathsf{catch}\ (T\ ;\ n := n - 1)$$

and let

$$p' = n \geq 0 \wedge p \vee n = -1 \wedge q$$

$$v' = (\lambda\,(n, \sigma).\ n + 1)$$

$$l' = (\lambda\, n\ n'.\ n < n')$$

$$B' = n > 0$$

$$S' = \mathsf{try}\ (S\ ;\ n := -1)\ \mathsf{catch}\ (T\ ;\ n := n - 1)$$

$$q'_2 = n \geq 0 \wedge p$$

then

$$(\forall\, w \in W.\ [\![p'_w \wedge \Delta B' \wedge \text{`}B'\text{'}]\!]\ S'\ [\![p'_{<w}, q'_2]\!])$$

From Theorem 6.4, we have

$$[\![p']\!]\ \mathsf{while}\ B'\ \mathsf{do}\ S'\ [\![p' \wedge \Delta B' \wedge \neg B', (p' \wedge \neg \Delta B') \vee q'_2]\!]$$

thus

$$[\![n \geq 0 \wedge p \vee n = -1 \wedge q]\!]\ loop\ [\![n = 0 \wedge p \vee n = -1 \wedge q, n \geq 0 \wedge p]\!]$$

so

$\langle\!|p|\!\rangle\ ra\ \langle\!|q|\!\rangle$

□

Repeated Attempts with Rollback

This pattern is a special case of the last one. It assumes that if S fails, T can re-establish the original state. This can be achieved by rolling back, provided that an initial backup is made, see Figure 7.1.

Theorem 7.5. ✓ *Let p, q be predicates in which integer variable n does not occur, let S, backup, restore be statements that do not assign to n, and let rr be defined by:*

$rr =$ *backup* ;
  while $n > 0$ **do**
    try $(S\,;\, n := -1)$
    catch $(\mathit{restore}\,;\, n := n - 1)$;
  if $n = 0$ **then raise**

If

$\langle\!|p|\!\rangle$ *backup* $\langle\!|p \wedge k|\!\rangle$ (establishes k from p or fails with p)

$[\![k]\!]$ *restore* $[\![p \wedge k]\!]$ (restores p from k)

$[\![p \wedge k]\!]\ S\ [\![q, k]\!]$ (establishes q or fails with k)

then:

$\langle\!|p|\!\rangle$ *rr* $\langle\!|q|\!\rangle$

The requirement on S is now weakened, as in case of failure S has only to preserve the backup k; S does not have to preserve k in case of successful termination.

Proof. Let S, T, p, q, h in Theorem 7.4 be S, *restore*, $p \wedge k$, q and k here, respectively, then the conclusion is straightforward. □

Conditional Retry

Instead of attempting a statement a fixed number of times, we may need to make attempts dependent on a condition. However, that condition has eventually to be become false. In the pattern of conditional retry, we ensure termination of attempts by requiring that the handler decreases a variant, see Figure 7.1. This pattern mimics the rescue and retry statements of Eiffel [Meyer, 1997].

Theorem 7.6. ✓ *Let p, q be predicates in which Boolean variable done does not occur, let S, T be statements that do not assign to done, let v be an integer expression, and let cr be defined by:*

$$
\begin{array}{l}
cr = \mathit{Bool}\ \mathit{done} := \mathsf{false}\,; \\
\quad \mathsf{while}\ \neg \mathit{done} \wedge_c B\ \mathsf{do} \\
\quad\quad \mathsf{try}\ (S\ ;\ \mathit{done} := \mathsf{true}) \\
\quad\quad \mathsf{catch}\ T\ ; \\
\quad \mathsf{if}\ \neg \mathit{done}\ \mathsf{then\ raise}
\end{array}
$$

If

$$[\![\Delta B \wedge \text{`}B\text{'} \wedge p \wedge (\lambda\,\sigma.\ v\,\sigma = w)]\!]\ S\ [\![q, h \wedge (\lambda\,\sigma.\ v\,\sigma \leq w)]\!]$$

(establishes q, or fails with h while not decreasing v)

$$[\![h \wedge (\lambda\,\sigma.\ v\,\sigma = w)]\!]\ T\ [\![p \wedge (\lambda\,\sigma.\ v\,\sigma < w)]\!]$$ (restores p and decreases v)

$$\Delta B \wedge \text{`}B\text{'} \wedge p \Rightarrow (\lambda\,\sigma.\ v\,\sigma > 0)$$ (v is positive on entrances of the loop body)

then:

$$\langle\!|p|\!\rangle\ cr\ \langle\!|q|\!\rangle$$

Proof. Let

$$
\begin{array}{l}
\mathit{loop} = \mathsf{while}\ \neg \mathit{done} \wedge_c B\ \mathsf{do} \\
\quad\quad \mathsf{try}\ (S\ ;\ \mathit{done} := \mathsf{true}) \\
\quad\quad \mathsf{catch}\ T
\end{array}
$$

and let

$$p' = \neg done \wedge p \vee done \wedge q$$

$$v' = (\lambda\,(k, \sigma).\,(k, v\,\sigma))$$

$$l' = (\lambda\,(k, n)\,(k, n').\,k < k' \vee (k = k' \wedge n < n'))$$

$$B' = \neg done \wedge_c B$$

$$S' = \mathsf{try}\ (S\ ;\ done := \mathsf{true})$$
$$\mathsf{catch}\ T$$

$$q_2' = \neg done \wedge p$$

then

$$(\forall\, w \in W.\, [\![p'_w \wedge \Delta B' \wedge \text{`}B'\text{'}]\!]\ S'\ [\![p'_{<w}, q_2']\!])$$

From Theorem 6.4, we have

$$[\![p']\!]\ \mathsf{while}\ B'\ \mathsf{do}\ S'\ [\![p' \wedge \Delta B' \wedge \neg B', (p' \wedge \neg \Delta B') \vee q_2']\!]$$

thus

$$[\![\neg done \wedge p \vee done \wedge q]\!]\ loop\ [\![\neg done \wedge p \vee done \wedge q, \neg done \wedge p]\!]$$

so

$$\langle\!|p|\!\rangle\ cr\ \langle\!|q|\!\rangle$$

□

7.3 Discussion

The correctness rules of loops allow formal reasoning about this control structure with total correctness and fail-safe correctness. The form of these correctness rules are intuitive. They give a way to reduce the reasoning about loops to the reasoning about guards and loop bodies.

The formalization of the three loop design patterns gives some further evidence on the usefulness of total correctness and fail-safe correctness notations, in addressing

"unanticipated" exceptions; its practicality subject to the category of failures that can be detected, which is not addressed here.

Chapter 8

The Retry Model of Exception Handling

The Eiffel language also has multiple exits for exception handling, but it does not use the termination model. The Eiffel exception handling mechanism is of interest because of two methodological aspects. First, it is combined with the specification of methods by pre- and postconditions that are evaluated at run-time. When a precondition does not hold, it is the caller's fault and an exception is signalled in the caller. When a postcondition does not hold, it is the callee's fault and an exception is signalled in the callee. If the callee cannot establish the desired postcondition by alternative means, the callee propagates the exception to the caller. The second methodological aspect in Eiffel is that an exception handler may *retry* a method, in which case execution continues at the beginning of a method. The rescue and retry statements combine catching an exception with a loop structure, thus requiring a dedicated form of correctness reasoning. The exception handler has to ensure that the precondition of the method holds, independently of where the exception in the body occurred, as

in following fragment:

```
meth
  require
    pre
  do
    body
  ensure
    post
  rescue
    handler
    retry
  end
```

Here, the rescue section is invoked if *post* does not hold at the end of *body*. The retry statement will restart the method, hence *handler* has to re-establish *pre*. Unlike in the termination model and the resumption model, the retrying model of exception handling leads to a loop structure [Buhr and Mok, 2000; Yemini and Berry, 1985]. In this section we are concerned with the correctness theory of exception handling in the retrying model of Eiffel.

We present verification rules for total correctness that take these two aspects into account. The rules handle normal loops and retry loop structures in an analogous manner by defining loops in terms of strong iteration. Here we have statements with three exits, *i.e.*, with three kinds of "sequential composition", one for each exit; three kinds of strong iteration are defined correspondingly. It also allows Eiffel's mechanism to be slightly generalized.

Nordio et al. [2009] proposes verification rules of Eiffel statements, but termination is not considered in the rules about the retry loop structure. We extend these rules by considering total correctness, which necessitates loop variants for normal loops and *retry variants* for methods with a retry statements. Loop variants were originally

considered in Eiffel, but not retry variants [Meyer, 1997]. Nordio et al. [2009] justifies the rules with respect to an operational semantics; here we derive the rules from a (denotational) predicate transformer semantics. Another difference is the linguistic form for retrying. Eiffel originally has a retry statement which can appear only in the exception handler. Nordio et al. [2009] proposes instead to have a *retry variable*, a Boolean variable which determines if at the end of an exception handler the body is attempted again. Tschannen et al. [2011] uses this form of retrying for translating Eiffel into the Boogie intermediate verification language. Here we consider retry statements, as they are of interest on their own and as the current versions of EiffelStudio (version 7) and SmartEiffel (version 2.3) only support retry statements. As a consequence, all statements have three exits, the normal, exceptional, and retry exit.

The work in this chapter originated in an effort to identify and formalize design patterns for exception handling; the conditional retry pattern in Section 7.2 is a simpler form of retrying in Eiffel. The formalization in this chapter covers specifically the Eiffel mechanism of retrying.

We consider a core language of statements with three exits, namely normal, exceptional, and retry exit, *i.e.*, the exit indices being $I = \{\mathsf{nrl}, \mathsf{exc}, \mathsf{ret}\}$. We also write (q_1, q_2, q_3) as an abbreviation of $\{\mathsf{nrl} \mapsto q_1, \mathsf{exc} \mapsto q_2, \mathsf{ret} \mapsto q_3\}$. On the basis of the statements defined in Chapter 3, we define some more statements that are used in Eiffel. We define raise as the abbreviation of $\mathsf{jump}\,\mathsf{exc}$, and retry as the abbreviation of $\mathsf{jump}\,\mathsf{ret}$. The *retry sequential composition* $S\ ;_{\mathsf{ret}}\ T$ continues with T on retrying termination of S.

For local variable declarations, let x_0 be the initial value of variables of type X[1]:

$\mathsf{local}\, x : X\ (q_1, q_2, q_3) \mathrel{\widehat{=}} q_1[x \backslash x_0]$

[1]Implicit state space conversion to precondition for readability.

The rescue statement $\mathsf{do}\, S\, \mathsf{rescue}\, T\, \mathsf{end}$ starts with *body* S and if S terminates normally, the whole statement terminates normally. If S terminates exceptionally, *handler* T is executed. If T terminates normally or exceptionally, the whole statement terminates exceptionally. This is captured by $U = S \;;_{\mathsf{exc}} (T \;;\; \mathsf{raise})$. If T terminates retrying, S the whole rescue statement is attempted again. Intuitively $U^{\omega_{\mathsf{ret}}} = \mathsf{skip} \sqcap U \sqcap (U \;;_{\mathsf{ret}}\; U) \sqcap (U \;;_{\mathsf{ret}}\; U \;;_{\mathsf{ret}}\; U) \ldots$ repeats zero or more times. However, $\mathsf{do}\, S\, \mathsf{rescue}\, T\, \mathsf{end}$ repeats indefinitely when T terminates retrying and may only terminate normally or retrying. This is captured by $U^{\omega_{\mathsf{ret}}} \;;_{\mathsf{ret}}\; \mathsf{stop}$, hence:

Definition 8.1. *In* do-rescue-end *statement, on retry exit the execution will be redirected to the beginning of the* do *section unconditionally, so there exists an implicit loop structure. We define:*

$$\mathsf{do}\, S\, \mathsf{rescue}\, T\, \mathsf{end} \mathrel{\widehat{=}} (S \;;_{\mathsf{exc}} (T \;;\; \mathsf{raise}))^{\omega_{\mathsf{ret}}} \;;_{\mathsf{ret}}\; \mathsf{stop}$$

This kind of exception handling differs from $\mathsf{try}\, S\, \mathsf{catch}\, T = S ;_{\mathsf{exc}} T$ in two respects: there is no loop structure in a try-catch statement and normal termination of handler T leads to normal termination of the whole statement but to exceptional termination in $\mathsf{do}\, S\, \mathsf{rescue}\, T\, \mathsf{end}$. This means that in Eiffel the handler cannot contain an alternative computation to establish the desired postcondition, but must instead direct the body S to attempt that, typically by setting a corresponding variable and retrying.

In retry model, with x being a variable and e being a relation, the *relational assignment* statement $x := e$ is defined in terms of an update statement that affects only component x of the state space. For this we assume that the state is a tuple and variables select elements of the tuple. The (Eiffel) *assignment* $x := E$, where E is now a program expression, terminates normally if E is defined, in which case the value of E is assigned to x, and terminates exceptionally if E is undefined, without changing any variables. The statement $\mathsf{check}\, B$ only evaluates Boolean expression B

without changing any variables and terminates exceptionally if B is undefined or its value is false. The statements $\mathsf{if}\ B\ \mathsf{then}\ S\ \mathsf{end}$ and $\mathsf{if}\ B\ \mathsf{then}\ S\ \mathsf{else}\ T\ \mathsf{end}$ also terminate exceptionally if B is undefined. Here we redefine these statements for Eiffel:

$$
\begin{aligned}
x := e &\mathrel{\widehat{=}} \lambda\,(x, y).\,\lambda\,(x', y').\,x' = e \wedge y' = y \\
x := E &\mathrel{\widehat{=}} [\Delta E, \neg\Delta E, \mathsf{false}]\ ;\ x := \text{`}E\text{'} \\
\mathsf{check}\ B &\mathrel{\widehat{=}} [\Delta B \wedge \text{`}B\text{'}, \neg\Delta B \vee \neg\text{`}B\text{'}, \mathsf{false}] \\
\mathsf{if}\ B\ \mathsf{then}\ S\ \mathsf{end} &\mathrel{\widehat{=}} ([\Delta B \wedge \text{`}B\text{'}, \neg\Delta B, \mathsf{false}]\ ;\ S) \sqcap \\
&\qquad [\Delta B \wedge \neg\text{`}B\text{'}, \neg\Delta B, \mathsf{false}] \\
\mathsf{if}\ B\ \mathsf{then}\ S\ \mathsf{else}\ T\ \mathsf{end} &\mathrel{\widehat{=}} ([\Delta B \wedge \text{`}B\text{'}, \neg\Delta B, \mathsf{false}]\ ;\ S) \sqcap \\
&\qquad ([\Delta B \wedge \neg\text{`}B\text{'}, \neg\Delta B, \mathsf{false}]\ ;\ T)
\end{aligned}
$$

Immediately we have that $\mathsf{check}\ B = \mathsf{if}\ B\ \mathsf{then\ skip\ else\ raise\ end}$ and $\mathsf{if}\ B\ \mathsf{then}\ S\ \mathsf{end} = \mathsf{if}\ B\ \mathsf{then}\ S\ \mathsf{else\ skip\ end}$ as consequences.

The loop $\mathsf{from}\ S\ \mathsf{until}\ B\ \mathsf{loop}\ T\ \mathsf{end}$ first executes S and then, as long as B is false, executes T, and repeats that provided T terminates normally. If S or T terminate exceptionally, the whole loop terminates immediately exceptionally. If S or T terminate retrying, the whole loop terminates immediately retrying.

$$
\begin{aligned}
\mathsf{from}\ S\ \mathsf{until}\ B\ \mathsf{loop}\ T\ \mathsf{end} \mathrel{\widehat{=}} S\ ;\ & \\
& ([\Delta B \wedge \neg\text{`}B\text{'}, \neg\Delta B, \mathsf{false}]\ ;\ T)^{\omega}\ ; \\
& [\Delta B \wedge \text{`}B\text{'}, \neg\Delta B, \mathsf{false}]
\end{aligned}
$$

Eiffel does not allow retry statements in the body S of $\mathsf{do}\ S\ \mathsf{rescue}\ T\ \mathsf{end}$. Above definition permits those, with the meaning that the whole statement is attempted again immediately.

8.1 Total Correctness

We start with two universal rules, generalizing analogous ones for single-exit statements. In a correctness assertion, the precondition can be strengthened and any of the three postconditions weakened.

$$\begin{aligned} & p' \Rightarrow p \\ & [\![p]\!]\ S\ [\![q_1, q_2, q_3]\!] \\ & (q_1 \Rightarrow q_1') \wedge (q_2 \Rightarrow q_2') \wedge (q_3 \Rightarrow q_3') \\ \Rightarrow & [\![p']\!]\ S\ [\![q_1', q_2', q_3']\!] \end{aligned}$$

Also, correctness assertions of a statement can be conjoined, thus allowing proofs to be split.

$$\begin{aligned} & [\![p]\!]\ S\ [\![q_1, q_2, q_3]\!] \\ & [\![p']\!]\ S\ [\![q_1', q_2', q_3']\!] \\ \Rightarrow & [\![p \wedge p']\!]\ S\ [\![q_1 \wedge q_1', q_2 \wedge q_2', q_3 \wedge q_3']\!] \end{aligned}$$

The first of these follows from the monotonicity of S and the second from the conjunctivity of S. The correctness rules for Eiffel statements are:

Theorem 8.1. ✓ *Let p, q_1, q_2, q_3, g_1, g_2, g_3 be predicates, B, E be program expressions, and S, T be statements:*

$$[\![p]\!]\ \mathsf{abort}\ [\![q_1, q_2, q_3]\!] \equiv p = \mathsf{false}$$

$$[\![p]\!]\ \mathsf{stop}\ [\![q_1, q_2, q_3]\!] \equiv \mathsf{true}$$

$$[\![p]\!]\ \mathsf{skip}\ [\![q_1, q_2, q_3]\!] \equiv p \Rightarrow q_1$$

$$[\![p]\!]\ \mathsf{raise}\ [\![q_1, q_2, q_3]\!] \equiv p \Rightarrow q_2$$

$$[\![p]\!]\ \mathsf{retry}\ [\![q_1, q_2, q_3]\!] \equiv p \Rightarrow q_3$$

$$[\![p]\!]\ [g_1, g_2, g_3]\ [\![q_1, q_2, q_3]\!] \equiv (p \wedge g_1 \leq q_1) \wedge (p \wedge g_2 \leq q_2) \wedge (p \wedge g_3 \leq q_3)$$

$$[\![p]\!]\ x := E\ [\![q_1, q_2, q_3]\!] \equiv (\Delta E \wedge p \leq q_1[x \backslash \text{`}E\text{'}]) \wedge (\neg \Delta E \Rightarrow q_2)$$

$$[\![p]\!]\ S\ ;\ T\ [\![q_1, q_2, q_3]\!] \equiv \exists\, h.\ [\![p]\!]\ S\ [\![h, q_2, q_3]\!] \wedge [\![h]\!]\ T\ [\![q_1, q_2, q_3]\!]$$

$$\llbracket p \rrbracket\ S\ ;_{\mathsf{exc}}\ T\ \llbracket q_1, q_2, q_3 \rrbracket \equiv \exists\, h.\ \llbracket p \rrbracket\ S\ \llbracket q_1, h, q_3 \rrbracket \wedge \llbracket h \rrbracket\ T\ \llbracket q_1, q_2, q_3 \rrbracket$$

$$\llbracket p \rrbracket\ S\ ;_{\mathsf{ret}}\ T\ \llbracket q_1, q_2, q_3 \rrbracket \equiv \exists\, h.\ \llbracket p \rrbracket\ S\ \llbracket q_1, q_2, h \rrbracket \wedge \llbracket h \rrbracket\ T\ \llbracket q_1, q_2, q_3 \rrbracket$$

$$\llbracket p \rrbracket\ S \sqcap T\ \llbracket q_1, q_2, q_3 \rrbracket \equiv \llbracket p \rrbracket\ S\ \llbracket q_1, q_2, q_3 \rrbracket \wedge \llbracket p \rrbracket\ T\ \llbracket q_1, q_2, q_3 \rrbracket$$

$$\begin{aligned}\llbracket p \rrbracket\ \mathsf{check}\ B\ \llbracket q_1, q_2, q_3 \rrbracket \equiv\ & (\Delta B \wedge \text{`}B\text{'} \wedge p \Rightarrow q_1) \wedge \\ & (\Delta B \wedge \neg\text{`}B\text{'} \wedge p \Rightarrow q_2) \wedge \\ & (\neg\Delta B \wedge p \Rightarrow q_2)\end{aligned}$$

$$\begin{aligned}\llbracket p \rrbracket\ \mathsf{if}\ B\ \mathsf{then}\ S\ \mathsf{else}\ T\ \llbracket q_1, q_2, q_3 \rrbracket \equiv\ & \llbracket \Delta B \wedge \text{`}B\text{'} \wedge p \rrbracket\ S\ \llbracket q_1, q_2, q_3 \rrbracket \wedge \\ & \llbracket \Delta B \wedge \neg\text{`}B\text{'} \wedge p \rrbracket\ T\ \llbracket q_1, q_2, q_3 \rrbracket \wedge \\ & (\neg\Delta B \wedge p \Rightarrow q_2)\end{aligned}$$

$$(\forall w \in W \cdot p_w \Rightarrow S\,(p_{<w}, q_2, q_3)) \Rightarrow (p \Rightarrow S^{\omega}\,(p, q_2, q_3))$$

$$(\forall w \in W \cdot p_w \Rightarrow S\,(q_1, p_{<w}, q_3)) \Rightarrow (p \Rightarrow S^{\omega_{\mathsf{exc}}}\,(q_1, p, q_3))$$

$$(\forall w \in W \cdot p_w \Rightarrow S\,(q_1, q_2, p_{<w})) \Rightarrow (p \Rightarrow S^{\omega_{\mathsf{ret}}}\,(q_1, q_2, p))$$

The first of these rules states that if under p_w statement S terminates normally while decreasing the rank of p_w, then under q_1 statement S terminates eventually with q_1; if S terminates exceptionally with q_2 or retrying with q_3, then S^{ω} terminates likewise. Similarly, the last of these rules states that if under p_w statement S terminates retrying while decreasing the rank of p_w, then under q_2 statement S terminates eventually with q_3; if S terminates normally with q_1 or exceptionally with q_2, then S^{ω} terminates likewise.

For the loop $\mathsf{from}\ S\ \mathsf{until}\ B\ \mathsf{loop}\ T\ \mathsf{end}$, we assume that the postconditions are of a particular form: at normal termination, the loop invariant holds, B is defined and true. At exceptional termination, either the exceptional postcondition of S or T holds (in case S or T failed), or the invariant holds and B is undefined (in case the evaluation of B failed). On retrying termination, the retrying postcondition of S or T holds (in case S or T executed retry). The role of S is to establish the loop invariant,

here p':

$$[\![p]\!]\ S\ [\![p', q_2, q_3]\!]$$
$$[\![p'_w \wedge \Delta B \wedge \neg\text{`}B\text{'}]\!]\ T\ [\![p'_{<w}, q_2, q_3]\!]$$
$$\Rightarrow [\![p]\!]\ \mathsf{from}\ S\ \mathsf{until}\ B\ \mathsf{loop}\ T\ \mathsf{end}\ [\![p' \wedge \Delta B \wedge \text{`}B\text{'}, q_2 \vee (p' \wedge \neg\Delta B), q_3]\!]$$

Recall that $p'_w = p' \wedge v = w$ where p' is the invariant, v is the variant, and $w \in W$. In Eiffel, variants are integer expressions and the well-founded set W of their values are non-negative integers. For integer variants, we have the following rule, where $w > 0$:

$$[\![p]\!]\ S\ [\![p', q_2, q_3]\!]$$
$$[\![p' \wedge v = w \wedge \Delta B \wedge \neg\text{`}B\text{'}]\!]\ T\ [\![p' \wedge v < w, q_2, q_3]\!]$$
$$\Rightarrow [\![p]\!]\ \mathsf{from}\ S\ \mathsf{until}\ B\ \mathsf{loop}\ T\ \mathsf{end}\ [\![p' \wedge \Delta B \wedge \text{`}B\text{'}, q_2 \vee (q_1 \wedge \neg\Delta B), q_3]\!]$$

Theorem 8.2. ✓ *The rule for* do S rescue T end *requires that progress towards termination is made whenever S or T exits retrying; termination here means normal termination if S terminates normally or exceptional termination if T terminates normally or exceptionally:*

$$[\![p_w]\!]\ S\ [\![q_1, t_w, p_{<w}]\!]$$
$$[\![t_w]\!]\ T\ [\![q_2, q_2, p_{<w}]\!]$$
$$\Rightarrow [\![p]\!]\ \mathsf{do}\ S\ \mathsf{rescue}\ T\ \mathsf{end}\ [\![q_1, q_2, q_3]\!]$$

For integer variants, we have following rule, where $w > 0$:

$$[\![p \wedge v = w]\!]\ S\ [\![q_1, t \wedge v = w, p \wedge v < w]\!]$$
$$[\![t \wedge v = w]\!]\ T\ [\![q_2, q_2, p \wedge v < w]\!]$$
$$\Rightarrow [\![p]\!]\ \mathsf{do}\ S\ \mathsf{rescue}\ T\ \mathsf{end}\ [\![q_1, q_2, q_3]\!]$$

Here p is the retry invariant *and v is the retry variant.*

8.2 Verification Rules

In Eiffel, each method is specified by a single precondition and single postcondition only. The normal exit is taken if the desired postcondition is established and the exceptional exit is taken if the desired postcondition cannot be established. Thus the situations under which an exceptional exit is taken is implicit in the method specification and a "defined" outcome is always possible, even in the presence of unanticipated failures. Since methods never terminate retrying, and some statements only terminate normally, we introduce two abbreviations:

$$[\![p]\!]\ S\ [\![q_1, q_2]\!] \mathrel{\widehat{=}} [\![p]\!]\ S\ [\![q_1, q_2, \mathsf{false}]\!]$$

$$[\![p]\!]\ S\ [\![q_1]\!] \mathrel{\widehat{=}} [\![p]\!]\ S\ [\![q_1, \mathsf{false}]\!]$$

In Chapter 7, we proposed to restrict the exceptional postcondition in case the specified postcondition cannot be established. Since classes typically have a class invariant, the class invariant should hold even at exceptional termination, as otherwise the program is left in an inconsistent state and a subsequent call to the same object may fail. (As a consequence, if re-establishing the class invariant cannot be guaranteed, the class invariant needs to be weakened appropriately.) More generally, let p be the condition that holds before a call to method m with body $\mathsf{local}\ x : X\ \mathsf{do}\ S\ \mathsf{rescue}\ T\ \mathsf{end}$, where p captures the computation that has been made by the whole program up to this point. We then require a call to m either to terminate normally with the desired postcondition q_1 or terminate exceptionally with p:

$$[\![p]\!]\ \mathsf{local}\ x : X\ \mathsf{do}\ S\ \mathsf{rescue}\ T\ \mathsf{end}\ [\![q_1, p]\!]$$

That is, in case of failure, the method may leave the state changed, but has to undo sufficiently such that p holds again. This regime allows then failures to be propagated back over arbitrarily many method calls. From the correctness theorems for statements, we get immediately following theorem.

Design Pattern: Repeated Attempts

Theorem 8.3. *Where p, q_1 are predicates that are independent of x and p'_w is a collection of ranked predicates, if*

$p \wedge x = x_0 \Rightarrow p'$

$\forall\, w \in W.\, [\![p'_w]\!]\ S\ [\![q'_1, t_w, p'_{<w}]\!]$ (decreases the variant on retry exit)

$\forall\, w \in W.\, [\![t_w]\!]\ T\ [\![p', p', p'_{<w}]\!]$ (decreases the variant on retry exit)

$p' \Rightarrow p$

$q'_1 \Rightarrow q_1$

Then

$\{p\}$

$\mathsf{local}\ x : X\ \mathsf{do}\ S\ \mathsf{rescue}\ T\ \mathsf{end}$

$\{q_1, p\}$

For integer variants, we have following rule, where $w > 0$:

Theorem 8.4. *Where p, q_1 are predicates that are independent of x and p'_w is a collection of ranked predicates, if*

$p \wedge x = x_0 \Rightarrow p'$

$\forall\, w > 0.\, [\![p' \wedge v = w]\!]\ S\ [\![q'_1, t \wedge v = w, p' \wedge v < w]\!]$

(decreases the variant on retry exit)

$\forall\, w > 0.\, [\![t \wedge v = w]\!]\ T\ [\![p', p', p' \wedge v < w]\!]$ (decreases the variant on retry exit)

$p' \Rightarrow p$

$q'_1 \Rightarrow q_1$

Then

$\{p\}$

$\mathsf{local}\ x : X\ \mathsf{do}\ S\ \mathsf{rescue}\ T\ \mathsf{end}$

$\{q_1, p\}$

8.3 Example: Binary Search of Square Root

Suppose the task is to compute the approximate non-negative integer square root of n, which is a non-negative integer itself, such that $\mathsf{result}^2 \leq n < (\mathsf{result}+1)^2$ using bounded arithmetic[2]. Assume that the result must be between l and u. The loop

$\mathsf{from\ until}\ u - l = 1\ \mathsf{loop}$
$\quad m := l + (u - l)\ \mathsf{div}\ 2$
$\quad \mathsf{if}\ n < m * m\ \mathsf{then}\ u := m\ \mathsf{else}\ l := m\ \mathsf{end}$
end

maintains the invariant $p \equiv 0 \leq l < u \wedge l^2 \leq n < u^2$. The statement $m := l+(u-l)\ \mathsf{div}\ 2$ will establish $m = (l+u)\ \mathsf{div}\ 2$ and never fail, according to Theorem 3.9. However, the if statement will fail if $m*m > \mathsf{max}$. Since necessarily $n \leq \mathsf{max}$, we know that in case of failure $n < m * m$, thus after assigning $u := m$ the loop can continue. We use the abbreviation $\{\mathsf{ret} : q_1\}$ for $\{\mathsf{false}, \mathsf{false}, q_1\}$. The full implementation with annotation is as follows:

$sqrt(n, l, u : INTEGER) : INTEGER$
$\quad \{p\}$
$\quad \mathsf{local}$
$\quad\quad m : INTEGER$
$\quad \{\text{retry invariant: } p\}$

[2]The Eiffel Standard [Ecma International, 2006] and Meyer [1997] suggest that an arithmetic overflow leads to an exception. SmartEiffel (version 2.3) does raise an exception, but EiffelStudio (version 7) does not. However, the example can be expressed in EiffelStudio by first formulating a class for safe arithmetic, see `http://www.cas.mcmaster.ca/~zhangt26/thesis/`

$\{$retry variant: $u - l\}$
do
  $\{$loop invariant: $p\}$
  $\{$loop variant: $u - l\}$
  from until $u - l = 1$ loop
    $m := l + (u - l)$ div 2
    $\{p \wedge m = (l + u)$ div $2\}$
    if $n < m * m$ then $u := m$ else $l := m$ end
    $\{p, p \wedge m = (l + u)$ div $2 \wedge n < m^2\}$
  end
  $\{p \wedge u - l = 1\}$
  result $:= l$
rescue
  $\{p \wedge m = (l + u)$ div $2 \wedge n < m^2\}$
  $u := m$
  $\{p\}$
  retry
  $\{$ret : $p\}$
end
$\{$result$^2 \leq n <$ (result $+1)^2\}$

Note that the retry loop only needs to decrease the variant on the retry exit.

8.4 Discussion

In this chapter we have derived verification rules for the retrying mechanism of Eiffel exceptions. Beside the contribution of total correctness rules, the novel aspects of the derivation are that we started with a weakest exceptional precondition semantics and defined both normal loops and retry loops through strong iteration. We did not study the fail-safe correctness and fail-safe refinement of Eiffel since the retry mechanism was the main focus.

The statements considered include the check statement, but we have not discussed ensure and require method specifications. Since these are evaluated at run-time in Eiffel, they are restricted to be program expressions (extended with the **old** notation). However, since these are evaluated program expressions they have be treated like the check statement. It is straightforward to extend the approach for method correctness accordingly.

We have neither considered dynamic objects, therefore no method calls, nor other features of Eiffel like inheritance. However exception handling is largely independent of other features and the treatment here carries over to a more general setting. We have not considered fail-safe correctness and fail-safe refinement in this retry model, since our main focus is the retry loop.

Strong and weak iteration are appealing because of their rich algebraic structure. However, we have not explored the resulting algebraic properties of rescue and retry statements. For example, following theorems can be shown to hold:

$$\mathsf{do\ skip\ rescue}\ S\ \mathsf{end} = \mathsf{skip}$$

$$\mathsf{do\ raise\ rescue\ retry\ end} = \mathsf{abort}$$

An interesting consequence of our definition of statements is that retry statements can also appear in the main body of a method, not only the exception handler. Theorem 8.2 supports this use. With this, the binary search of the square root example can be rewritten without the from / until loop, using only the retry loop:

$sqrt2(n, l, u : INTEGER) : INTEGER$
  $\{p\}$
  local
    $m : INTEGER$
  {retry invariant: p}
  {retry variant: $u - l$}
  do

$m := l + (u - l)$ div 2
$\{p \wedge m = (l + u)$ div $2\}$
if $n < m * m$ then $u := m$ else $l := m$ end
$\{p, p \wedge m = (l + u)$ div $2 \wedge n < m^2\}$
if $u - l > 1$ then retry end
$\{p \wedge u - l = 1, \mathsf{ret} : p \wedge u - l > 1\}$
result $:= l$
rescue
$\{p \wedge m = (l + u)$ div $2 \wedge n < m^2\}$
$u := m$
$\{p\}$
retry
$\{\mathsf{ret} : p\}$
end
$\{\mathsf{result}^2 \leq n < (\mathsf{result} + 1)^2\}$

Nordio et al. [2009] proposes to replace the retry statement with a retry variable in order to avoid the third exit. Below is their example of safe division, with annotation to show termination of the retry loop; the example shows that the third exit does not cause further complications:

safe_division $(x, y : INTEGER) : INTEGER$
local
$z : INTEGER$
{retry invariant: $(y \neq 0 \wedge z = 0) \vee (y = 0 \wedge (z = 1 \vee z = 0))$}
{retry variant: $1 - z$}
do
result $:= x$ div$(y + z)$
$\{(y = 0 \Rightarrow \mathsf{result} = x) \wedge (y \neq 0 \Rightarrow \mathsf{result} = x$ div $y), y = 0 \wedge z = 0\}$
rescue
$\{y = 0 \wedge z = 0\}$
$z := 1$
$\{y = 0 \wedge z = 1\}$
retry

$\{\mathsf{ret} : y = 0 \wedge z = 1\}$

end

$\{(y = 0 \Rightarrow \mathsf{result} = x) \wedge (y \neq 0 \Rightarrow \mathsf{result} = x \,\mathsf{div}\, y)\}$

Chapter 9

Coroutines

9.1 Overview

Conway [1963] proposes the concept of coroutines, which is defined as "subroutines all at the same level, each acting as if it were the master program when in fact there is no master program". The entry point of a subroutine is always its beginning, while the entry point of the main routine or a coroutine is always the point where it last exited [Knuth, 1968].

Coroutines are closely related to multiple-pass algorithms, which have better comprehensibility and lower space requirements compared to one-pass algorithms [Knuth, 1968]. Applications of coroutines include business data processing, text processing, simulation, and various kinds of data structure manipulation [Marlin, 1980].

Coroutines have been implemented in various programming languages, *e.g.*, Simula [Dahl and Nygaard, 1966], Modula-2 [Gurevich and Morris, 1988], Python [van Rossum and Eby, 2011], fibers in .NET [Shankar, 2003], *etc.* However, most of them implement coroutines on top of existing programming structures, which in general prevents making full use of coroutines [Barclay, 2009].

Unlike subroutines, which have a unified basic semantics across languages, the semantics of coroutines vary in implementations. With respect to control transfer mechanism, coroutine facilities can be divided into completely symmetric ones as in Simula [Dahl and Nygaard, 1966], and asymmetric ones as in Python [van Rossum and Eby, 2011]. As pointed out by [Moura and Ierusalimschy, 2009], symmetric coroutine mechanisms provide a single control-transfer operation for resume/resume mechanism, in which coroutines can resume each other; asymmetric ones provide two control-transfer operation for call/detach mechanism, in which a caller calls a coroutine and the callee can detach during execution. Coroutine implementations also vary on first-classness (a coroutine being a first-class object) and stackfulness (a coroutine being able to be suspended within its nested calls).

Moura and Ierusalimschy [2009] also prove that symmetric coroutines and asymmetric ones are equally powerful with respect to theoretical expressivity, while asymmetric coroutines are easier to manage and understand. However, first-classness as well as stackfulness can enhance the expressive power.

A generator is a coroutine that returns a value each time [van Rossum and Eby, 2011]. Generators are commonly used as implementation of iterators. Generators and coroutines can simulate each other by using returning variables to replace common variables and vice versa.

Below is a generator of the Fibonacci sequence. In the generator body, the yield statement gives the control back to the caller, and when the caller calls the coroutine again, it is resumed right after the last exit point.

```
def fg() :
    x, y = 0, 1
    for x in range(9) :
        yield x
        x, y = x + y, x
print(list(fg()))
```

Running the program would output "[0, 1, 1, 2, 3, 5, 8, 13, 21]" as the result.

Clint [1973] and Clarke [1980] prove the correctness of asymmetric coroutines using one core rule:

$$\frac{R'\{\mathsf{exit}\ c\}P' \vdash P\{Q_1\}R \qquad P'\{\mathsf{call}\ c\}R' \vdash R'\&P\{Q_2\}R}{P\{(x)\ c\ \mathsf{coroutine}\ Q_1; Q_2\}R}$$

Here P' must hold whenever the coroutine is entered, and R' must hold whenever the coroutine exits. It is similar to loop invariants and *rely/guarantee conditions* in $\pi o\beta\lambda$ [Collette and Jones, 1995]. Clint [1973] uses history variables in proofs, and Clarke [1980] shows how to avoid using history variables in proofs of simple coroutines. However, their verification rules do not consider termination. The correctness of the rule itself relies more on intuition than formal semantics. In this chapter we attempt to consider termination by applying total correctness rules for loop, and developing a new semantics of coroutine on the basis of data refinement.

9.2 Our Coroutine Mechanism

For simplicity, we choose a language that excludes exception handling and nested coroutine calls. The coroutines in our language are:

1. Asymmetric (caller and callee). We assume that subroutines can all create

instances of a coroutine, but each subroutine has exclusive access to all coroutine instances created by itself.

2. Non-first-class (coroutines not treated as first-class objects), for simplicity.

3. Non-stackful (cannot be suspended from within nested functions), since nested coroutine calls are not allowed.

In this chapter we do not consider exception handling, so we only use value expressions. Also we only consider coroutines with only one yield statement, which is the most common usage in practice. It simplifies the verification rules, and the potential of extending our verification rules to coroutines with multiple yield statement will be discussed at the end of this chapter.

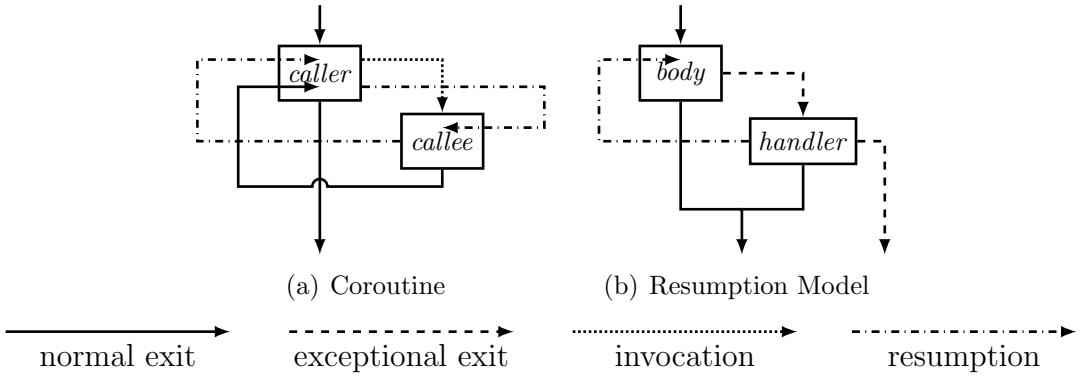


Figure 9.1: Coroutines and Resumption Model

Coroutines and the resumption model of Mesa [Mitchell et al., 1979] allow entrance of a program block in the middle, as shown in Figure 9.1. Consequently, the isolation between a program block and its context is breached, and to reason about the correctness we will need a different approach for the specification and verification of coroutines.

First we define our own notations of subroutines and coroutines as:

```
subroutine subroutine name
  local local variable list
  common common variable list
  create instance name : coroutine name
  ...
  invoke coroutine name
  ...
end
coroutine coroutine name
  local variable list
  common common variable list
  ...
  yield
  ...
end
```

No parameter is passed in our coroutine mechanism. Result parameters can be simulated by declaring them as common variables and initializing them at the beginning of the coroutine. One implicit restriction is that the common variables of a caller and of its callees must match in types.

9.3 Verification

We start by studying two programs with coroutines, both for summing up all elements in an read-only global array a:

```
subroutine sum1
  local int i := 0
  common s := 0
  create do : dosum1
```

```
subroutine sum2
  local int i := 0
  common s := 0
  create do : dosum2
```

```
    while i < len(a) do
      invoke do
      i := i + 1
  end
  coroutine dosum1
    local int j := 0
    common s
    while j < len(a)
      j, s := j + 1, s + a[j]
      yield
  end
```

```
    while i < len(a) do
      invoke do
      i := i + 1
  end
  coroutine dosum2
    local int j, ss := 0, 0
    common s
    while j < len(a)
      j, ss := j + 1, ss + a[j]
      yield
    s := ss
  end
```

The expected results are the same: assigning the summation to the common variable s on the termination of $sum1$ or $sum2$. However, their intermediate states are different: with rules in [Clint, 1973] and Clarke [1980], to guarantee the correctness, $sum1$ only needs to know that for any $n \in [0..len(a))$:

$$[\![i = n \wedge s = \sum x \in [0..i) \cdot a[x]]\!]\ dosum1\ [\![i = n \wedge s = \sum x \in [0..i] \cdot a[x])]\!]$$

Similarly, $sum2$ only needs to know that for any $n \in [0..len(a))$:

$$[\![i = n \wedge ss = \sum x \in [0..i) \cdot a[x]]\!]\ dosum2\ [\![i = n \wedge ss = \sum x \in [0..i] \cdot a[x]]\!]$$

However, this makes ss visible to the caller, which reveals the implementation details of $dosum2$. These two papers assume that a subroutine is bounded to only one coroutine, so it is acceptable to have the local variables of coroutine visible to the subroutine. However, as we want to generalize our mechanism to allow a coroutine to be reused by multiple subroutines, each with their own instance, and to allow a routine to create and call instances of multiple coroutines, proper information hiding is necessary. Besides, we want to have verification rules that takes termination into consideration. The first goal is from the caller's perspective: the local state of the

callee should not be directly visible, but some knowledge must be available (as necessary in the second solution); the second goal is from the callees' perspective: the coroutine needs to show its total correctness to the caller.

Assuming that the state spaces of caller local variables, of common variables, and of callee local variables are Σ, Σ_c, and Σ' respectively. In this chapter we assume all the initialization of local variables is done at the beginning of subroutines and coroutines, to reduce the effort of formalizing state spaces and their conversions. Thus the state space of the subroutine is (Σ, Σ_c), the coroutine is a predicate transformer on (Σ_c, Σ'), and only its effect on Σ_c is visible to the caller.

Caller's Perspective

Single-exit predicate transformer semantics suffices for reasoning about the correctness of the caller. It is an special case of indexed predicate transformer semantics in Chapter 3 with writing q as the abbreviation of Q on domain $\{\mathsf{nrl} \mapsto q\}$.

The total correctness theorems of single-exit statements are as following:

Theorem 9.1. *Let p, q, g be predicates on Σ, b, e be value functions, and S, T be statements:*

$$\begin{aligned}
[\![p]\!]\ \mathsf{abort}\ [\![q]\!] &\equiv p = \mathsf{false} \\
[\![p]\!]\ \mathsf{stop}\ [\![q]\!] &\equiv \mathsf{true} \\
[\![p]\!]\ \mathsf{skip}\ [\![q]\!] &\equiv p \Rightarrow q \\
[\![p]\!]\ x := e\ [\![q]\!] &\equiv p \leq q[x \backslash e] \\
[\![p]\!]\ S\ ;\ T\ [\![q]\!] &\equiv \exists\, h.\ [\![p]\!]\ S\ [\![h]\!] \wedge [\![h]\!]\ T\ [\![q]\!] \\
[\![p]\!]\ S \sqcap T\ [\![q]\!] &\equiv [\![p]\!]\ S\ [\![q]\!] \wedge [\![p]\!]\ T\ [\![q]\!] \\
[\![p]\!]\ \mathsf{if}\ b\ \mathsf{then}\ S\ \mathsf{else}\ T\ [\![q]\!] &\equiv [\![b \wedge p]\!]\ S\ [\![q]\!] \wedge [\![\neg b \wedge p]\!]\ T\ [\![q]\!]
\end{aligned}$$

The *while loop* $\mathsf{while}\ b\ \mathsf{do}\ S$, in which b is a Boolean value function and S is a

statement, is defined as the least fixed point of $\lambda X.\ \mathsf{if}\ b\ \mathsf{then}(S\ ;\ X)$ with respect to the refinement ordering, *i.e.*,

$$\mathsf{while}\ b\ \mathsf{do}\ S \mathrel{\widehat{=}} (\mu X \cdot \mathsf{if}\ b\ \mathsf{then}(S\ ;\ X))$$

Theorem 9.2. *Assume that b is a Boolean value expression, p is a predicate and S is a statement. Assume that p_w for $w \in W$ is a ranked collection of predicates. Then*

$$\forall\, w \in W.\, [\![p_w \wedge b]\!]\ S\ [\![p_{<w}]\!] \Rightarrow [\![p]\!]\ \mathsf{while}\ b\ \mathsf{do}\ S\ [\![p \wedge \neg b]\!]$$

Now the only missing piece in the puzzle is the semantics of invoke. To reason about the semantics of coroutines, one possible method is to map all the states to a global state space $(\Sigma, \Sigma_c, \Sigma')$, and project all predicate transformers to the space. However, since in this part we reason from the caller's perspective, this would again reveal implementation details of the coroutine, when it is necessary to state that the coroutine local state is maintained through in the subroutine. Besides, when extended to reasoning with stackful coroutines, the global state space would explode and be too complicated to model and reason about.

We propose an alternative. When the details of the callees are hidden, the caller knows their effect on the common variables, and it also knows how many invocations have been completed, a ghost variable IC, as the invocation counter. It is initialized to be 0 and increased by 1 during each invocation. We consider a IC common variable (part of Σ_c), since it is readable to the caller and writeable to the callee. Since the caller knows what the callee is supposed to do each time, there is a relation connecting common variables and coroutine local variables, $r \in \Sigma_c \leftrightarrow (\Sigma_c, \Sigma')$, which should be a bijection. Then from the caller's point of view, all it needs to know is:

$$\forall\, n \geq 0.\, [\![p \wedge IC = n]\!]\ \mathsf{invoke}\ co\ [\![q \wedge IC = n + 1]\!]$$

For example, any program that calls the Fibonacci sequence generator, only needs to

know is that (assume F is the Fibonacci sequence):

$$\forall\, n \geq 0.\, [\![\mathsf{result} = F(n) \wedge IC = n]\!]\ \mathsf{invoke}\, fg\ [\![\mathsf{result} = F(n+1) \wedge IC = n+1]\!]$$

Then we only need to prove that the callee, which is a predicate transformer on (Σ_c, Σ'), is a data refinement of the specification through some relation r, which is visible to the callee but invisible to the caller.

Callee's Perspective

The semantics of coroutines is more complicated as yield is present, which leads to two exits: coroutines exit normally at the end, or by the yield statements. Correspondingly, there are two entries: a coroutine is entered normally at the beginning, or resumed after the yield statement. However, this is transparent to the caller. Thus, we consider each invocation of a coroutine as single-entry single-exit to the caller, similar to a subroutine, only that the semantics can vary on different invocations. Besides, the program counter of the last exit point, should be invisible to the caller. We use an implicit variable EL, local to the coroutine, as the exit label that records the last exit point. With l possible exit points, the range of EL is $0..l+1$, with 0 as the initial value, implying normal entrance (starting), $1..l$ for the l possible exit points, and $l+1$ for normal termination. In this thesis we only consider coroutines with only one yield statement for simplicity, so $l = 1$ and $EL \in 0..2$.

To verify that a coroutine implements a coroutine specification, we need to define the semantics of a coroutine body first. We formalize a coroutine statement as a predicate pair transformer, *i.e.*, of type $\mathcal{P}\Sigma \times \mathcal{P}\Sigma \to \mathcal{P}\Sigma \times \mathcal{P}\Sigma$. For each q_1 and q_2, $S\,(q_1, q_2)$ calculates the weakest preconditions p_1 and p_2, such that if p_1 holds on normal entry of S or if p_2 holds on resume entry of S, then S either terminates normally with q_1 or terminates on resume exit with q_2.

We assume that $fst\,(q_1, q_2) \mathrel{\widehat{=}} q_1$ and $snd\,(q_1, q_2) \mathrel{\widehat{=}} q_2$ being the first and second element of a pair. The coroutine version of sequential composition connects on two exits to two entries:

$$S \,;_{\mathsf{c}}\, T \mathrel{\widehat{=}} \lambda\,(q_1, q_2).\, S\,(T\,(q_1, q_2))$$

To allow skipping resume entry, assignments are defined using single-entry single-exit predicate transformer:

$$x :=_{\mathsf{c}} e \mathrel{\widehat{=}} \lambda\,(q_1, q_2).\,((x := e)\,q_1, q_2)$$

To encapsulate the two exits and make the coroutine only appear as single-entry single-exit predicate transformer to the caller, we add cobegin and coend statements at the beginning and the end of a coroutine. On the entry of a coroutine, cobegin will switch the execution to the resume exit unless $EL = 0$. On resume entry, most statements will do nothing and terminate on resume exit, until an yield statement catches the control flow $EL = 1$ and switch to normal exit. After that the execution continues on normal exit, and will be switched to resume exit on the next yield statement encountered. At the end of the coroutine, a coend statement always switches to normal exit.

$$\mathsf{cobegin} \mathrel{\widehat{=}} \lambda\,(q_1, q_2).\,((\lambda\,\sigma.\, EL = 0) \wedge q_1 \vee (\lambda\,\sigma.\, EL \neq 0) \wedge q_2)$$

$$\mathsf{coend} \mathrel{\widehat{=}} \lambda\,(q.\,(EL := 2)\,(q, q))$$

Additionally, since we do not consider exceptional handling, we assume that a coroutine does not change the state when resumed after termination ($EL = 2$).

After being encapsulated by cobegin and coend, there is a normal entry and two exits. The resume exit being blocked by coend, which makes it single-entry single-exit predicate transformer. Then with proper relation r, we can define the semantics of invoke as:

$$\mathsf{invoke}\; co = [r]\,;\mathsf{cobegin}\,;_{\mathsf{c}}\, co\,;_{\mathsf{c}}\,\mathsf{coend}\,;\{r^{-1}\}$$

Here IC (combined with the common variables) serves like a backup of the coroutine's local state. $[r]$ restores the coroutine local state, and $\{r^{-1}\}$ does the "backup" again.

The idea is similar to data refinement in Chapter 6.

The yield statement is the only point where the control switches from normal to resume or vice versa. On normal entry, yield redirects the control to the resume exit unconditionally. On resume exit, yield redirects to normal entry when $EL = 1$. At the beginning of the coroutine, an additional cobegin statement will direct the control to the resume exit. Then statements will be skipped until a coend statement redirects the control back to normal exit unconditionally.

$$\mathsf{yield} \mathrel{\widehat{=}} EL := 1 \,;_{\mathsf{c}} (\lambda\,(q_1, q_2).\,(q_2, (q_1 \wedge EL = 1) \vee q_2))$$

The control flow of some coroutine statements is illustrated in Figure 9.2:

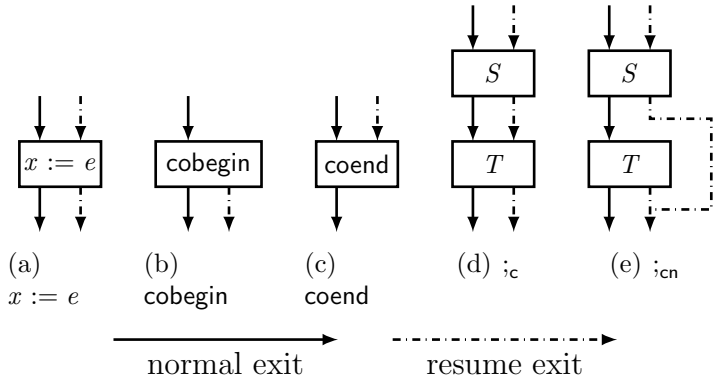


Figure 9.2: Flowcharts of Coroutine Statements

Then we formalize conditional statements. On resume entry of $\mathsf{if_c}\ b\ \mathsf{then_c}\ S$, the execution is pass on to the resume entry of S. If it contains the yield statement, then it would be caught when $EL = 1$, otherwise skipped.

$$\mathsf{if_c}\ b\ \mathsf{then_c}\ S \mathrel{\widehat{=}} \lambda\,(q_1, q_2).\,(b \Rightarrow \mathit{fst}\,(S\,(q_1, q_2)) \wedge \neg b \Rightarrow q_1, \mathit{snd}\,(S\,(q_1, q_2))$$

The loop is again defined on the basis of conditional statement and least fixed point:

$$S \;;_{\mathsf{cn}}\; T \mathrel{\widehat{=}} \lambda\,(q_1, q_2).\,S\,(\mathit{fst}(T\,(q_1, q_2)), q_2)$$

$$\mathsf{while_c}\ b\ \mathsf{do_c}\ S \mathrel{\widehat{=}} (\mu X \cdot \mathsf{if_c}\ b\ \mathsf{then_c}(S \;;_{\mathsf{cn}}\; X))$$

The definition of loop uses $;_{\mathsf{cn}}$, which connects two statements on normal entry and

exit only, instead of $;_{\mathsf{c}}$. The modification is to let the yield statement terminate the loop on resume exit. If $;_{\mathsf{c}}$ would be used, when the loop body terminates on resume exit, on the next iteration it will be caught on the resume entry of the yield statement again, then the loop can be non-terminatingsince it cannot exit properly on yield statements. So by using $;_{\mathsf{cn}}$, at the end of each iteration, if the execution is on resume exit, then the loop terminates on resume exit immediately, instead of continuing to the next iteration on resume entry. We assume that $;_{\mathsf{c}}$ has higher priority than $;_{\mathsf{cn}}$, which is useful to bind the loop body tighter.

The proof about the loop does not require rules about ranked predicates. Using the least fixed point property $(\mu x \cdot f\ x) = f\ (\mu x \cdot f\ x)$, we can unwind the loop and reason. For normal cases in which the loop body yield each time, unwinding twice is enough.

Now we take a look back at the second solution $sum2$. The correctness of the subroutine is straightforward:

subroutine $sum2$
  local int $i := 0$
  common $s := 0$
  create $do : dosum2$
  $\{$loop invariant : $i \leq len(a) \wedge i = IC \wedge (i < len(a) \Rightarrow s = 0) \wedge$
    $(i = len(a) \Rightarrow s = \sum x \in [0..len(a)) \cdot a[x]))\}$
  while $i < len(a)$ do
    $\{i < len(a) \wedge i = IC \wedge s = 0\}$
    invoke do
    $\{i = IC - 1 \wedge (IC < len(a) \Rightarrow s = 0) \wedge$
      $(IC = len(a) \Rightarrow s = \sum x \in [0..len(a)) \cdot a[x]))\}$
    $i := i + 1$
  $\{i = len(a) \wedge s = \sum x \in [0..len(a)) \cdot a[x])\}$
end

The correctness of invoke *do* relies on reasoning inside the coroutine. Here we have *dosum*2 with three versions of proof notes: on starting (normal exit of cobegin), during execution (resume exit of cobegin and resume entry of coend), and on terminating of the coroutine (normal entry of coend).

The bijection relation between subroutine state and coroutine state is

$$r = \lambda\,(IC, s)(s', j, ss, EL).\, IC = j \wedge s = s' \wedge ss = \sum x \in [0..j) \cdot a[x] \wedge$$
$$(j = 0 \Rightarrow EL = 0) \wedge (0 < j < len(a) \Rightarrow EL = 1) \wedge (j = len(a) \Rightarrow EL = 2)$$

On normal entry $len(a) > 0$ can be deduced from the precondition of invoke.

$[\![IC = 0 \wedge s = 0]\!]$
$[r]\,;$
$\{s = 0 \wedge j = 0 \wedge ss = 0 \wedge EL = 0\}$
cobegin $;_{\mathsf{c}}$
$\{s = 0 \wedge j = 0 \wedge ss = 0 \wedge EL = 0, \mathsf{false}\}$
$\mathsf{if_c}\ j < len(a)\ \mathsf{then_c}$
  $\{s = 0 \wedge j = 0 \wedge ss = 0 \wedge EL = 0, \mathsf{false}\}$
  $j, ss := j + 1, ss + a[j]\ ;_{\mathsf{c}}$
  $\{s = 0 \wedge j = 1 \wedge ss = \sum x \in [0..j) \cdot a[x] \wedge EL = 0, \mathsf{false}\}$
  yield $;_{\mathsf{cn}}$ (switch)
  $\{\mathsf{false}, s = 0 \wedge j = 1 \wedge ss = \sum x \in [0..j) \cdot a[x] \wedge EL = 1\}$
  while $j < len(a)$ (skipped)
    $j, ss := j + 1, ss + a[j]$ (skipped)
    yield (skipped)
$s := ss$ (skipped)
$\{\mathsf{false}, s = 0 \wedge j = 1 \wedge ss = \sum x \in [0..j) \cdot a[x] \wedge EL = 1\}$
coend
$\{s = 0 \wedge j = 1 \wedge ss = \sum x \in [0..j) \cdot a[x] \wedge EL = 1\}$
$\{r^{-1}\}$
$[\![IC = 1 \wedge s = 0]\!]$

For any $0 < n < len(a)$,

$[\![IC = n \wedge s = 0]\!]$
$[r]\,;$
$\{s = 0 \wedge j = n \wedge ss = \sum x \in [0..j) \cdot a[x] \wedge EL = 1\}$
$\mathsf{cobegin}\ ;_{\mathsf{c}}$
$\{\mathsf{false}, s = 0 \wedge j = n \wedge ss = \sum x \in [0..j) \cdot a[x] \wedge EL = 1\}$
$\mathsf{if}_{\mathsf{c}}\ j < len(a)\ \mathsf{then}_{\mathsf{c}}$ (skipped)
  $j, ss := j + 1, ss + a[j]\ ;_{\mathsf{c}}$ (skipped)
  $\mathsf{yield}\ ;_{\mathsf{cn}}$ (switch)
  $\{s = 0 \wedge j = n \wedge ss = \sum x \in [0..j) \cdot a[x] \wedge EL = 1, \mathsf{false}\}$
  $\mathsf{if}_{\mathsf{c}}\ j < len(a)\ \mathsf{then}_{\mathsf{c}}$
    $j, ss := j + 1, ss + a[j]\ ;_{\mathsf{c}}$
    $\{s = 0 \wedge j = n + 1 \wedge ss = \sum x \in [0..j) \cdot a[x] \wedge EL = 1, \mathsf{false}\}$
    $\mathsf{yield}\ ;_{\mathsf{cn}}$ (switch)
    $\{\mathsf{false}, s = 0 \wedge j = n + 1 \wedge ss = \sum x \in [0..j) \cdot a[x] \wedge EL = 1\}$
    $\mathsf{while}_{\mathsf{c}}\ j < len(a)\ \mathsf{do}_{\mathsf{c}}$ (skipped)
      $j, ss := j + 1, ss + a[j]\ ;_{\mathsf{c}}$ (skipped)
      $\mathsf{yield}\ ;_{\mathsf{cn}}$ (skipped)
$\{\mathsf{false}, s = 0 \wedge j = n + 1 \wedge ss = \sum x \in [0..j) \cdot a[x] \wedge EL = 1\}$
$s := ss$ (skipped)
coend
$\{s = 0 \wedge j = n + 1 \wedge ss = \sum x \in [0..j) \cdot a[x] \wedge EL = 1\}$
$\{r^{-1}\}$
$1[\![IC = n + 1 \wedge s = 0]\!]$

For normal termination of the coroutine:

$[\![IC = len(a) \wedge s = 0]\!]$
$[r]\,;$
$\{s = 0 \wedge j = len(a) \wedge ss = \sum x \in [0..j) \cdot a[x] \wedge EL = 1\}$
$\mathsf{cobegin}\ ;_{\mathsf{c}}$
$\{\mathsf{false}, s = 0 \wedge j = len(a) \wedge ss = \sum x \in [0..j) \cdot a[x] \wedge EL = 1\}$
$\mathsf{if}_{\mathsf{c}}\ j < len(a)\ \mathsf{then}_{\mathsf{c}}$ (skipped)
  $j, ss := j + 1, ss + a[j]\ ;_{\mathsf{c}}$ (skipped)
  $\mathsf{yield}\ ;_{\mathsf{cn}}$ (skipped)

$\mathsf{while_c}\ j < len(a)\ \mathsf{do_c}$ (skipped)
$j, ss := j + 1, ss + a[j]\ ;_{\mathsf{c}}$ (skipped)
$\mathsf{yield}\ ;_{\mathsf{cn}}$ (skipped)
$\{s = 0 \wedge j = len(a) \wedge ss = \sum x \in [0..j) \cdot a[x] \wedge EL = 1, \mathsf{false}\}$
$s := ss$
$\{s = 0 \wedge j = len(a) \wedge ss = \sum x \in [0..j) \cdot a[x] \wedge EL = 1, \mathsf{false}\}$
coend
$\{s = ss \wedge j = len(a) \wedge ss = \sum x \in [0..j) \cdot a[x] \wedge EL = 2\}$
$\{r^{-1}\}$
$[\![IC = len(a) \wedge s = \sum x \in [0..j) \cdot a[x]]\!]$

9.4 Discussion

In this section we studied the verification rules of coroutines. We not only presented the rules, but also showed how the rules themselves can be deducted from formal semantics. Our rules take termination into consideration.

A possible extension is to consider coroutines with multiple yield statements. However, in that case it would be necessary to define the syntax of coroutine, since the labelling of yield statements would be dependent on that. Then the semantics of each yield needs to be conditional on comparison of the current EL and the label of the yield statement.

Another possible extension is to introduce nested coroutine calls and stackfulness. The double-entry double-exit internal representation can be convenient, because each coroutine call inside a coroutine can be directly treated as a double-entry double-exit predicate transformer, with its own exit label.

Chapter 10

Conclusion

10.1 Multi-Exit Programs

The work in this thesis models the semantics of multi-exit programs as indexed predicate transformers, studies its algebraic properties and verification rules, then instantiate them for several semantic models: the termination model in exception handling, the retry model in exception handling, and a coroutine model. The instantiated verification rules are illustrated with examples, and the termination model is illustrated with six design patterns.

An alternative of using indexed predicate transformers is to encode the exit index into the state and model programs as predicate transformers with the state as the argument, which we have not pursued. The advantage of indexed predicate transformers is that the state spaces on exits can be different. This is useful for statements like local variable declaration, which either succeed and add variables into the state space, or fail and leave the state space unchanged. Besides, having separate postconditions may be methodologically stronger and syntactically shorter.

10.2 Future Work

Our work could be extended in several aspects.

1. The normal form of disjunctive statements is unimplementable, but nevertheless of theoretical interest.

2. Fail-safe refinement rules for loops and recursive procedures can be useful. Besides, common programming languages provide named exceptions and statements with several exits. A generalization of fail-safe refinement to multiple exits is worth exploring.

3. Data refinement in Chapter 6 is a generalization of *forward data refinement* to two exits. Forward data refinement is known to be incomplete; for the predicate transformer model of statements, several alternatives have been studied, *e.g.*, [Gardiner and Morgan, 1993; von Wright, 1994]. It remains to be explored how these can be used for partial data refinement.

4. For simplicity of the theory itself, we directly define the semantics in most chapters, and the examples are converted by hand in formalization; only in Chapter 9 we defined the syntax (which is a mixture of syntax and logical terms) and the conversion from syntax to semantics. It would facilitate the conversion of examples if a full conversion is formalized.

5. Also, our formalization of coroutines allows it to be supplemented with an additional exit for exception handling. Adding the support of stackfulness is also useful for implementing recursive algorithms like traversal of trees.

Appendix A

Isabelle Formalization

The theorems and patterns are formalized and verified in Isabelle 2011-1, in which the sets and predicates are of the same type, so all the predefined theorems of sets are directly available. However, from Isabelle 2012 on, the set is no longer typed as a mapping to Boolean type, so we chose not to migrate.

The Isabelle mechanization uses shallow embedding, which defines the semantics directly. It follows the formalization in this thesis closely, however there are two differences. First, some notations are different, either to distinguish from other notations (to speed up the automatic reasoning for implicit types in Isabelle), *e.g.*, using $[[R]]$ in Isabelle for demonic update $[R]$ to distinguish from $[G]$, which represents assumption $[G]$. Secondly, the formalization of indexed predicates and indexed relations in Chapter 3 allows the state spaces on arbitrary number of exits to be different. To model arbitrary number of potentially different state spaces, dependent types are required. However, Isabelle does not support this feature, so in the Isabelle proofs, for theorems in Chapters 3, 4, and 5, we model arbitrary number of exits with the same state space; for theorems in Chapters 6, 7, 8 and 9, we model potentially different state spaces on exits of a fixed number. The proofs in the thesis do not rely on the uniformity of the

final state spaces, unless restricted by certain constructs, *e.g.*, loop on exit i requires the initial state space to be the same as the state space on exit i.

The complete Isabelle formalization is available at `http://www.cas.mcmaster.ca/~zhangt26/thesis/`. It contains more than 500 theorems.

Bibliography

Abrial, J.-R. (1996). *The B-book: assigning programs to meanings*. New York, NY, USA: Cambridge University Press.

Abrial, J.-R., M. K. O. Lee, D. Neilson, P. N. Scharbach, and I. H. Sørensen (1991). The B-method. In *VDM Europe (2)*, pp. 398–405.

AMD (2012, September). AMD64 architecture programmer's manual volume 3: General-purpose and system instructions. `http://support.amd.com/us/Embedded_TechDocs/24594.pdf`.

ANSI (1966). ANSI Fortran X3.9-1966.

ANSI (1997). ANSI/ISO/IEC 1539-1:1997: Information technology — programming languages — Fortran — part 1: Base language.

Back, R. (1981). On correct refinement of programs. *Journal of Computer and System Sciences 23*(1), 49 – 68.

Back, R. and J. Wright (1990). Refinement concepts formalised in higher order logic. *Formal Aspects of Computing 2*(1), 247–272.

Back, R.-J. and J. von Wright (1998). *Refinement Calculus: A Systematic Introduction*. Springer-Verlag.

Back, R.-J. R. and M. Karttunen (1983). A predicate transformer semantics for statements with multiple exits.

Banach, R., M. Poppleton, C. Jeske, and S. Stepney (2007). Engineering and theoretical underpinnings of retrenchment. *Science of Computer Programming 67*(2–3), 301 – 329.

Barclay, D. (2009, April). A language of coroutines. Dissertation, Bachelor of Science in Computer Science with Honours, The University of Bath.

Barnett, M., K. Leino, and W. Schulte (2005). The Spec# programming system: An overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean (Eds.), *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, Volume 3362 of *Lecture Notes in Computer Science*, pp. 49–69. Springer Berlin / Heidelberg.

Birkhoff, G. (1967). Lattice theory. In *Colloquium Publications* (3 ed.), Volume 25. Amer. Math. Soc.

Bloch, J. (2006, June). Extra, extra - read all about it: Nearly all binary searches and mergesorts are broken. `http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html`.

Böhm, C. and G. Jacopini (1966, May). Flow diagrams, Turing machines and languages with only two formation rules. *Commun. ACM 9*(5), 366–371.

Böhme, S., K. R. Leino, and B. Wolff (2008). HOL-Boogie – an interactive prover for the Boogie program-verifier. In *TPHOLs '08: Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, Berlin, Heidelberg, pp. 150–166. Springer-Verlag.

Boiten, E. and J. Derrick (1998). IO-refinement in Z. In *Proceedings of the 3rd BCS-FACS conference on Northern Formal Methods*, 3FACS'98, Swinton, UK, UK, pp. 3–3. British Computer Society.

Borba, P., A. Sampaio, A. Cavalcanti, and M. Cornélio (2004). Algebraic reasoning for object-oriented programming. *Science of Computer Programming 52*(1-3), 53 – 100.

Bowen, J. and V. Stavridou (1993). Safety-critical systems, formal methods and standards. *Software Engineering Journal 8*(4), 189–209.

Brucker, A. and B. Wolff (2002). A proposal for a formal OCL semantics in Isabelle/HOL. In V. Carreño, C. Muñoz, and S. Tahar (Eds.), *Theorem Proving in Higher Order Logics*, Volume 2410 of *Lecture Notes in Computer Science*, pp. 147–175. Springer Berlin / Heidelberg.

Buhr, P. and W. Mok (2000, sep). Advanced exception handling mechanisms. *Software Engineering, IEEE Transactions on 26*(9), 820 –836.

Cavalcanti, A. and J. Woodcock (1998). A weakest precondition semantics for Z. *The Computer Journal 41*(1), 1–15.

Clarke, E. M. (1980). Proving correctness of coroutines without history variables. *Acta Informatica 13*, 169–188.

Clarke, E. M. and J. M. Wing (1996). Formal methods: state of the art and future directions. *ACM Comput. Surv. 28*(4), 626–643.

Clint, M. (1973). Program proving: Coroutines. *Acta Informatica 2*, 50–63.

Cohen, E., M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies (2009). VCC: A practical system for verifying concurrent

C. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel (Eds.), *Theorem Proving in Higher Order Logics*, Volume 5674 of *Lecture Notes in Computer Science*, pp. 23–42. Springer Berlin / Heidelberg.

Collette, P. and C. B. Jones (1995). Enhancing the tractability of rely/guarantee specifications in the development of interfering operations. In *Proof, Language and Interaction, chapter 10*, pp. 275–305. MIT Press.

Conway, M. E. (1963, July). Design of a separable transition-diagram compiler. *Commun. ACM 6*, 396–408.

Cristian, F. (1982, june). Exception handling and software fault tolerance. *Computers, IEEE Transactions on C-31*(6), 531 –540.

Cristian, F. (1984). Correct and robust programs. *IEEE Trans. Software Eng. 10*(2), 163–174.

Cristian, F. (1989). Exception handling. In *Dependability of Resilient Computers*, pp. 68–97.

Dahl, O.-J. and K. Nygaard (1966, September). SIMULA: an ALGOL-based simulation language. *Commun. ACM 9*(9), 671–678.

Derrick, J. and E. Boiten (2001). *Refinement in Z and Object-Z, Foundations and Advanced Applications*. Formal Approaches to Computing and Information Technology. Springer.

Dijkstra, E. W. (1968, March). Letters to the editor: Go To statement considered harmful. *Commun. ACM 11*(3), 147–148.

Dijkstra, E. W. (1970, August). Structured programming. In *Software Engineering Techniques*, pp. 65–68. NATO Science Committee.

Dijkstra, E. W. (1975). Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM 18*(8), 453–457.

Dowson, M. (1997). The Ariane 5 software failure. *SIGSOFT Softw. Eng. Notes 22*(2), 84.

Ecma International (2006, June). *Eiffel: Analysis, Design and Programming Language* (2nd ed.). Standard ECMA-367. Ecma International.

Ewing, G. (2012, January). Syntax for delegating to a subgenerator. `http://www.python.org/dev/peps/pep-0380/`.

Flanagan, D. and Y. Matsumoto (2008). *The Ruby programming language* (First ed.). O'Reilly.

Garcia, A. F., C. M. F. Rubira, A. Romanovsky, and J. Xu (2001). A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of Systems and Software 59*(2), 197 – 222.

Gardiner, P. and C. Morgan (1991). Data refinement of predicate transformers. *Theoretical Computer Science 87*(1), 143 – 162.

Gardiner, P. and C. Morgan (1993). A single complete rule for data refinement. *Formal Aspects of Computing 5*(4), 367–382.

Google (2009). Go programming language. `http://www.golang.org`.

Gosling, J., B. Joy, G. Steele, G. Bracha, and A. Buckley (2013, February). The Java language specification, Java SE 7 edition. `http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf`.

Graham, R. L., D. E. Knuth, and O. Patashnik (1994). *Concrete Mathematics: A Foundation for Computer Science* (2nd ed.). Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Gurevich, Y. and J. Morris (1988). Algebraic operational semantics and Modula-2. In E. Börger, H. Büning, and M. Richter (Eds.), *CSL '87*, Volume 329 of *Lecture Notes in Computer Science*, pp. 81–101. Springer Berlin / Heidelberg.

Harrison, J. (1998). Formalizing Dijkstra. In J. Grundy and M. Newey (Eds.), *Theorem Proving in Higher Order Logics*, Volume 1479 of *Lecture Notes in Computer Science*, pp. 171–188. Springer Berlin / Heidelberg.

Harrison, J. (2011, January). HOL Light tutorial (for version 2.20). Technical report, Intel JF1-13.

He, J., C. Hoare, and J. Sanders (1986). Data refinement refined resume. In B. Robinet and R. Wilhelm (Eds.), *ESOP 86*, Volume 213 of *Lecture Notes in Computer Science*, pp. 187–196. Springer Berlin Heidelberg.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM 12*(10), 576–580.

Hoare, C. A. R., I. J. Hayes, H. Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin (1987, August). Laws of programming. *Commun. ACM 30*(8), 672–686.

Hoare, C. A. R., H. Jifeng, and A. Sampaio (1993). Normal form approach to compiler design. *Acta Informatica 30*, 701–739.

Horning, J., H. Lauer, P. Melliar-Smith, and B. Randell (1974). A program structure for error detection and recovery. In E. Gelenbe and C. Kaiser (Eds.), *Operating*

Systems, Volume 16 of *Lecture Notes in Computer Science*, pp. 171–187. Springer Berlin / Heidelberg.

Howie, J. M. (1995). *Fundamentals of semigroup theory*. Clarendon Press.

Ierusalimschy, R., L. H. de Figueiredo, and W. Celes (2005, July). The implementation of Lua 5.0. *Journal of Universal Computer Science 11*(7), 1159–1176.

Intel (2013, March). Intel®64 and IA-32 architectures software developer's manual, volume 2 (2a, 2b & 2c): Instruction set reference, a-z. `http://download.intel.com/products/processor/manual/325383.pdf`.

ISO (1998, September). ISO/IEC 14882:1998: Programming languages — C++.

ISO (1999). ISO/IEC 9899:1999: Programming Languages — C.

ISO (2011). ISO/IEC 9899:2011: Programming Languages — C.

ISO (2012, February). ISO/IEC 14882:2011 Information technology — Programming languages — C++.

Jeffords, R., C. Heitmeyer, M. Archer, and E. Leonard (2009). A formal method for developing provably correct fault-tolerant systems using partial refinement and composition. In A. Cavalcanti and D. Dams (Eds.), *FM 2009: Formal Methods*, Volume 5850 of *Lecture Notes in Computer Science*, pp. 173–189. Springer Berlin Heidelberg.

King, S. and C. Morgan (1995). Exits in the refinement calculus. *Formal Asp. Comput. 7*(1), 54–76.

Knuth, D. E. (1968). *The art of computer programming, volume 1: fundamental algorithms*. Redwood City, CA, USA: Addison Wesley Publishing Company, Inc.

Knuth, D. E. (1974). Structured programming with go to statements. *ACM Comput. Surv. 6*(4), 261–301.

Kozen, D. (1997, May). Kleene algebra with tests. *ACM Trans. Program. Lang. Syst. 19*(3), 427–443.

Lamport, L. (1980). The 'Hoare logic' of concurrent programs. *Acta Informatica 14*(1), 21–37.

Lamport, L. (1990, July). *win* and *sin*: predicate transformers for concurrency. *ACM Trans. Program. Lang. Syst. 12*, 396–428.

Leavens, G. T., A. L. Baker, and C. Ruby (2006). Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes 31*(3), 1–38.

Lecomte, T., T. Servat, and G. Pouzancre (2007, August). Formal methods in safety-critical railway systems. Ouro Preto, Brazil. 10th Brasilian Symposium on Formal Methods.

Leinenbach, D. and T. Santen (2009). Verifying the Microsoft Hyper-V hypervisor with VCC. In A. Cavalcanti and D. Dams (Eds.), *FM 2009: Formal Methods*, Volume 5850 of *Lecture Notes in Computer Science*, pp. 806–809. Springer Berlin / Heidelberg.

Leveson, N. and C. Turner (1993, jul). An investigation of the Therac-25 accidents. *Computer 26*(7), 18 –41.

Liskov, B. and J. Guttag (2000). *Program Development in Java: Abstraction, Specification, and Object-Oriented Design* (1st ed.). Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Luckham, D. C. and W. Polak (1980, April). Ada exception handling: an axiomatic approach. *ACM Trans. Program. Lang. Syst. 2*(2), 225–233.

Marlin, C. (1980). *Coroutines*, Volume 95 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg.

Métayer, C., J.-R. Abrial, and L. Voisin (2005). *Event-B Language, in RODIN Project Deliverable 3.2*.

Meyer, B. (1987). Eiffel: programming for reusability and extendibility. *SIGPLAN Not. 22*(2), 85–94.

Meyer, B. (1988). Eiffel*: A language and environment for software engineering. *The Journal of Systems and Software*.

Meyer, B. (1997). *Object-Oriented Software Construction* (2nd ed.). Prentice-Hall.

Microsoft (2013a). Notimplementedexception class. `http://msdn.microsoft.com/en-us/library/system.notimplementedexception.aspx`.

Microsoft (2013b). Overflowexception class. `http://msdn.microsoft.com/en-us/library/system.overflowexception.aspx`.

Mikhajlova, A. and E. Sekerinski (1997). Class refinement and interface refinement in object-oriented programs. In *FME '97: Proceedings of the 4th International Symposium of Formal Methods Europe on Industrial Applications and Strengthened Foundations of Formal Methods*, London, UK, pp. 82–101. Springer-Verlag.

Mitchell, J. G., W. Maybury, and R. Sweet (1979, Apr). Mesa language manual. Technical report, Xerox Research Center, Palo Alto, California.

Moura, A. L. D. and R. Ierusalimschy (2009, February). Revisiting coroutines. *ACM Trans. Program. Lang. Syst. 31*, 6:1–6:31.

Myers, G. J. (2004, June). *The Art of Software Testing* (2nd ed.). Wiley.

Nipkow, T. and L. Paulson (1992). Isabelle-91. In D. Kapur (Ed.), *Automated Deduction—CADE-11*, Volume 607 of *Lecture Notes in Computer Science*, pp. 673–676. Springer Berlin / Heidelberg.

Nipkow, T., M. Wenzel, and L. C. Paulson (2002). *Isabelle/HOL: a proof assistant for higher-order logic*. Berlin, Heidelberg: Springer-Verlag.

Nordio, M., C. Calcagno, P. Müller, and B. Meyer (2009). A sound and complete program logic for Eiffel. In W. Aalst, J. Mylopoulos, N. M. Sadeh, M. J. Shaw, C. Szyperski, M. Oriol, and B. Meyer (Eds.), *Objects, Components, Models and Patterns*, Volume 33 of *Lecture Notes in Business Information Processing*, pp. 195–214. Springer Berlin Heidelberg.

Parnas, D. L. (1978). Designing software for ease of extension and contraction. In *Proceedings of the 3rd international conference on Software engineering*, ICSE '78, Piscataway, NJ, USA, pp. 264–277. IEEE Press.

Plotkin, G. D. (1981). The origins of structural operational semantics. *Journal of Logic and Algebraic Programming 60*, 60–61.

Python Software Foundation (2013, May). The Python language reference. `http://docs.python.org/3/reference/`.

Randell, B. (1975). System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, New York, NY, USA, pp. 437–449. ACM.

Reed, M., C. George, and W. H. I. Wa (2007, February). UNU-IIST annual report 2006. Technical report, United Nations University International Institute for Software Technology.

Schmidt, D. A. (1986). *Denotational semantics: a methodology for language development*. Dubuque, IA, USA: William C. Brown Publishers.

Sekerinski, E. (2011). Exceptions for dependability. In L. Petre, K. Sere, and E. Troubitsyna (Eds.), *Dependability and Computer Engineering: Concepts for Software-Intensive Systems—a Handbook on Dependability Research*, pp. 11–35. IGI Global.

Sekerinski, E. and T. Zhang (2011). A new notion of partial correctness for exception handling. In B. Bonakdarpour and T. Maibaum (Eds.), *2nd International Workshop on Logical Aspects of Fault-Tolerance*, pp. 116–132.

Sekerinski, E. and T. Zhang (2012). Verification rules for exception handling in Eiffel. In R. Gheyi and D. Naumann (Eds.), *Formal Methods: Foundations and Applications*, Volume 7498 of *Lecture Notes in Computer Science*, pp. 179–193. Springer Berlin / Heidelberg.

Shankar, A. (2003, September). Implementing coroutines for .NET by wrapping the unmanaged fiber API. `http://msdn.microsoft.com/en-us/magazine/cc164086.aspx`.

Stepney, S., D. Cooper, and J. Woodcock (1998). More powerful Z data refinement: Pushing the state of the art in industrial refinement. In J. Bowen, A. Fett, and M. Hinchey (Eds.), *ZUM '98: The Z Formal Specification Notation*, Volume 1493 of *Lecture Notes in Computer Science*, pp. 284–307. Springer Berlin Heidelberg.

Tarski, A. (1955). A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics 5*, 285–309.

Tennent, R. D. (1976, August). The denotational semantics of programming languages. *Commun. ACM 19*(8), 437–453.

Tschannen, J., C. A. Furia, M. Nordio, and B. Meyer (2011). Verifying Eiffel programs with Boogie. In *BOOGIE workshop*.

van Rossum, G. and P. J. Eby (2011, June). Coroutines via enhanced generators. `http://www.python.org/dev/peps/pep-0342/`.

von Oheimb, D. (2001). Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience 13*(13), 1173–1214.

von Wright, J. (1994). The lattice of data refinement. *Acta Informatica 31*(2), 105–135.

Watson, G. (2002). Refining exceptions using King and Morgan's exit construct. In *Software Engineering Conference, 2002. Ninth Asia-Pacific*, pp. 43–51.

Wirth, N. (1971). Program development by stepwise refinement. *Commun. ACM 14*(4), 221–227.

Yemini, S. and D. M. Berry (1985, April). A modular verifiable exception handling mechanism. *ACM Trans. Program. Lang. Syst. 7*(2), 214–243.

Ying, M. (2003). Reasoning about probabilistic sequential programs in a probabilistic logic. *Acta Informatica 39*(5), 315–389.