Vrije Universiteit Amsterdam
Universiteit van Amsterdam

Master's thesis

Submitted in partial fulfillment of the requirements for the joint degree of Master of Science in Computer Science.

Formalizing the Semantics of Concurrent Revisions

Roy Overbeek
(ID: 1983768)

supervisors

Vrije Universiteit Amsterdam	ING
dr. J.C. Blanchette	ir. R.W. van Dalen
prof. dr. W.J. Fokkink	

November 28, 2018

Abstract

Concurrent revisions is a concurrency control model developed by Microsoft Research. It has many interesting properties that distinguish it from other well-known models such as transactional memory. One of these properties is *determinacy*: programs written within the model always produce the same outcome, independent of scheduling activity. The concurrent revisions model has an operational semantics, with an informal proof of determinacy. This thesis describes an Isabelle/HOL formalization of this semantics and the proof of determinacy. We identify a number of subtle ambiguities in the specification of the semantics, the resolution of which requires the semantics to be modified. We also work out many details of the proof of determinacy, and show that the proof can be simplified. While the uncovered issues do not appear to map to bugs in existing implementations, the formalization highlights some of the challenges that are involved in the general design and verification of concurrency control models.

Acknowledgements

I would like to express my gratitude to the following people. Jasmin Blanchette, Robbert van Dalen and Wan Fokkink for providing a wealth of useful feedback and for being incredibly attentive supervisors overall. Joost Bosman for hiring me as a funded research intern at ING, for his excitement about my project and for offering me virtually boundless freedom and flexibility. Jorryt Dijkstra for helping me find my way around ING, and getting me in touch with the right people. All of my X-LinQ colleagues for the enjoyable time I have spent in their group, and for being supportive of my project—special thanks go to Viet Nguyen for accepting me aboard. Johannes Hölzl for helping me get started with Isabelle, in particular by clearing some of the confusions I had due to being accustomed to Coq. Sebastian Burckhardt for taking the time to answer some of my questions about concurrent revisions, and for expressing his interest in my project. All of the wonderful people of the Theoretical Computer Science section at the Vrije Universiteit, past and present (I'd like to especially mention Jasmin Blanchette, Jörg Endrullis, Wan Fokkink, Clemens Grabmayer and Femke van Raamsdonk): I have spent quite a few years studying and working in your department, and I think the present thesis is a product not only of your expertise but also of the great care you show for your students. I am especially grateful to Jörg Endrullis, who has actively involved me in his research (on this aspect, my thanks go to Jan Willem Klop, too!) and software development projects during my studies—you have been a true mentor to me. Of my friends, I'd like to thank Gabór Kozar, Christopher Esterhuyse, Hans-Dieter Hiep and Ivar Meijs for the useful discussions we've had about writing and about formal methods generally. Finally, I feel very indebted to my family, all my friends, and of course my girlfriend, Verónica Seba, for their unending support during this past year: thank you all!

Contents

List of Figures

Chapter 1

Introduction

This thesis presents an Isabelle/HOL formalization of the semantics of concurrent revisions. The subject of the formalization, concurrent revisions, is a relatively recent concurrency control model developed by Microsoft Research. The formalization tool, Isabelle/HOL, is a higher-order logic interactive theorem prover with strong proof automation.

1.1 Motivation

Concurrency refers to the phenomenon in which the executions of multiple processes are interleaved non-deterministically. If these processes interact through shared state or through some message passing protocol, then it is important to ensure that any unwelcome interactions are ruled out. This requires one to first identify which of the exponentially many interleavings are unsafe, and to then prevent them from ever occurring. Should a programmer fail to meet either of these requirements, then it may take many executions before a dormant bug manifests itself. When it inadvertently does manifest, the consequences can be severe, and the cause exceptionally difficult to pinpoint in the post-mortem analysis.

Survey studies support the folklore claim that developing concurrent software is challenging. Godefroid and Nagappan [GN08] conducted a large internal questionnaire on the subject at Microsoft, which revealed that its engineers judge concurrency bugs to be difficult to detect, reproduce, debug and fix, and that they are usually classified as severe. Relatedly, Lu et al. [LPSZ08] surveyed the bug characteristics of 105 randomly selected real world concurrency bugs. With respect to severity, they found that 34 of these bugs caused program crashes, and 37 bugs caused programs to hang. With respect to diagnosis, they found that bug reporters complained about the inability to reproduce bugs, sometimes leading to bug reports being closed prematurely, or bugs being incorrectly fixed based on guesswork. They also found that none of the bug reports mentioned the use of automatic diagnostic tools—a situation that contrasts starkly with, e.g., memory bug reports, in which programmers frequently report the use of diagnostic tools such as Valgrind.

Case studies also vividly illustrate the catastrophic potential of concurrency bugs. A particularly egregious example is found in Facebook's initial public offering (IPO) on the NASDAQ stock exchange, which was the third largest IPO in U.S. history [KL13]. Usually, NASDAQ's IPO Cross software takes less than 40 microseconds to compute an opening price based on a stock's initial bids and offers. Due to the heavy interest in Facebook, however, this time it took 5 milliseconds. Concurrently to this calculation, investors were allowed to change their orders, which causes the software to recalculate the opening price. An endless loop ensued—fuelled by panic spreading among investors—which eventually required manual intervention: the publication of the opening bid ended up being delayed by half an hour. Some estimates claim that this glitch cost traders a total of $100 million, and it was reported that hundreds of hours of testing failed to expose the concurrency bug.

What can be done? Part of the solution is to provide programmers with concurrency control abstractions that help write safe and understandable concurrent software. In essence, such abstractions establish one or more properties that function as simplifying assumptions for writing concurrent software. Well-known abstractions include the lock and the family of transactional memory (TM) [HLR10] models, with software transactional memory (STM) [ST97] being a particularly influential variant. The concurrent revisions model—of central interest to the present thesis—is another, much less well-known concurrency control abstraction, whose interesting details we reserve for Chapter 2.

Since the programmer relies on the properties of a concurrency control abstraction, it is vital that they are well defined and well understood, and that any underlying design or implementation actually establishes them. The research related to the formal specification and verification of a variety of TM models, for instance, is abundant. Relevant studies include publications by

- Harris et al. [HMPJH05], who extended the default STM model with blocking and choice mechanics, in the form of both an implementation (STM Haskell) and a formal operational semantics;

- Manovit et al. [MHC$^+$06], who developed a formal axiomatic framework and pseudorandom testing methodology for TM systems, and used it to uncover bugs in the relatively well-known Transactional memory Coherence and Consistency (TCC) [HWC$^+$04] system;

- Abadi et al. [ABHI08], who developed a formal semantics for the Automatic Mutual Exclusion (AME) model (a transactional model related to TM), and used it to study design trade-offs and errors that occur in known STM implementations, such as 'Bartok-STM' [HPST06];

- Cohen et al. [CPZ08] and Doherty et al. [DGLM13], who both developed frameworks for the formal verification of TM implementations, using the interactive theorem prover PVS; and
- Doherty et al. [DDD$^+$17], who presented the first formal verification of a so-called 'pessimistic' (i.e., non-aborting) STM algorithm using Isabelle/HOL, extending a refinement strategy pursued in their earlier work [DGLM13].

By contrast, the formal theory related to the concurrent revisions model consists of just a single publication: a technical report describing a formal semantics [BL11] (inspired by the semantics for AME), which serves as an accompaniment to a more practically-oriented paper (describing an implementation in C#) [BBL10]. The technical report establishes a number of defining properties of the concurrent revisions model. One of these requires us to distinguish between determinism and determinacy, a practice originating from the parallel programming community [KM66]: a program is *deterministic* if it always give rise to the same execution, and it is *determinate* if it always produces the same outcome. Programs written within the concurrent revisions model are determinate, assuming the absence of external sources of indeterminacy (such as random number generation). This is quite different from the situation for locks, where the outcome of an execution may depend on which process obtains a lock first, and the situation for TM, where the outcome of an execution may depend on which process first commits its transaction.

This thesis describes a formal verification of the concurrent revisions semantics, with an emphasis on the proof of determinacy. The motivation is twofold. First, we want to strengthen the theoretical foundations of the concurrent revisions model in particular: are the formal semantics well specified, and does it indeed establish determinacy? If not, what changes to the semantics are necessary? Second, we wish to provide a general case study for the application of formal methods in the design of concurrency control abstractions—one that is intended to be more accessible to the non-specialist than most published literature on formal verification.

1.2 Contributions

This main contributions of this thesis are the following:

- We raise interpretation questions for three details of the operational semantics (namely, the definition of a program expression, and the side conditions on the operational rules (*fork*) and (*new*)). We show that the straightforward interpretations of these definitions lead to an indeterminate concurrency model. We suggest how this situation should be remedied. We also show that two other side conditions (namely, those on rules (*get*) and (*set*)) are redundant.

- We fill out many of the details omitted in the original proof of determinacy, and provide a proof simplification.
- We formalized all the proofs using Isabelle/HOL. The resulting artifact is a little over 3000 lines, and will be published to The Archive of Formal Proofs, the official archive for Isabelle verifications.[1]

1.3 Thesis outline

The thesis is structured as follows. In Chapter 2, we provide the required high-level background on the concurrent revisions model (Section 2.1), and describe its formal semantics (Section 2.2). In describing the formal semantics, we highlight three perceived ambiguities, which we address and resolve in Chapter 3, leading to a modified version of the formal semantics. We prove determinacy for this version of the semantics in Chapter 4, and include an explicit comparison with the original proof (Section 4.3.2). In Chapter 5, we provide extensive background on Isabelle/HOL, tailored to readers unacquainted with formal methods. This background chapter contains most of the information needed to acquire a global understanding of the Isabelle/HOL formalization, described in Chapter 6. Finally, we discuss the significance of our findings and comment on the formalization process in Chapter 7.

[1]The Archive of Formal Proofs is available at `https://www.isa-afp.org`.

Chapter 2

Concurrent revisions

The concurrent revisions model was originally formulated by Burckhardt, Baldassin and Leijen in 2010 [BBL10]. Their aim was to formulate a mechanism for managing shared data between asynchronous tasks, that would moreover be relatively easy to write and reason about (relative to, arguably, locks). The original paper included an implementation in C#, and was accompanied by a technical report describing a formal semantics [BL11], for which certain desired properties (such as determinacy of executions) were proven to hold. Since then, multiple papers have been published on concurrent revisions as part of an encompassing Microsoft research project.[1] Among the contributions of these publications are model extensions, such as support for incremental computation [BLS$^+$11], as well as an implementation in Haskell [LFB11].

In this chapter we first provide a high-level description of the original concurrent revisions model (Section 2.1), highlighting its virtues. We then provide a detailed description of the formal semantics (Section 2.2), which is the subject of the formalization described by this thesis.

2.1 High-level description

The central unit of concurrency in the concurrent revisions model is the *revision*. A revision is a task that operates on a (conceptually) isolated, local copy of shared data, and is uniquely identified by an *identifier* (which is part of the program logic). All computation takes place within some revision. Initially, there is only one revision: the *main revision*.

Revisions do not interact, unless an explicit synchronization operation is performed, of which only two exist. The first synchronization operation is the *fork*. When a revision r forks some task, a new revision r' is created that executes the task. At creation, the

[1] An overview of the project is available at https://www.microsoft.com/en-us/research/project/concurrent-revisions/.

$$\begin{tikzcd}
r_3 & & & & & \cdot \arrow[r] & \cdot \arrow[dr, dotted] & & \\
r_2 & & & \cdot \arrow[r] & \cdot \arrow[r] \arrow[ur, dotted] & \cdot \arrow[r] & \cdot \arrow[r] & c \arrow[dr, dotted] & \\
r_1 & \cdot \arrow[r] & a \arrow[r] \arrow[ur, dotted] & \cdot \arrow[r] & b \arrow[rrrr] & & & & d
\end{tikzcd}$$

Figure 2.1: A simple revision diagram. Dotted arrows denote fork and join relations.

store of revision r' is initialized with a copy of the data of revision r, and the identifier of revision r' is exposed to revision r. The second synchronization operation is the *join*. When revision r knows the identifier of revision r', then r can join r'. This causes r to block until r' has finished its computation. When r' is finished, the stores of r and r' are *merged* at r to form a new store, and r' ceases to exist. Joining a non-existent revision is considered an error.

The merge of two stores can be best explained in relation to a *revision diagram*, which is a diagram that depicts the interactions between revisions. In these diagrams, we use solid arrows to depict computational steps *within* revisions, and dotted arrows to depict fork and join relations *between* revisions. Figure 2.1 shows a simple example, in which four states are marked (a, b, c and d). In state a, r_1 forks r_2. In state b, r_1 initiates a join on r_2, which blocks until r_2 reaches its terminal state c. The stores of b and c contain modifications relative to the *closest common ancestor* a: if none of these modifications conflict (i.e., no data object in the original store was updated to two distinct values), then the merged store at d is effectively the store that is obtained by applying all of b and c's modifications to store a.

Stores that need to be merged may have performed writes to the same values (a *write-write* conflict). Whereas lock-based programs are designed to avoid conflicts, and STM discards tentative transactions that turn out to conflict with the committed memory, concurrent revisions asks one to resolve all conflicts. This happens through the following *data-centric* approach. Every declaration of a shared data object is annotated with an *isolation type*. The isolation type determines which *merge function* is used to resolve a conflict on that object. The merge function computes a value based on (1) the value at the closest common ancestor in the revision diagram, (2) the value at the joiner and (3) the value at the joinee. The merge function should be deterministic, as to not threaten the important determinacy property of concurrent revisions (discussed in Section 1.1). Other than that, which merge function to use is up to the programmer, as it is application-dependent.

We illustrate the concept of an isolation type with two standard examples: *Versioned* and *Cumulative* isolation types. If an object has the *Versioned* isolation type, then any conflict is resolved by choosing the joinee's value. This isolation type is particularly useful if there exists a clear priority order between tasks. The *Cumulative* isolation type, by contrast, is more sophisticated: it effectively applies the changes by both revisions to

Versioned <Int>

$$\begin{tikzcd}
r_2 & & & \cdot \arrow[r, "x := 2"] & \cdot \arrow[rd, dotted, bend left] & \\
r_1 & \cdot \arrow[r, "x := 3"] & \cdot \arrow[r] \arrow[ru, dotted, bend left] & \cdot \arrow[r, "x := 7"] & \cdot \arrow[r] & {[x \mapsto 2]}
\end{tikzcd}$$

Cumulative <Int>

$$\begin{tikzcd}
r_2 & & & \cdot \arrow[r, "x := 5"] & \cdot \arrow[rd, dotted, bend left] & \\
r_1 & \cdot \arrow[r, "x := 3"] & \cdot \arrow[r] \arrow[ru, dotted, bend left] & \cdot \arrow[r, "x := 5"] & \cdot \arrow[r] & {[x \mapsto 7]}
\end{tikzcd}$$

Figure 2.2: Merging: *Versioned* <*Int*> versus *Cumulative* <*Int*>.

$$\begin{tikzcd}
r_3 & & & \cdot \arrow[r] & \cdot \arrow[r] & \cdot \arrow[rdd, dotted, bend left] & & \\
r_4 & & & & & \cdot \arrow[r] & \cdot \arrow[rdd, dotted, bend left] & \\
r_2 & & \cdot \arrow[r] \arrow[ruu, dotted, bend left] & \cdot \arrow[rd, dotted, bend left] & & & & \\
r_1 & \cdot \arrow[r] \arrow[ru, dotted, bend left] & \cdot \arrow[r] & \cdot \arrow[r] & \cdot \arrow[r] \arrow[ruu, dotted, bend left] & \cdot \arrow[r] & \cdot \arrow[r] & \cdot
\end{tikzcd}$$

Figure 2.3: A valid 'bridge-nested' revision diagram.

the original object. Thus, in the case of a *Cumulative* <*Int*> merge, where x is the original value, y is the value at the joiner, and z is the value at the joinee, the result of the merge is $y + z - x$ (Figure 2.2).

Just like any other data, revision identifiers can be stored. Thus, it is possible for a revision to gain access to a revision identifier r by joining a revision that has a reference to r in its store. This fact explains complex revision diagrams, such as the so-called 'bridge-nested' diagram (Figure 2.3). By contrast, a 'cross-over' diagram (Figure 2.4) is impossible, since there is no way in which revision r_1 could have seen identifier r_3. Burckhardt and Leijen [BL11] show that each pair of states has a unique closest common ancestor, which implies that the merge operation is well defined.

Apart from determinacy, the concurrent revisions approach to concurrency differs

$$\begin{tikzcd}
r_3 & & & \cdot \arrow[r] & \cdot \arrow[rdd, dotted, bend left] & \\
r_2 & & \cdot \arrow[r] \arrow[ru, dotted, bend left] & \cdot & & \\
r_1 & \cdot \arrow[r] \arrow[ru, dotted, bend left] & \cdot \arrow[r] & \cdot \arrow[r] & \cdot \arrow[r] & \cdot
\end{tikzcd}$$

Figure 2.4: An invalid 'cross-over' diagram.

from other paradigms in another important way. Unlike most building blocks of transactional paradigms, revisions are not *serializable*, meaning that they cannot always be arranged in some serial (i.e. non-overlapping) order. For instance, consider the following pseudocode program, in which the main revision forks two revisions r_1 and r_2, and then joins them:

```
Versioned <Int> $x := 0$
Versioned <Int> $y := 0$
$r_1$ := fork { if $x = 0$ then $y := 1$ }
$r_2$ := fork { if $y = 0$ then $x := 1$ }
join $r_1$
join $r_2$
print $(x, y)$
```

It is easy to check that the main revision will print (1,1): neither r_1 nor r_2 can therefore in any way be said to have occurred before the other.

Serializability is often desired because the behaviour of serial executions is well understood. Burckhardt and Leijen argue that this reduction to the sequential world is not strictly required: programmers can reason about non-linear histories of state using revision diagrams, while relying on the fact that executions are determinate. This, combined with the declarative, data-centric approach to conflict resolution, make concurrent revisions an elegant model for concurrent applications in which conflicts are resolvable.

2.2 Formal semantics

Burckhardt and Leijen present a semantics of concurrent revisions [BL11]. The semantics takes the form of an operational semantics, which they call the *revision calculus*. The calculus is very expressive: it defines the types, syntax and semantics of a rudimentary programming language; and it captures the concepts of a simple memory model, a very precise (but implicit) evaluation order, and asynchronous computation. The calculus is also very concise, as it fits on a single page. These properties make it highly suitable as a reference for implementations and as a tool for studying extensions of the model.

This section describes the revision calculus as it was presented by Burckhardt and Leijen. On a small number of aspects, more than one interpretation seemed reasonable to us: we highlight these perceived ambiguities here, and fully explore them in Chapter 3. We also introduce some additional notation that is needed for our own purposes.

The function notations (which are largely adopted from the original paper) have the following meanings:

- $A \rightharpoonup B$ denotes the set of partial functions from A to B;
- $f[\![x \mapsto y]\!]$ denotes a function f for which $f\ x = y$ (we write $f[\![x_1 \mapsto y_1, x_2 \mapsto y_2]\!]$ for $f[\![x_1 \mapsto y_1]\!][\![x_2 \mapsto y_2]\!]$);

Syntactic symbols

$$\begin{array}{llll}
x & \in & \mathit{Var} & \\
l & \in & \mathit{Loc} & \\
r & \in & \mathit{Rid} & \\
c & \in & \mathit{Const} & ::= \mathsf{unit} \mid \mathsf{false} \mid \mathsf{true} \\
v & \in & \mathit{Val} & ::= c \mid x \mid l \mid r \mid \lambda x.e \\
e & \in & \mathit{Expr} & ::= v \mid e\ e \mid e\ ?\ e : e \\
 & & & \quad\mid \mathsf{ref}\ e \mid !e \mid e := e \\
 & & & \quad\mid \mathsf{rfork}\ e \mid \mathsf{rjoin}\ e
\end{array}$$

State

$$\begin{array}{lllll}
\sigma & \in & \mathit{Snapshot} & = & \mathit{Loc} \rightharpoonup \mathit{Val} \\
\tau & \in & \mathit{LocalStore} & = & \mathit{Loc} \rightharpoonup \mathit{Val} \\
 & & \mathit{LocalState} & = & \mathit{Snapshot} \times \mathit{LocalStore} \times \mathit{Expr} \\
s & \in & \mathit{GlobalState} & = & \mathit{Rid} \rightharpoonup \mathit{LocalState}
\end{array}$$

Execution contexts

$$\begin{array}{llll}
\mathcal{E} & \in & \mathit{Cntxt} & ::= \square \\
 & & & \quad\mid \mathcal{E}\ e \mid v\ \mathcal{E} \mid \mathcal{E}\ ?\ e : e \\
 & & & \quad\mid \mathsf{ref}\ \mathcal{E} \mid !\mathcal{E} \mid \mathcal{E} := e \mid l := \mathcal{E} \\
 & & & \quad\mid \mathsf{rjoin}\ \mathcal{E}
\end{array}$$

Operational semantics

$$\begin{array}{llll}
(\mathit{apply}) & s[\![r \mapsto \langle\sigma,\tau,\mathcal{E}[(\lambda x.e)\ v]\rangle]\!] & \to_r & s(r \mapsto \langle\sigma,\tau,\mathcal{E}[[v/x]e]\rangle) & \\
(\mathit{if\text{-}true}) & s[\![r \mapsto \langle\sigma,\tau,\mathcal{E}[\mathsf{true}\ ?\ e_1 : e_2]\rangle]\!] & \to_r & s(r \mapsto \langle\sigma,\tau,\mathcal{E}[e_1]\rangle) & \\
(\mathit{if\text{-}false}) & s[\![r \mapsto \langle\sigma,\tau,\mathcal{E}[\mathsf{false}\ ?\ e_1 : e_2]\rangle]\!] & \to_r & s(r \mapsto \langle\sigma,\tau,\mathcal{E}[e_2]\rangle) & \\
 & & & & \\
(\mathit{new}) & s[\![r \mapsto \langle\sigma,\tau,\mathcal{E}[\mathsf{ref}\ v]\rangle]\!] & \to_r & s(r \mapsto \langle\sigma,\tau(l \mapsto v),\mathcal{E}[l]\rangle) & \text{if } l \notin s \\
(\mathit{get}) & s[\![r \mapsto \langle\sigma,\tau,\mathcal{E}[!l]\rangle]\!] & \to_r & s(r \mapsto \langle\sigma,\tau,\mathcal{E}[(\sigma::\tau)\ l]\rangle) & \text{if } l \in \mathsf{dom}\ (\sigma::\tau) \\
(\mathit{set}) & s[\![r \mapsto \langle\sigma,\tau,\mathcal{E}[l := v]\rangle]\!] & \to_r & s(r \mapsto \langle\sigma,\tau(l \mapsto v),\mathcal{E}[\mathsf{unit}]\rangle) & \text{if } l \in \mathsf{dom}\ (\sigma::\tau) \\
 & & & & \\
(\mathit{fork}) & s[\![r \mapsto \langle\sigma,\tau,\mathcal{E}[\mathsf{rfork}\ e]\rangle]\!] & \to_r & s(r \mapsto \langle\sigma,\tau,\mathcal{E}[r']\rangle, r' \mapsto \langle\sigma::\tau,\epsilon,e\rangle) & \text{if } r' \notin s \\
(\mathit{join}) & s[\![r \mapsto \langle\sigma,\tau,\mathcal{E}[\mathsf{rjoin}\ r']\rangle, r' \mapsto \langle\sigma',\tau',v\rangle]\!] & \to_r & s(r \mapsto \langle\sigma,\tau::\tau',\mathcal{E}[\mathsf{unit}]\rangle, r' \mapsto \bot) & \\
(\mathit{join}_\epsilon) & s[\![r \mapsto \langle\sigma,\tau,\mathcal{E}[\mathsf{rjoin}\ r']\rangle, r' \mapsto \bot]\!] & \to_r & \epsilon &
\end{array}$$

Figure 2.5: The syntax and operational semantics of the revision calculus.

- $f(x \mapsto y)$ denotes an updated function, i.e. it denotes the function that maps x to y and $z \neq x$ to $f\ z$ (we write $f(x_1 \mapsto y_1, x_2 \mapsto y_2)$ for $f(x_1 \mapsto y_1)(x_2 \mapsto y_2)$);
- $\mathsf{dom}\ f$ and $\mathsf{ran}\ f$ respectively denote the domain and range of f;
- $f\ x = \bot$ means $x \notin \mathsf{dom}\ f$;
- f^{-1} denotes the inverse of an injective function f;
- $f :: g$ is a function that maps all $x \in \mathsf{dom}\ g$ to $g\ x$ and all $x \notin \mathsf{dom}\ g$ to $f\ x$;
- ϵ denotes the empty function.

2.2.1 Definition

Figure 2.5 defines the syntax and semantics of the revision calculus.[2] We discuss each of the four sections in turn.

[2]Ignoring some presentational modifications, the figure is an exact reproduction of Figure 5 of the original article.

Syntactic symbols The basic syntactic units of the calculus are variables (*Var*), location identifiers (*Loc*), revision identifiers (*Rid*) and constants (*Const*). The set of values (*Val*) and the set of expressions (*Expr*) are defined through mutual induction: any value v is an expression, and if x is a variable and e is an expression, then $\lambda x.\, e$ is a value.

For readability, we will sometimes write $x \bullet y$ to denote the application $x\ y$.

For values and expressions x, we write *LID* x to denote the set of all location identifiers occurring in x. Similarly, we write *RID* x to denote the set of all revision identifiers occurring in x.

State Snapshots (stores inherited from the forker) and local stores (stores tracking local changes) are partial functions from location identifiers to values; local states are triples consisting of a snapshot, a local store and an expression; and global states are partial functions from revision identifiers to local states.

For a local state L, we will write L_σ, L_τ and L_e to denote respectively the first, second and third projection of L. Furthermore, we will write $\mathsf{doms}\ L$ to denote $\mathsf{dom}\ L_\sigma \cup \mathsf{dom}\ L_\tau$ ('the domains of L').

Let $f\ `\ S$ denote the set $\{f\ x \mid x \in S\}$. For local stores and snapshots $\sigma \in \mathit{Loc} \rightharpoonup \mathit{Val}$, we define

$$\mathit{LID}\ \sigma = \mathsf{dom}\ \sigma \cup \bigcup (\mathit{LID}\ `\ \mathsf{ran}\ \sigma)$$

and

$$\mathit{RID}\ \sigma = \bigcup (\mathit{RID}\ `\ \mathsf{ran}\ \sigma).$$

For local states $\langle \sigma, \tau, e \rangle$, we define

$$\mathit{LID}\ \langle \sigma, \tau, e \rangle = \mathit{LID}\ \sigma \cup \mathit{LID}\ \tau \cup \mathit{LID}\ e$$

and

$$\mathit{RID}\ \langle \sigma, \tau, e \rangle = \mathit{RID}\ \sigma \cup \mathit{RID}\ \tau \cup \mathit{RID}\ e.$$

For global states s, finally, we define

$$\mathit{LID}\ s = \bigcup (\mathit{LID}\ `\ \mathsf{ran}\ s)$$

and

$$\mathit{RID}\ s = \mathsf{dom}\ s \cup \bigcup (\mathit{RID}\ `\ \mathsf{ran}\ s).$$

Execution contexts An execution context is an expression with exactly one hole ($\square$) in it. The expression obtained by 'plugging' an expression e into the hole of some context $\mathcal{E}$ is denoted by $\mathcal{E}[e]$. Given an expression $\mathcal{E}[e]$, we will say that e *completes* $\mathcal{E}$, and that $\mathcal{E}[e]$ is a *completion*. If r is a reducible expression (*redex*), then we call $\mathcal{E}[r]$ a *redex-completion*.

Execution contexts provide a concise syntactic method for defining an evaluation order. This follows from two facts: (1) the rules of the operational semantics exclusively

$$\begin{tikzcd}
r_2 & & \cdot & & \\
r_3 & & & \cdot \arrow[r, "\mathsf{rjoin}\ r_2"] & \cdots \\
r_1 & \cdot \arrow[r, "\mathsf{rfork\ unit}"] \arrow[uur, dotted, bend left] & \cdot \arrow[r] \arrow[ur, dotted, bend left] & \cdot \arrow[r, "\mathsf{rjoin}\ r_2"] & \cdots
\end{tikzcd}$$

Figure 2.6: Without rule (*join*$_\epsilon$), either r_1 or r_3 would first join r_2, and then the other revision would be blocked.

match against redex-completions, and (2) for any expression e containing redexes, there exists a *unique* redex-completion $\mathcal{E}[r] = e$. From (1) and (2) it follows that the next redex to contract in some given expression is always uniquely determined.

Example 1. The expression $((\lambda x.\, x)\ x)\ ((\lambda y.\, y)\ y)$ can be decomposed into the context $\square\ ((\lambda y.\, y)\ y)$ and redex $(\lambda x.\, x)\ x$, since $\square\ ((\lambda y.\, y)\ y)$ is a context by constructor $\mathcal{E}\ e$. The decomposition into the context $((\lambda x.\, x)\ x)\ \square$ and redex $(\lambda y.\, y)\ y$, by contrast, is invalid: the application $(\lambda x.\, x)\ x$ is not a value, so the application constructor $v\ \mathcal{E}$ cannot be used.

A more detailed explanation of execution contexts is given by Harper [Har16].

We sometimes underline a redex r in an expression e to signify that e would decompose uniquely into $\mathcal{E}[r]$, with $\mathcal{E}$ the context surrounding r in e. In the context of execution traces, we will refer to r as the *active site* (of a computation).

Operational semantics The operational semantics define an indexed family of relations $\rightarrow_r \subseteq \mathit{GlobalState} \times \mathit{GlobalState}$, where the r intuitively denotes the revision that performs the transition.

The rules of the operational semantics are divided into groups of three. The first group contains rules that only affect the local expression, the second group contains local rules that interact with the stores, and the third group contains the rules in which revisions interact.

In rule (*apply*), the notation $[v/x]e$ denotes the operation 'v substituted for x in e'. Hence rule (*apply*) is β-contraction. From Burckhardt and Leijen's proofs it is clear that they consider rule (*apply*) to be deterministic. The typical definition of β-contraction is non-deterministic because the variable renaming in capture-avoiding substitution is not defined deterministically. Such renamings can be made deterministic by linearizing the set of variables and choosing, e.g., the smallest variable that correctly implements capture avoidance.

The rules (*new*) and (*fork*) are the only non-deterministic rules. The side condition on rule (*new*) is $l \notin s$, which Burckhardt and Leijen write as a shorthand for 'l does not appear in any snapshot or local store of s'. The side condition on rule (*fork*) is $r \notin s$, and is a shorthand for 'r does not appear in any snapshot or local store of s, and is not mapped by s'. In terms of the notations we have introduced, we believe the most faithful formalizations of these phrasings are respectively

$$l \notin s \iff l \notin \bigcup \{\mathit{LID}\ L_\sigma \cup \mathit{LID}\ L_\tau \mid L \in \mathsf{ran}\ s\}$$

and

$$r \notin s \iff r \notin \mathsf{dom}\ s \cup \bigcup \{\mathit{RID}\ L_\sigma \cup \mathit{RID}\ L_\tau \mid L \in \mathsf{ran}\ s\},$$

rather than the more strict (and more straightforward) candidates

$$l \notin s \iff l \notin \mathit{LID}\ s$$

and

$$r \notin s \iff r \notin \mathit{RID}\ s.$$

We explore the consequences of these interpretation choices in Chapter 3.

Rule (*join*) resolves all conflicts according to the *Versioned* isolation type: custom merge functions are not yet considered. We briefly discuss how this rule could be generalized in Section 2.2.4.

Rule (join_ϵ) is included so that when two revisions race to join a third revision r, the global state collapses to the error state ϵ as soon as the second join is performed. Without this rule, the calculus would be indeterminate, since only the second joiner would be forced to block on the join of r (Figure 2.6).

2.2.2 Executions

Since determinacy is a property of execution traces, we need a small vocabulary for reasoning about executions.

An *initial state* is of the form $\epsilon(r \mapsto \langle \epsilon, \epsilon, e\rangle)$, with $r \in \mathit{Rid}$ and e an arbitrary program expression. A *program expression* is an expression that 'does not contain any revision identifiers'. A question is whether this means

$$\mathit{RID}\ e = \varnothing$$

or whether the phrase should be interpreted as 'does not contain any identifiers of the revision calculus':

$$\mathit{RID}\ e = \mathit{LID}\ e = \varnothing.$$

This question is explored in Chapter 3.

Define $\rightarrow\ = \bigcup \{\rightarrow_r \mid r \in \mathit{Rid}\}$. We write $\rightarrow^=$ for the reflexive closure (zero or one step), $\rightarrow^+$ for the transitive closure (one or more steps), $\rightarrow^*$ for the reflexive transitive closure (zero or more steps), and $\rightarrow^n$ for the n-fold composition (n steps) of $\rightarrow$. As is customary, mirrored versions of these symbols denote their inverses. An *execution* is a sequence $s \rightarrow^* s'$, with s an initial state. Such an execution is *maximal* if there exists no state s'' such that $s' \rightarrow s''$. We say that a state s is *reachable* if there exists an execution towards s. Finally, we write $e \downarrow s$ if there exists a maximal execution for a program expression e that ends in s.

2.2.3 Renaming-equivalence

The side conditions for the rules (*new*) and (*fork*) make the revision calculus indeterminate, even though the non-deterministically chosen identifier names have no real significance (similar to the names of bound variables in the lambda calculus). Hence, determinacy of the calculus should be proven modulo renaming of location and revision identifiers.

More precisely, let α be a permutation of revision identifiers (i.e. a bijective function from *Rid* to itself), and let $\alpha\ s$ denote the global state obtained by renaming every revision identifier r in s to $\alpha\ r$. Let β and $\beta\ s$ be defined analogously for location identifiers. Two states s and s' are said to be *renaming-equivalent under* α *and* β, denoted $s \approx_{\alpha\beta} s'$, if $\alpha\ (\beta\ s) = s'$. Two states s and s' are said to be *renaming-equivalent*, denoted $s \approx s'$, if there exist some α and β such that $s \approx_{\alpha\beta} s'$.

2.2.4 Generalizing rule (*join*)

Rule (*join*) is quite restrictive, in the sense that it resolves all conflicts according to the *Versioned* isolation type. To generalize the calculus, Burckhardt and Leijen suggest that this rule can be replaced by

$$\begin{array}{ll} (\textit{join-merge}) & s[\![r \mapsto \langle \sigma, \tau, \mathcal{E}[\mathsf{rjoin}\ r'] \rangle, r' \mapsto \langle \sigma', \tau', v \rangle]\!] \to_r \\ & s(r \mapsto \langle \sigma, \mathsf{merge}(\tau, \tau', \sigma'), \mathcal{E}[\mathsf{unit}] \rangle, r' \mapsto \bot) \end{array}$$

with

$$\mathsf{merge}(\tau, \tau', \sigma')\ l = \begin{cases} \tau\ l, & \textit{if } \tau'\ l = \bot \\ \tau'\ l, & \textit{if } \sigma'\ l = \tau\ l \\ \mathsf{merge}_l(\tau\ l, \tau'\ l, \sigma'\ l) & \textit{otherwise} \end{cases}$$

and merge_l a merge function specifically defined for the values stored at location l. Burckhardt and Leijen claim that the choice of these individual merge functions does not influence determinacy, as long as they are a function of their three inputs.

We see two issues with this approach. First, because of how rule (*new*) is defined, a value v that is stored can end up at any fresh location l. Consequently, the merge function that will be invoked on v is arbitrary, which seems to violate the principle that merge functions are specific to particular data types.

Second, even if a location-specific merge function is a function of its three inputs, it can still violate determinacy. As an example, consider

$$\mathsf{merge}_l(v, v', v'') = \begin{cases} \mathsf{true} & \textit{if } l' \in \textit{LID}\ v \\ \mathsf{false} & \textit{otherwise} \end{cases}$$

where l and l' are fixed location identifiers. While merge_l is a function of its three inputs, whether $l' \in \textit{LID}\ v$ could depend on previous applications of rule (*new*), which allocates identifiers non-deterministically. Thus in some executions the merge operation may resolve to true, while in others it may resolve to false.

We believe these issues could be remedied as follows. First, which individual merge function is invoked should depend on the types of its values, rather than the location at which these values are stored. This might require further subtyping in the calculus. Second, the behaviour of individual merge functions should never be allowed to be contingent on non-deterministic aspects of its arguments. For the present formulation of the calculus, we believe it is sufficient to forbid references to location and revision identifiers in these function definitions. However, we do not explore the topic further, and restrict ourselves to the calculus that uses rule (*join*).

Chapter 3

Formal semantics: investigating ambiguities

In the previous chapter, we identified three potential ambiguities in the formal definition of the revision calculus:

1. A program expression e is an expression that 'does not contain any revision identifiers'. Does this imply that a program expression is allowed to contain location identifiers? Or should we read 'revision identifiers' as 'identifiers of the revision calculus', resulting in a more restrictive definition? In other words, we must choose between the definitions

$$e \text{ is a program expression } \iff \mathit{RID}\ e = \varnothing$$

and

$$e \text{ is a program expression } \iff \mathit{RID}\ e = \mathit{LID}\ e = \varnothing.$$

2. The side condition on rule (*fork*) states that the freshly allocated revision identifier r 'does not appear in any snapshot or local store of [the source state] s, and is not mapped by s'. The third projections or *expressions* of local states seem explicitly excluded from this definition. Is this because it is not necessary to require that r is fresh relative to expressions? In other words, we must choose between the side conditions

$$r \notin \mathsf{dom}\ s \cup \bigcup \{\mathit{RID}\ \mathsf{L}_\sigma \cup \mathit{RID}\ \mathsf{L}_\tau \mid \mathsf{L} \in \mathsf{ran}\ s\}$$

and

$$r \notin \mathit{RID}\ s.$$

3. Similarly, for the side condition on rule (*new*), we must choose between

$$l \notin \bigcup \{\mathit{LID}\ L_\sigma \cup \mathit{LID}\ L_\tau \mid L \in \mathsf{ran}\ s\}$$

and

$$l \notin \mathit{LID}\ s.$$

In this chapter, we first argue that the definition

$$e \text{ is a program expression } \iff \mathit{RID}\ e = \mathit{LID}\ e = \varnothing.$$

is preferable (Section 3.1). We then address item (2), and show that the stronger side condition on fork (i.e., $r \notin \mathit{RID}\ s$) is required for determinacy (Section 3.2), regardless of the interpretation that is fixed for item (1). Finally, we show that assuming the interpretation

$$e \text{ is a program expression } \iff \mathit{RID}\ e = \mathit{LID}\ e = \varnothing,$$

the weaker formulation of the side condition on (*new*) actually suffices (Section 3.3). The reason is that for any local state $\langle\sigma,\tau,e\rangle$, the property $\mathit{LID}\ \langle\sigma,\tau,e\rangle = \mathsf{dom}\ \sigma \cup \mathsf{dom}\ \tau$ is invariant under execution. As a corollary of this result, both side conditions for (*set*) and (*get*) can be omitted as well, as they are always guaranteed to be satisfied.

3.1 Definition of program expressions

Can location identifiers occur in program expressions? To answer this question, we start with the observation that *if* location identifiers are allowed to occur in program expressions *and* expressions are not checked for location identifiers upon allocation, indeterminacy results. A counter-example to determinacy under these assumptions is given by the program expression

$$\mathsf{ref\ unit} \bullet !l$$

which can be checked to admit two distinct maximal executions when initialized on some arbitrary $r \in \mathit{Rid}$, namely

$$\begin{aligned}\epsilon(r \mapsto \langle\epsilon,\epsilon,\underline{\mathsf{ref\ unit}} \bullet !l\rangle) &\rightarrow_r \epsilon(r \mapsto \langle\epsilon,\epsilon(l \mapsto \mathsf{unit}), l \bullet \underline{!l}\rangle)\\ &\rightarrow_r \epsilon(r \mapsto \langle\epsilon,\epsilon(l \mapsto \mathsf{unit}), l \bullet \mathsf{unit}\rangle)\end{aligned}$$

and

$$\epsilon(r \mapsto \langle\epsilon,\epsilon,\underline{\mathsf{ref\ unit}} \bullet !l\rangle) \rightarrow_r \epsilon(r \mapsto \langle\epsilon,\epsilon(l' \mapsto \mathsf{unit}), l' \bullet !l\rangle)$$

for $l' \neq l$. The question, then, is *which of the two assumptions should be modified*.

In our opinion, there seems to be little reason for allowing location identifiers to occur in program expressions. This is mainly because there appears to be no clear use case for user-defined location identifiers: the side condition on rule (*set*) requires an assigned location to be in the domain of one of the stores, while location identifiers can *only* be introduced into the domains of stores through rule (*new*). Thus, assuming that the counter-example above is avoided, user-defined location identifiers would never become readable or assignable: they would remain completely inert throughout the execution.

Our decision, therefore, is to allow neither revision nor location identifiers in program expressions. Burckhardt agrees that this is a better definition.[1]

3.2 (*fork*) side condition

We ask whether a revision identifier r can be safely allocated in a global state s if one *only* ensures that r does not occur in any local store or snapshot, and is not mapped by s. The answer is no: this would result in indeterminacy, even if the initial program expression does not contain any location and revision identifiers.

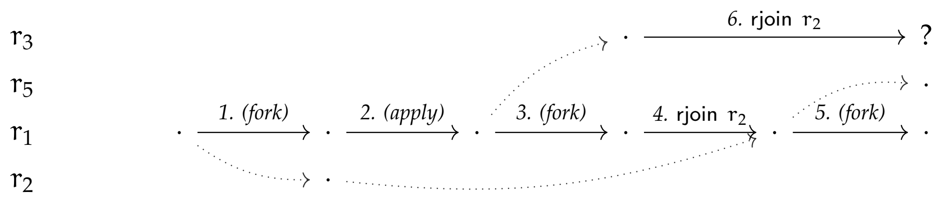

Figure 3.1: Revision diagram for the counter-example described in Section 3.2. The numbers denote the order in which the steps are performed.

What follows is a counter-example to determinacy. Define P to be the following program expression:[2]

$$P = \big(\lambda x.\, \mathsf{rfork}\ (\mathsf{rjoin}\ x) \bullet (\mathsf{rjoin}\ x \bullet \mathsf{rfork}\ \mathsf{unit})\big) \bullet \underline{\mathsf{rfork}\ \mathsf{unit}}.$$

Now consider some initial state that initializes P on some $r_1 \in \mathit{Rid}$:

$$r_1 \quad \mapsto \quad \langle \epsilon, \epsilon, P \rangle$$

An execution trace demonstrating indeterminacy is the following, where the numbers of the enumeration correspond to the numbered transitions in Figure 3.1.

1. The main revision r_1 performs the (*fork*)-step. The global state becomes

$$\begin{array}{lll} r_1 & \mapsto & \langle \epsilon, \epsilon, \underline{\big(\lambda x.\, \mathsf{rfork}\ (\mathsf{rjoin}\ x) \bullet (\mathsf{rjoin}\ x \bullet \mathsf{rfork}\ \mathsf{unit})\big) \bullet r_2} \rangle \\ r_2 & \mapsto & \langle \epsilon, \epsilon, \mathsf{unit} \rangle \end{array}$$

   for some $r_2 \neq r_1$.

[1] Personal communication through email (August 13, 2018).
[2] In general, we fork unit whenever we just want create some arbitrary revision that can be joined.

2. Revision r_1 performs the (*apply*)-step, giving

$$\begin{array}{lcl} r_1 & \mapsto & \langle \epsilon, \epsilon, \underline{\mathsf{rfork}\ (\mathsf{rjoin}\ r_2)} \bullet (\mathsf{rjoin}\ r_2 \bullet \mathsf{rfork}\ \mathsf{unit}) \rangle \\ r_2 & \mapsto & \langle \epsilon, \epsilon, \mathsf{unit} \rangle \end{array}$$

3. Revision r_1 performs the (*fork*)-step, giving

$$\begin{array}{lcl} r_1 & \mapsto & \langle \epsilon, \epsilon, r_3 \bullet (\underline{\mathsf{rjoin}\ r_2} \bullet \mathsf{rfork}\ \mathsf{unit}) \rangle \\ r_2 & \mapsto & \langle \epsilon, \epsilon, \mathsf{unit} \rangle \\ r_3 & \mapsto & \langle \epsilon, \epsilon, \underline{\mathsf{rjoin}\ r_2} \rangle \end{array}$$

   for some r_3 with $r_3 \neq r_1$ and $r_3 \neq r_2$.

4. Now r_3 and r_1 are concurrent. Let revision r_1 again perform the next step, giving

$$\begin{array}{lcl} r_1 & \mapsto & \langle \epsilon, \epsilon, r_3 \bullet (\mathsf{unit} \bullet \underline{\mathsf{rfork}\ \mathsf{unit}}) \rangle \\ r_3 & \mapsto & \langle \epsilon, \epsilon, \underline{\mathsf{rjoin}\ r_2} \rangle \end{array}$$

   At this point, the revision identifier r_2 exists in an expression only. Hence, it may be forked.

5. Let r_1 perform its final step, forking some r_4 with $r_4 \neq r_1$ and $r_4 \neq r_3$:

$$\begin{array}{lcl} r_1 & \mapsto & \langle \epsilon, \epsilon, r_3 \bullet (\mathsf{unit} \bullet r_4) \rangle \\ r_3 & \mapsto & \langle \epsilon, \epsilon, \underline{\mathsf{rjoin}\ r_2} \rangle \\ r_4 & \mapsto & \langle \epsilon, \epsilon, \mathsf{unit} \rangle \end{array}$$

6. We now perform a case distinction on $r_2 = r_4$ to determine the effect of r_3's final transition. If $r_2 = r_4$, then the step by r_3 results in the terminal global state

$$\begin{array}{lcl} r_1 & \mapsto & \langle \epsilon, \epsilon, r_3 \bullet (\mathsf{unit} \bullet r_4) \rangle \\ r_3 & \mapsto & \langle \epsilon, \epsilon, \mathsf{unit} \rangle \end{array}$$

   If $r_2 \neq r_4$, however, r_3 performs an erroneous join and the global state collapses to the error state ϵ.

Thus, the revision calculus is shown to be non-deterministic.

Replacing the informal (*fork*) side condition $r \notin s$ by the side condition $r \notin \mathit{RID}\ s$ solves the problem: if a revision r in s is about to join a nonexistent revision r', then $r' \in \mathit{RID}\ s$, and so r' is guaranteed to remain unallocated.

Remark. Right-association in the subexpression

$$\mathsf{rfork}\ (\mathsf{rjoin}\ x) \bullet (\mathsf{rjoin}\ x \bullet \mathsf{rfork}\ \mathsf{unit})$$

of P is required for the evaluation at r_1 to go through completely. If left-association is used, then a normal form is already reached in the expression $(r_3 \bullet \mathsf{unit}) \bullet \mathsf{rfork\ unit}$. The reason for this is that the application $r_3 \bullet \mathsf{unit}$ is not a value, while it must be if $(r_3 \bullet \mathsf{unit})\ \square$ is to be an evaluation context for $\mathsf{rfork\ unit}$.

More generally, let each expression e_i be able to independently normalize to a value v_i that is *not* an abstraction. By the evaluation order, the right-associated expression

$$\underline{e_1}\ (e_2\ (\dots(e_{n-1}\ e_n))\dots)$$

normalizes to

$$v_1\ (v_2\ (\dots(v_{n-1}\ v_n))\dots),$$

while the left-associated expression

$$(\dots((\underline{e_1}\ e_2)\ e_3)\dots)\ e_n$$

normalizes to

$$(\dots((v_1\ v_2)\ e_3)\dots)\ e_n$$

To us this seems like a curious asymmetry. Perhaps replacing the evaluation context definition $v\ \mathcal{E}$ by $(\lambda x.\, e)\ \mathcal{E}$ should be considered, so that function arguments are only evaluated if function applications have been established to be 'proper'.

We further note that while the counter-example we gave is no longer valid if $v\ \mathcal{E}$ is replaced by $(\lambda x.\, e)\ \mathcal{E}$, it does not solve the problem of indeterminacy. Define

$$P' = \big(\lambda x.\, (\lambda y.\, (\lambda z.\, \mathsf{rfork\ unit}) \bullet \mathsf{rfork}\ (\mathsf{rjoin}\ x)) \bullet \mathsf{rfork}\ (\mathsf{rjoin}\ x)\big) \bullet \mathsf{rfork\ unit}$$

We can check that P' exhibits the same behaviour as P, except that it performs some additional (*apply*) steps:

$$\begin{aligned}
&\big(\lambda x.\, (\lambda y.\, (\lambda z.\, \mathsf{rfork\ unit}) \bullet \mathsf{rjoin}\ x) \bullet \mathsf{rfork}\ (\mathsf{rjoin}\ x)\big) \bullet \underline{\mathsf{rfork\ unit}} \\
&\quad \to \underline{\big(\lambda x.\, (\lambda y.\, (\lambda z.\, \mathsf{rfork\ unit}) \bullet \mathsf{rjoin}\ x) \bullet \mathsf{rfork}\ (\mathsf{rjoin}\ x)\big) \bullet r_2} \\
&\quad \to (\lambda y.\, (\lambda z.\, \mathsf{rfork\ unit}) \bullet \mathsf{rjoin}\ r_2) \bullet \underline{\mathsf{rfork}\ (\mathsf{rjoin}\ r_2)} \\
&\quad \to \underline{(\lambda y.\, (\lambda z.\, \mathsf{rfork\ unit}) \bullet \mathsf{rjoin}\ r_2) \bullet r_3} \\
&\quad \to (\lambda z.\, \mathsf{rfork\ unit}) \bullet \underline{\mathsf{rjoin}\ r_2} \\
&\quad \to \underline{(\lambda z.\, \mathsf{rfork\ unit}) \bullet \mathsf{unit}} \\
&\quad \to \underline{\mathsf{rfork\ unit}} \\
&\quad \to r_4
\end{aligned}$$

Thus, a counter-example to indeterminacy is the same as in Figure 3.1, except that revision r_1 performs additional (*apply*) steps after steps *3. (fork)* and *4.* $\mathsf{rjoin}\ r_2$.

3.3 (*new*) side condition

Can a location identifier l be safely allocated in a global state s if we *only* ensure that l does not occur in any local store or snapshot? The answer is yes, assuming that program expressions are not allowed to contain location identifiers (Section 3.1).

Definition 1 (Subsuming domains)**.** *The domains of a local state* L *subsume its location identifiers, denoted* $\mathcal{S}\ L$*, when LID* $L \subseteq \mathsf{doms}\ L$*. We write* $\mathcal{S}_G\ s$ *for a global state* s *when* $\mathcal{S}\ L$ *holds for all local states* $L \in \mathsf{ran}\ s$*.*

We have that $\mathcal{S}_G$ is an execution invariant for global states s. An *execution invariant* is a property that holds for all reachable states of a transition system. Hence, it would suffice to *only* consider the domains of local stores and snapshots when allocating a fresh location identifier.

A common method for formally proving an execution invariant P is to prove an *inductive invariant* that implies P. An inductive invariant is a property I that satisfies two conditions:

- I holds for all initial states, and
- if I holds for s and $s \to s'$, then I holds for s'.

Inductive invariants do not discriminate between reachable and unreachable states, since some transitions might not be part of any execution trace. This can make finding a suitable inductive invariant significantly harder than finding an execution invariant.

Unfortunately, property $\mathcal{S}_G$ is too weak to be an inductive invariant. The culprit is rule (*join*):

$$s[\![r \mapsto \langle \sigma, \tau, \mathcal{E}[\mathsf{rjoin}\ r'] \rangle, r' \mapsto \langle \sigma', \tau', v \rangle]\!] \to_r s(r \mapsto \langle \sigma, \tau{::}\tau', \mathcal{E}[\mathsf{unit}] \rangle, r' \mapsto \bot)$$

The problem is that the inductive assumptions $\mathcal{S}\ \langle \sigma, \tau, \mathcal{E}[\mathsf{rjoin}\ r'] \rangle$ and $\mathcal{S}\ \langle \sigma', \tau', v \rangle$ are not strong enough to prove $\mathcal{S}\ \langle \sigma, \tau{::}\tau', \mathcal{E}[\mathsf{unit}] \rangle$. The reason for this is that τ' may map to a value containing some identifier l that is only subsumed by the snapshot of $s\ r'$. More precisely, the case in which

- $l \in \mathit{LID}\ v$ for some $v \in \mathsf{ran}\ \tau'$,
- $l \in \mathsf{dom}\ \sigma'$, and
- $l \notin \mathsf{dom}\ \sigma \cup \mathsf{dom}\ \tau{::}\tau'$

is not ruled out.

How should property $\mathcal{S}_G$ be strengthened? Informally, we would like to say that if revision r_1 has access to the handle of r_2 in the context of some global state s, then the set of location identifiers in r_2's snapshot is a subset of the domains at $s\ r_1$. The following definition captures this property.

Definition 2 (Subsuming accessed snapshots)**.** *Let* s *be a global state with* $r_1, r_2 \in \mathsf{dom}\ s$. *We write* $\mathcal{A}\ r_1\ r_2\ s$ *if*

$$r_2 \in \mathit{RID}\ (s\ r_1) \Rightarrow \mathit{LID}\ (s\ r_2)_\sigma \subseteq \mathsf{doms}\ (s\ r_1).$$

If we have $\mathcal{A}\ r_1\ r_2\ s$ *for all* $r_1, r_2 \in \mathsf{dom}\ s$, *then we write* $\mathcal{A}_G\ s$.

We have the following result.

Lemma 1. $\mathcal{S}_G\ s \wedge \mathcal{A}_G\ s$ *is an inductive invariant for global states* s.

Proof. Both properties hold trivially for any initial state $\epsilon(r \mapsto \langle \epsilon, \epsilon, e \rangle)$, since the stores are empty and because program expressions e do not contain any revision or location identifiers.

For proving inductive step, assume that $s \to_r s'$ with $\mathcal{S}_G\ s$ and $\mathcal{A}_G\ s$.

We first establish $\mathcal{S}_G\ s'$ by a case distinction on the step $s \to_r s'$. It suffices to show $\mathcal{S}\ (s'\ r')$ for indices r' that have been updated, i.e. for which $s\ r' \neq s'\ r'$. This is because unlike $\mathcal{A}_G$, the property $\mathcal{S}$ is not dependent on the context of the global state.

Cases (*app*), (*if-true*) and (*if-false*) are similar in that they do not introduce location identifiers into the local states, while leaving the domains unchanged:

$$\begin{array}{lll} & \mathit{LID}\ (s'\ r) & \\ \subseteq & \mathit{LID}\ (s\ r) & \\ \subseteq & \mathsf{doms}\ (s\ r) & (\mathcal{S}\ (s\ r)) \\ = & \mathsf{doms}\ (s'\ r) & \end{array}$$

Case (*new*) introduces a new identifier l, but also adds it to the domain of the local store:

$$\begin{array}{lll} & \mathit{LID}\ \langle \sigma, \tau(l \mapsto v), \mathcal{E}[l] \rangle & \\ = & \mathit{LID}\ \langle \sigma, \tau, \mathcal{E}[\mathsf{ref}\ v] \rangle \cup \{l\} & \\ \subseteq & \mathsf{dom}\ \sigma \cup \mathsf{dom}\ \tau \cup \{l\} & (\mathcal{S}\ (s\ r)) \\ \subseteq & \mathsf{dom}\ \sigma \cup \mathsf{dom}\ (\tau(l \mapsto v)) & \end{array}$$

Case (*get*) effectively shuffles location identifiers around:

$$\begin{array}{lll} & \mathit{LID}\ \langle \sigma, \tau, \mathcal{E}[(\sigma{::}\tau) l] \rangle & \\ = & \mathit{LID}\ \langle \sigma, \tau, \mathcal{E}[!l] \rangle & \\ \subseteq & \mathsf{dom}\ \sigma \cup \mathsf{dom}\ \tau & (\mathcal{S}\ (s\ r)) \end{array}$$

Case (*get*) does the same, but overwrites a value in τ, possibly causing a loss of location identifiers:

$$\begin{array}{lll} & \mathit{LID}\ \langle \sigma, \tau(l \mapsto v), \mathcal{E}[\mathsf{unit}] \rangle & \\ \subseteq & \mathit{LID}\ \langle \sigma, \tau, \mathcal{E}[l := v] \rangle & \\ \subseteq & \mathsf{dom}\ \sigma \cup \mathsf{dom}\ \tau & (\mathcal{S}\ (s\ r)) \\ \subseteq & \mathsf{dom}\ \sigma \cup \mathsf{dom}\ (\tau(l \mapsto v)) & \end{array}$$

Case (*fork*) creates two new local states:

$$\begin{array}{ll} & \mathit{LID}\ \langle \sigma, \tau, \mathcal{E}[\mathsf{r}'] \rangle \\ \subseteq & \mathit{LID}\ \langle \sigma, \tau, \mathcal{E}[\mathsf{rfork}\ e] \rangle \\ \subseteq & \mathsf{dom}\ \sigma \cup \mathsf{dom}\ \tau \qquad (\mathcal{S}\ (\mathsf{s}\ \mathsf{r})) \end{array}$$

$$\begin{array}{ll} & \mathit{LID}\ \langle \sigma{::}\tau, \epsilon, e \rangle \\ \subseteq & \mathit{LID}\ \langle \sigma, \tau, \mathcal{E}[\mathsf{rfork}\ e] \rangle \\ \subseteq & \mathsf{dom}\ \sigma \cup \mathsf{dom}\ \tau \qquad (\mathcal{S}\ (\mathsf{s}\ \mathsf{r})) \\ = & \mathsf{dom}\ (\sigma{::}\tau) \cup \mathsf{dom}\ \epsilon \end{array}$$

Case (*join*) is the only case that requires the assumption $\mathcal{A}_G\ s$:

$$\begin{array}{lll} & \mathit{LID}\ \langle \sigma, \tau{::}\tau', \mathcal{E}[\mathsf{unit}] \rangle & \\ \subseteq & \mathit{LID}\ \langle \sigma, \tau, \mathcal{E}[\mathsf{rjoin}\ \mathsf{r}'] \rangle \cup \mathit{LID}\ \tau' & \\ \subseteq & \mathsf{dom}\ \sigma \cup \mathsf{dom}\ \tau \cup \mathit{LID}\ \tau' & (\mathcal{S}\ (\mathsf{s}\ \mathsf{r})) \\ \subseteq & \mathsf{dom}\ \sigma \cup \mathsf{dom}\ \tau \cup \mathsf{dom}\ \sigma' \cup \mathsf{dom}\ \tau' & (\mathcal{S}\ (\mathsf{s}\ \mathsf{r}')) \\ = & \mathsf{dom}\ \sigma \cup \mathsf{dom}\ \tau \cup \mathsf{dom}\ \sigma \cup \mathsf{dom}\ \tau \cup \mathsf{dom}\ \tau' & (\mathcal{A}\ \mathsf{r}\ \mathsf{r}'\ \mathsf{s}) \\ = & \mathsf{dom}\ \sigma \cup \mathsf{dom}\ \tau \cup \mathsf{dom}\ \tau' & \\ = & \mathsf{dom}\ \sigma \cup \mathsf{dom}\ (\tau{::}\tau') & \end{array}$$

Case (*join*$_\epsilon$), finally, is vacuous since s' is the empty map. This concludes the subproof for $\mathcal{S}_G\ \mathsf{s}'$.

It remains to show $\mathcal{A}_G\ \mathsf{s}'$. Note two things. First, $\mathcal{A}\ \mathsf{r}\ \mathsf{r}\ \mathsf{s}'$ for all $\mathsf{r} \in \mathsf{dom}\ \mathsf{s}'$ follows from $\mathcal{S}_G\ \mathsf{s}'$.[3] Second, if $\mathsf{s}'\ \mathsf{r}_1 = \mathsf{s}\ \mathsf{r}_1$ and $\mathsf{s}'\ \mathsf{r}_2 = \mathsf{s}\ \mathsf{r}_2$, then $\mathcal{A}\ \mathsf{r}_1\ \mathsf{r}_2\ \mathsf{s}'$ follows directly from $\mathcal{A}\ \mathsf{r}_1\ \mathsf{r}_2\ \mathsf{s}$. Hence, it suffices to show that $\mathcal{A}\ \mathsf{r}_1\ \mathsf{r}_2\ \mathsf{s}'$ for all *distinct* $\mathsf{r}_1, \mathsf{r}_2 \in \mathsf{dom}\ \mathsf{s}'$ with $\mathsf{s}'\ \mathsf{r}_1 \neq \mathsf{s}\ \mathsf{r}_1$ or $\mathsf{s}'\ \mathsf{r}_2 \neq \mathsf{s}\ \mathsf{r}_2$.

We again perform a case analysis on the step $\mathsf{s} \rightarrow_\mathsf{r} \mathsf{s}'$. Case (*join*$_\epsilon$) again holds vacuously.

Let us consider the six rules that modify *only* the revision r. We have to show $\mathcal{A}\ \mathsf{r}\ \mathsf{r}^\star\ \mathsf{s}'$ and $\mathcal{A}\ \mathsf{r}^\star\ \mathsf{r}\ \mathsf{s}'$ for arbitrary $\mathsf{r}^\star \in \mathsf{dom}\ \mathsf{s}'$ with $\mathsf{r}^\star \neq \mathsf{r}$.

1. $\mathcal{A}\ \mathsf{r}\ \mathsf{r}^\star\ \mathsf{s}'$*:* Assume $\mathsf{r}^\star \in \mathit{RID}\ (\mathsf{s}'\ \mathsf{r})$. Since none of the six rules under consideration could have introduced $\mathsf{r}^\star$, $\mathsf{r}^\star \in \mathit{RID}\ (\mathsf{s}\ \mathsf{r})$. $\mathcal{A}\ \mathsf{r}\ \mathsf{r}^\star\ \mathsf{s}'$ is shown as follows:

$$\begin{array}{lll} & \mathit{LID}\ (\mathsf{s}'\ \mathsf{r}^\star)_\sigma & \\ = & \mathit{LID}\ (\mathsf{s}\ \mathsf{r}^\star)_\sigma & (\mathsf{r}^\star \text{ was not updated}) \\ \subseteq & \mathsf{doms}\ (\mathsf{s}\ \mathsf{r}) & (\mathsf{r}^\star \in \mathit{RID}\ (\mathsf{s}\ \mathsf{r}) \text{ and } \mathcal{A}\ \mathsf{r}\ \mathsf{r}^\star\ \mathsf{s}) \\ \subseteq & \mathsf{doms}\ (\mathsf{s}'\ \mathsf{r}) & (\text{steps do not delete from domains}) \end{array}$$

[3]It seems obvious that a revision can never have access to its own handle. However, we need not prove it.

2. $\mathcal{A}\ \mathsf{r}^\star\ \mathsf{r}\ \mathsf{s}'$*:* Assume $\mathsf{r} \in \mathit{RID}\ (\mathsf{s}'\ \mathsf{r}^\star)$. We also have $\mathsf{r} \in \mathit{RID}\ (\mathsf{s}\ \mathsf{r}^\star)$ since $\mathsf{r}^\star$ was not updated. We have

$$\begin{array}{lll} & \mathit{LID}\ (\mathsf{s}'\ \mathsf{r})_\sigma & \\ = & \mathit{LID}\ (\mathsf{s}\ \mathsf{r})_\sigma & \text{(none of the rules modify the snapshot)} \\ \subseteq & \mathsf{doms}\ (\mathsf{s}\ \mathsf{r}^\star) & (\mathsf{r} \in \mathit{RID}\ (\mathsf{s}\ \mathsf{r}^\star) \text{ and } \mathcal{A}\ \mathsf{r}^\star\ \mathsf{r}\ \mathsf{s}) \\ = & \mathsf{doms}\ (\mathsf{s}'\ \mathsf{r}^\star) & (\mathsf{r}^\star \text{ was not updated}) \end{array}$$

For (*join*), we also only have to show $\mathcal{A}\ \mathsf{r}\ \mathsf{r}^\star\ \mathsf{s}'$ and $\mathcal{A}\ \mathsf{r}^\star\ \mathsf{r}\ \mathsf{s}'$ for arbitrary $\mathsf{r}^\star \in \mathsf{dom}\ \mathsf{s}'$ with $\mathsf{r}^\star \neq \mathsf{r}$:

1. $\mathcal{A}\ \mathsf{r}\ \mathsf{r}^\star\ \mathsf{s}'$*:* Assume $\mathsf{r}^\star \in \mathit{RID}\ (\mathsf{s}'\ \mathsf{r})$. We perform a case distinction on $\mathsf{r}^\star \in \mathit{RID}\ \tau'$.

   If $\mathsf{r}^\star \in \mathit{RID}\ \tau'$, then $\mathcal{A}\ \mathsf{r}\ \mathsf{r}^\star\ \mathsf{s}'$ is shown as follows:

$$\begin{array}{lll} & \mathit{LID}\ (\mathsf{s}'\ \mathsf{r}^\star)_\sigma & \\ = & \mathit{LID}\ (\mathsf{s}\ \mathsf{r}^\star)_\sigma & (\mathsf{r}^\star \text{ was not updated}) \\ \subseteq & \mathsf{doms}\ (\mathsf{s}\ \mathsf{r}') & (\mathsf{r}^\star \in \mathit{RID}\ (\mathsf{s}\ \mathsf{r}') \text{ and } \mathcal{A}\ \mathsf{r}'\ \mathsf{r}^\star\ \mathsf{s}) \\ = & \mathsf{dom}\ \sigma' \cup \mathsf{dom}\ \tau' & \\ \subseteq & \mathit{LID}\ \sigma' \cup \mathsf{dom}\ \tau' & \\ \subseteq & \mathsf{dom}\ \sigma \cup \mathsf{dom}\ \tau \cup \mathsf{dom}\ \tau' & (\mathsf{r}' \in \mathit{RID}\ (\mathsf{s}\ \mathsf{r}) \text{ and } \mathcal{A}\ \mathsf{r}\ \mathsf{r}'\ \mathsf{s}) \\ = & \mathsf{dom}\ \sigma \cup \mathsf{dom}\ (\tau{::}\tau') & \\ = & \mathsf{doms}\ (\mathsf{s}'\ \mathsf{r}) & \end{array}$$

   If $\mathsf{r}^\star \notin \mathit{RID}\ \tau'$, then $\mathsf{r}^\star \in \mathit{RID}\ (\mathsf{s}\ \mathsf{r})$. We have

$$\begin{array}{lll} & \mathit{LID}\ (\mathsf{s}'\ \mathsf{r}^\star)_\sigma & \\ = & \mathit{LID}\ (\mathsf{s}\ \mathsf{r}^\star)_\sigma & (\mathsf{r}^\star \text{ was not updated}) \\ \subseteq & \mathsf{doms}\ (\mathsf{s}\ \mathsf{r}) & (\mathsf{r}^\star \in \mathit{RID}\ (\mathsf{s}\ \mathsf{r}) \text{ and } \mathcal{A}\ \mathsf{r}\ \mathsf{r}^\star\ \mathsf{s}) \\ \subseteq & \mathsf{doms}\ (\mathsf{s}'\ \mathsf{r}) & \end{array}$$

2. $\mathcal{A}\ \mathsf{r}^\star\ \mathsf{r}\ \mathsf{s}'$*:* Exactly like the $\mathcal{A}\ \mathsf{r}^\star\ \mathsf{r}\ \mathsf{s}'$ case for the six rules covered before.

That leaves only case (*fork*). Since fork creates two new local states at r and r', there is a total of six cases, in which $\mathsf{r}^\star \in \mathsf{dom}\ \mathsf{s}'$ again denotes an arbitrary *unchanged* revision ($\mathsf{r}^\star \neq \mathsf{r}$ and $\mathsf{r}^\star \neq \mathsf{r}'$):

1. $\mathcal{A}\ \mathsf{r}\ \mathsf{r}^\star\ \mathsf{s}'$*:* Analogous to the $\mathcal{A}\ \mathsf{r}^\star\ \mathsf{r}\ \mathsf{s}'$ case for the six rules covered before, using $\mathsf{r}^\star \neq \mathsf{r}'$.

2. $\mathcal{A}\ \mathsf{r}^\star\ \mathsf{r}\ \mathsf{s}'$*:* Exactly like the $\mathcal{A}\ \mathsf{r}^\star\ \mathsf{r}\ \mathsf{s}'$ case for the six rules covered before.

3. $\mathcal{A}\ \mathsf{r}'\ \mathsf{r}^\star\ \mathsf{s}'$*:* Assume $\mathsf{r}^\star \in \mathit{RID}\ (\mathsf{s}'\ \mathsf{r}')$. This implies $\mathsf{r}^\star \in \mathit{RID}\ (\mathsf{s}\ \mathsf{r})$, since r was the forker of r'. Since $\mathsf{s}'\ \mathsf{r}^\star = \mathsf{s}\ \mathsf{r}^\star$ and $\mathcal{A}\ \mathsf{r}\ \mathsf{r}^\star\ \mathsf{s}$, $\mathit{LID}\ (\mathsf{s}'\ \mathsf{r}^\star)_\sigma \subseteq \mathsf{doms}\ (\mathsf{s}\ \mathsf{r}) = \mathsf{doms}\ (\mathsf{s}'\ \mathsf{r}')$.

4. $\mathcal{A}\ r^{\star}\ r'\ s'$*:* Holds vacuously: the freshness condition $r' \notin \mathit{RID}\ s$ implies that the assumption $r' \in \mathit{RID}\ (s'\ r^{\star}) = \mathit{RID}\ (s\ r^{\star})$ is unsatisfiable.

5. $\mathcal{A}\ r\ r'\ s'$*:* We know that $r' \in \mathit{RID}\ (s'\ r)$. The goal is shown by

$$\begin{array}{ll} & \mathit{LID}\ (s'\ r')_{\sigma} \\ = & \mathit{LID}\ (\sigma{::}\tau) \\ \subseteq & \mathit{LID}\ \sigma \cup \mathit{LID}\ \tau \\ \subseteq & \mathit{LID}\ (s'\ r) \\ \subseteq & \mathsf{doms}\ (s'\ r) \qquad (\mathcal{S}\ (s'\ r)) \end{array}$$

6. $\mathcal{A}\ r'\ r\ s'$*:* Suppose that $r \in \mathit{RID}\ (s'\ r')$. We have

$$\begin{array}{ll} & \mathit{LID}\ (s'\ r)_{\sigma} \\ \subseteq & \mathit{LID}\ (s'\ r) \\ \subseteq & \mathsf{doms}\ (s'\ r) \qquad (\mathcal{S}\ (s'\ r)) \\ = & \mathsf{dom}\ \sigma \cup \mathsf{dom}\ \tau \\ = & \mathsf{dom}\ (\sigma{::}\tau) \cup \mathsf{dom}\ \epsilon \\ = & \mathsf{doms}\ (s'\ r') \end{array}$$

□

Corollary 1. *$\mathcal{S}_G\ s$ is an execution invariant for global states s.*

Let the family $\rightarrow'_r$ be defined as $\rightarrow_r$ (with $r \notin \mathit{RID}\ s$ as the side condition on (*fork*)), except that

- the side condition for (*new*) is replaced with $l \notin \bigcup \{\mathsf{doms}\ l' \mid l' \in \mathsf{ran}\ s\}$, and
- the side conditions for (*get*) and (*set*) are omitted.

We have that $\rightarrow'_r$ and $\rightarrow_r$ define the same transition system.

Lemma 2. *Let s be a reachable state in the original system. Then*

$$\forall r\ s'.\ s \rightarrow_r s' \iff s \rightarrow'_r s'.$$

Chapter 4

Determinacy

The revision calculus is determinate (modulo renaming-equivalence): for any program expression, there is at most one final state. In this chapter we build towards a proof of this claim. We split the proof into three parts:

1. We prove what we call *rule determinism*: at any point, a fixed revision r can perform a step according to at most one rule (Section 4.1). This relies subtly on the definition of execution contexts, and is not proven in Burckhardt and Leijen's account.

2. By using rule determinism and analyzing the behaviour of pairs of diverging steps, we prove *strong local confluence* (Section 4.2):

$$s_2 \leftarrow_r s_1 \rightarrow_{r'} s_2' \Longrightarrow s_2 \rightarrow_{r'}^{=} s_3 \approx s_3' \leftarrow_r^{=} s_2'.$$

   The main proof in this section is highly indebted to Burckhardt and Leijen.

3. From strong local confluence we prove *confluence (modulo renaming-equivalence)*:

$$s_2 \leftarrow^* s_1 \approx s_1' \rightarrow^* s_2' \Longrightarrow s_2 \rightarrow^* s_3 \approx s_3' \leftarrow^* s_2',$$

   which has determinacy as its corollary (Section 4.3.1).

   The first part of the proof of confluence roughly follows the outline described by Burckhardt and Leijen. For completeness, we work out the details of this proof. Our proof differs from Burckhardt and Leijen's proof in one way, however: we avoid reasoning about reductions on the level of equivalence classes. For the purposes of formalization, we consider this a simplification. We provide a brief comparison of the two approaches in Section 4.3.2.

4.1 Rule determinism

We begin by showing that the plug operation is injective for a fixed context $\mathcal{E}$.

Lemma 3 (Plug is injective)**.**

$$\mathcal{E}[e] = \mathcal{E}[e'] \iff e = e'.$$

Proof. Direction $\Longleftarrow$ is trivial. Direction $\Longrightarrow$ follows by a straightforward structural induction on $\mathcal{E}$. $\square$

Lemma 4 (Redex-completions are not values)**.** *If* r *is a redex, then* $\mathcal{E}[r] \notin \mathit{Val}$*.*

Proof. By a case distinction on the context $\mathcal{E}$. If $\mathcal{E} = \square$, the conclusion follows from the fact that none of the constructors for *Val* can be used to construct a redex, while $\square[r] = r$ is a redex. In all other cases, the result of $\mathcal{E}[r]$ is some complex expression (such as an application or branching statement) that similarly cannot be constructed using the constructors for *Val*. $\square$

One immediate consequence of Lemma 4 is that active revisions cannot be joined. For if a revision can still perform some step, then its expression is of the form $\mathcal{E}[r]$ (with r a redex), and a revision can only be subject to a join if its expression has resolved to a value.

Suppose that we are given an expression e containing redexes. Intuitively, the following formal definition describes how the active site of e should be found, and it also provides the context in which it occurs.

Definition 3 (Decomposition)**.** *The* decomposition *of an expression* e *into a context* $\mathcal{E}$ *and a (reducible) expression* r*, denoted* $e \rhd (\mathcal{E}, r)$*, is defined inductively by the rules in Figure 4.1.*

$$\mathit{rdx}: \frac{\mathit{redex}\ e}{e \rhd (\square, e)} \qquad \mathit{ite}: \frac{\neg(\mathit{redex}\ (e_1\ ?\ e_2 : e_3)) \quad e_1 \rhd (\mathcal{E}, r)}{e_1\ ?\ e_2 : e_3 \rhd (\mathcal{E}\ ?\ e_2 : e_3, r)}$$

$$\mathit{apply}_L: \frac{\neg(\mathit{redex}\ (e_1\ e_2)) \quad e_1 \rhd (\mathcal{E}, r)}{e_1\ e_2 \rhd (\mathcal{E}\ e_2, r)} \qquad \mathit{ref}: \frac{\neg(\mathit{redex}\ (\mathsf{ref}\ e)) \quad e \rhd (\mathcal{E}, r)}{\mathsf{ref}\ e \rhd (\mathsf{ref}\ \mathcal{E}, r)}$$

$$\mathit{apply}_R: \frac{\neg(\mathit{redex}\ (v\ e_2)) \quad e_2 \rhd (\mathcal{E}, r)}{v\ e_2 \rhd (v\ \mathcal{E}, r)} \qquad \mathit{get}: \frac{\neg(\mathit{redex}\ (!e)) \quad e \rhd (\mathcal{E}, r)}{!e \rhd (!\mathcal{E}, r)}$$

$$\mathit{set}_L: \frac{\neg(\mathit{redex}\ (e_1 := e_2)) \quad e_1 \rhd (\mathcal{E}, r)}{e_1 := e_2 \rhd (\mathcal{E} := e_2, r)} \qquad \mathit{join}: \frac{\neg(\mathit{redex}\ (\mathsf{rjoin}\ e)) \quad e \rhd (\mathcal{E}, r)}{\mathsf{rjoin}\ e \rhd (\mathsf{rjoin}\ \mathcal{E}, r)}$$

$$\mathit{set}_R: \frac{\neg(\mathit{redex}\ (l := e_2)) \quad e_2 \rhd (\mathcal{E}, r)}{l := e_2 \rhd (l := \mathcal{E}, r)}$$

Figure 4.1: The decomposition rules.

It is clear that any decomposition ends with an application of rule *rdx*. This means that a normal form cannot be decomposed. An alternative sensible definition would allow one to derive $e \rhd \langle \mathcal{E}, e' \rangle$ whenever $\mathcal{E}[e'] = e$. However, our strict definition is sufficient since we are only decomposing redex-completions $\mathcal{E}[r]$.

Example 2. Recall the term $e = ((\lambda x.\, x)\ x)\ ((\lambda y.\, y)\ y)$ from Example 1. We can derive $e \rhd (\Box\ ((\lambda y.\, y)\ y), (\lambda x.\, x)\ x)$ as follows:

$$\dfrac{\neg(\textit{redex}\ e) \qquad \dfrac{\textit{redex}\ ((\lambda x.\, x)\ x)}{(\lambda x.\, x)\ x \rhd (\Box, (\lambda x.\, x)\ x)}\,\textit{rdx}}{e \rhd (\Box\ ((\lambda y.\, y)\ y), (\lambda x.\, x)\ x)}\,\textit{apply}_L$$

By contrast, it is easy to see that $e \rhd (((\lambda x.\, x)\ x)\ \Box, (\lambda y.\, y)\ y)$ cannot match the conclusion of any of the derivation rules.

The following lemma demonstrates a fundamental connection between plugging and decomposing (for redexes r): if e decomposes into $\mathcal{E}$ and r, then r can be plugged back into $\mathcal{E}$ to recover e; and if plugging r into $\mathcal{E}$ results in e, then there exists some derivation that decomposes e into $\mathcal{E}$ and r again.

Lemma 5 (Plug-decomposition equivalence)**.** *For any redex* r

$$e \rhd (\mathcal{E}, r) \iff \mathcal{E}[r] = e.$$

Proof. We discuss each direction in turn.

($\Longrightarrow$) By rule induction on $e \rhd (\mathcal{E}, r)$.

(*rdx*) By the conclusion of the rule, $\mathcal{E} = \Box$ and $r = e$, hence $\mathcal{E}[r] = r = e$.

($\textit{apply}_L$) By the conclusion of the rule, $e = e_1\ e_2$, $\mathcal{E} = \mathcal{E}'\ e_2$ and $r = r'$ (for some e_1, e_2, $\mathcal{E}'$ and r'), and by the right premiss of the rule, $e_1 \rhd (\mathcal{E}', r')$. We must thus show $(\mathcal{E}'\ e_2)[r'] = (\mathcal{E}'[r']\ e_2) = e_1\ e_2$, which holds since the induction hypothesis can be used to show that $\mathcal{E}'[r'] = e_1$.

The remaining cases are similar to $\textit{apply}_L$.

($\Longleftarrow$) By structural induction on the context $\mathcal{E}$.

($\mathcal{E} = \Box$) We have $\Box[r] = r = e$. We thus have to show $r \rhd (\Box, r)$, which follows from rule *rdx*.

($\mathcal{E} = \mathcal{E}'\ e_2$) We have $(\mathcal{E}'\ e_2)[r] = (\mathcal{E}'[r])\ e_2 = e$. We thus have to show $(\mathcal{E}'[r])\ e_2 \rhd (\mathcal{E}'\ e_2,\ r)$. We apply rule $\textit{apply}_L$, which requires us to show that (1) $(\mathcal{E}'[r])\ e_2$ is not a redex and that (2) $\mathcal{E}'[r] \rhd (\mathcal{E}', r)$. Requirement (1) follows from the following impossibility: for the application $(\mathcal{E}'[r])\ e_2$ to be a redex, $(\mathcal{E}'[r])$ must be an abstraction, and thus more generally a value, contradicting Lemma 4. Requirement (2) follows from the induction hypothesis.

The remaining cases are similar to case ($\mathcal{E} = \mathcal{E}'\ e_2$).

□

Lemma 5 does not rule out the possibility where an expression can be decomposed in two different ways. The following lemma demonstrates that always at most one decomposition is possible.[1]

Lemma 6 (Unique decomposition)**.** *Assume* $e \rhd (\mathcal{E}_1, r_1)$ *and* $e \rhd (\mathcal{E}_2, r_2)$*. Then* $\mathcal{E}_1 = \mathcal{E}_2$*.*

Proof. By rule induction on $e \rhd (\mathcal{E}_1, r_1)$, letting $\mathcal{E}_2$ be arbitrary.

(*rdx*) e is a redex and $\mathcal{E}_1 = \square$. Since e is a redex, $e \rhd (\mathcal{E}_2,\ r_2)$ can only have been derived using rule *rdx* as well, forcing $\mathcal{E}_2 = \square$.

(*apply$_L$*) e is an application $e_1\ e_2$ with $e_1 \rhd (\mathcal{E}_1, r_1)$. Since e is an application, $e \rhd (\mathcal{E}_2, r_2)$ must have been derived using either rule *apply$_L$* or *apply$_R$*.

If $e \rhd (\mathcal{E}_2, r_2)$ was derived using rule *apply$_L$*, then $e_1 \rhd (\mathcal{E}_2, r_2)$ by the rule's premiss, and consequently $\mathcal{E}_1 = \mathcal{E}_2$ by the induction hypothesis.

If $e \rhd (\mathcal{E}_2, r_2)$ was derived using *apply$_R$*, then $e_1 \in \mathit{Val}$. By $e_1 \rhd (\mathcal{E}_1, r_1)$ and Lemma 5, $\mathcal{E}_1[r_1] = e_1$, so by Lemma 4, $e_1 \notin \mathit{Val}$. Contradiction.

The remaining cases are similar to case *apply$_L$*. □

We can now state our main result: if two redex-completions are equal, then so are the arguments of the plug operation.

Lemma 7 (Completion equivalence)**.** *Let* r *and* r' *be redexes. We have*

$$\mathcal{E}[r] = \mathcal{E}'[r'] \iff \mathcal{E} = \mathcal{E}' \wedge r = r'.$$

Proof. Direction $\Longleftarrow$ is trivial. For direction $\Longrightarrow$, assume $\mathcal{E}[r] = \mathcal{E}'[r']$. By Lemma 5, we have $\mathcal{E}[r] \rhd (\mathcal{E}, r)$ and $\mathcal{E}'[r'] \rhd (\mathcal{E}', r')$ (using $\mathcal{E}[r] = \mathcal{E}[r]$ and $\mathcal{E}'[r'] = \mathcal{E}'[r']$, respectively). From Lemma 6 and $\mathcal{E}[r] = \mathcal{E}'[r']$ we then obtain $\mathcal{E} = \mathcal{E}'$. By Lemma 3 and $\mathcal{E}[r] = \mathcal{E}[r']$, finally, we derive $r = r'$. □

Lemma 7 is crucial for proving determinacy. It implies that if the source state of a step matches the source state of a certain rule, then we can infer that this rule must have been performed.

Lemma 8 (Rule determinism)**.** *We have the following equivalences:*

1. $s[\![r \mapsto \langle \sigma, \tau, \mathcal{E}[(\lambda x.e)\ v]\rangle]\!] \rightarrow_r s' \iff$
$s' = s(r \mapsto \langle \sigma, \tau, \mathcal{E}[[v/x]e]\rangle)$

[1] $r_1 = r_2$ can be shown using context injectivity (Lemma 3).

2. $s[\![r \mapsto \langle \sigma, \tau, \mathcal{E}[\mathsf{true} \mathbin{?} e_1 : e_2]\rangle]\!] \to_r s' \iff$
   $s' = s(r \mapsto \langle \sigma, \tau, \mathcal{E}[e_1]\rangle)$

3. $s[\![r \mapsto \langle \sigma, \tau, \mathcal{E}[\mathsf{false} \mathbin{?} e_1 : e_2]\rangle]\!] \to_r s' \iff$
   $s' = s(r \mapsto \langle \sigma, \tau, \mathcal{E}[e_2]\rangle)$

4. $s[\![r \mapsto \langle \sigma, \tau, \mathcal{E}[\mathsf{ref}\ v]\rangle]\!] \to_r s' \iff$
   $\exists l.\, l \notin s \wedge s' = s(r \mapsto \langle \sigma, \tau(l \mapsto v), \mathcal{E}[l]\rangle)$

5. $s[\![r \mapsto \langle \sigma, \tau, \mathcal{E}[!l]\rangle]\!] \to_r s' \iff$
   $s' = s(r \mapsto \langle \sigma, \tau, \mathcal{E}[(\sigma{::}\tau)\ l]\rangle)$

6. $s[\![r \mapsto \langle \sigma, \tau, \mathcal{E}[l := v]\rangle]\!] \to_r s' \iff$
   $s' = s(r \mapsto \langle \sigma, \tau(l \mapsto v), \mathcal{E}[\mathsf{unit}]\rangle)$

7. $s[\![r \mapsto \langle \sigma, \tau, \mathcal{E}[\mathsf{rfork}\ e]]\rangle]\!] \to_r s' \iff$
   $\exists r'.\, r' \notin s \wedge s' = s(r \mapsto \langle \sigma, \tau, \mathcal{E}[r']\rangle, r' \mapsto \langle \sigma{::}\tau, \epsilon, e\rangle)$

8. $s[\![r \mapsto \langle \sigma, \tau, \mathcal{E}[\mathsf{rjoin}\ r']\rangle, r' \mapsto \langle \sigma', \tau', v\rangle]\!] \to_r s' \iff$
   $s' = s(r \mapsto \langle \sigma, \tau{::}\tau', \mathcal{E}[\mathsf{unit}]\rangle, r' \mapsto \bot)$

9. $s[\![r \mapsto \langle \sigma, \tau, \mathcal{E}[\mathsf{rjoin}\ r']\rangle, r' \mapsto \bot]\!] \to_r s' \iff$
   $s' = \epsilon$

Proof. Direction $\Longleftarrow$ of each sublemma follows directly from the operational semantics. For the $\Longrightarrow$ direction, assume that $s\ r$ matches two rules, where one rule matches $(s\ r)_e$ against the redex-completion $\mathcal{E}_1[r_1]$ and the other rule matches $(s\ r)_e$ against the redex-completion $\mathcal{E}_2[r_2]$ (with r_1 and r_2 redexes). By Lemma 7, $r_1 = r_2$. Thus either the rules are the same or one rule is (*join*) and the other rule is (*join*$_\epsilon$): the latter case is impossible since it would imply that the joinee r' is both defined and undefined. □

4.2 Strong local confluence

For proving strong local confluence, we follow Burckhardt and Leijen in first proving the preliminary lemma below.

Lemma 9 (Local determinism)**.** $s_2 \leftarrow_r s_1 \to_r s_2' \Longrightarrow s_2 \approx s_2'$.

Proof. By a case analysis on the left step $s_2 \leftarrow_r s_1$. In every case other than (*new*) and (*fork*), we have $s_2' = s_2$ using Lemma 8, and therefore $s_2 \approx s_2'$ by reflexivity of $\approx$.

In case (*new*), we are given that $s_2 = s(r \mapsto \langle \sigma, \tau(l \mapsto v), \mathcal{E}[l]\rangle)$ (for $l \notin \mathit{LID}\ s$), and by Lemma 8, we can infer $s_2' = s(r \mapsto \langle \sigma, \tau(l' \mapsto v), \mathcal{E}[l']\rangle)$ (for $l' \notin \mathit{LID}\ s$). Define $\alpha = \mathit{id}$ and $\beta = \mathit{id}(l := l', l' := l)$. We have that β is bijective and that $\alpha\ (\beta\ s_2) = s_2'$, hence $s_2 \approx s_2'$.

The argument for case (*fork*) is analogous to case (*new*). □

The following proposition states two obvious preservation laws for freshness of location and revision identifiers.

Proposition 1 (Freshness preservation laws)**.** *Suppose* $s \to s'$*. Then*

1. *RID* $s' \subseteq$ *RID* s *if* $s \to s'$ *is not a (fork) step, and*
2. *LID* $s' \subseteq$ *LID* s *if* $s \to s'$ *is not a (new) step.*

Proof. By a trivial case distinction on $s \to s'$. □

In addition, the following lemma is needed for proving strong local confluence. It implies that there always exists a fresh identifier, on the condition that the universes of revision and location identifiers are infinite.

Lemma 10 (Finiteness)**.** *If* s *is reachable, then LID* s *and RID* s *are finite.*

Proof. Since s is reachable, there exists a reduction sequence $s_0 \to^n s$, with $s_0 = \epsilon(r \mapsto \langle \epsilon, \epsilon, e \rangle$ some initial state. The proof proceeds by induction on the length n. If $n = 0$, $s_0 = s$. Since program expressions do not contain identifiers, *RID* $s_0 = \{r\}$ and *LID* $s_0 = \varnothing$, hence *RID* s and *LID* s are finite. For the inductive step $s_0 \to^n s' \to s$ with *RID* s' and *LID* s' finite, a simple case analysis on the step $s' \to s$ shows that at most one location or revision identifier is introduced (through (*new*) or (*fork*), respectively). Hence *RID* s and *LID* s are finite also. □

We will represent binary relations and reduction patterns using graphs. To this end we introduce the notions of a segment and of a reduction diagram.

Notation 1 (Segment)**.** An *elementary segment* is an edge between two logical variables, and represents a binary relation. A segment's length is irrelevant. We introduce the following elementary segments S for global states s and s', where $\mathfrak{I}(S)$ denotes the *interpretation* of S:

segment S	$\mathfrak{I}(S)$
$s \longrightarrow s'$	$s \to s$
$s \xrightarrow{=} s'$	$s \to^{=} s'$
$s \xrightarrow{n} s'$	$s \to^{n} s$
$s \twoheadrightarrow s'$	$s \to^{*} s'$
$s =\!=\!= s'$	$s = s'$
$s - \approx - s'$	$s \approx s'$
$s \xrightarrow{r} s'$	$s \to_r s'$

In addition, we sometimes add additional annotations segments, whose meaning is then fixed by accompanying text.

Segments can be composed to form a *complex segment*: if $x \overset{\dagger}{—} y$ and $y \overset{\star}{—} z$ are segments, then so is $x \overset{\dagger}{—} y \overset{\star}{—} z$. For complex segments we define

$$\mathfrak{I}(x \overset{\dagger}{—} y \overset{\star}{—} z) = \mathfrak{I}(x \overset{\dagger}{—} y) \wedge \mathfrak{I}(y \overset{\star}{—} z).$$

Finally, segments can be oriented vertically, in which case

$$\mathfrak{I}(\begin{smallmatrix} x \\ | \\ y \end{smallmatrix}) = \mathfrak{I}(x — y).$$

Definition 4 (Reduction diagram). *Let* $x_1, x_2, \ldots, x_n$ *denote the* $n \geqslant 0$ *logical variables occurring in the arbitrary segments* $a - y \to b$ *and* $a - x \to c$ *(excluding* a, b *and* c*). Similarly, let* $y_1, y_2, \ldots, y_m$ *denote the* $m \geqslant 0$ *logical variables occurring in the arbitrary segments* $c - w \to d$ *and* $b - z \to d$ *(excluding* b, c *and* d*).*

A reduction diagram *is a rectangular graph*

$$\begin{tikzcd} a \arrow[r, "y"] \arrow[d, "x"] & b \arrow[d, "z"] \\ c \arrow[r, "w"] & d \end{tikzcd}$$

expressing

$$\forall a\ b\ c\ x_1\ \ldots\ x_n. \mathfrak{I}(a - y - b) \wedge \mathfrak{I}(a - x \to c) \Longrightarrow$$
$$\exists d\ y_0\ \ldots\ y_m. \mathfrak{I}(b - z \to d) \wedge \mathfrak{I}(c - w \to d).$$

Example 3. The diagram

$$\begin{tikzcd} a \arrow[r, dash, "\approx"] \arrow[d] & b \arrow[r, "r"] & c \arrow[d, equal] \\ d \arrow[rr, two heads] & & e \end{tikzcd}$$

expresses the false proposition

$$\forall a\ b\ c\ d.\ a \approx b \wedge b \to_r c \wedge a \to d \Longrightarrow c = e \wedge d \to^* e.$$

In natural language:

> Assume $a \approx b$, where a takes some diverging step $a \to d$ and b takes some diverging step $b \to_r c$. Then there exists an e such that the reductions can converge again using $c = e$ and $d \to^* e$.

Lemma 11 (Strong local confluence)**.** *Let s_1 be a reachable state, and assume that the universes of location and revision identifiers are infinite. We have*

$$s_2 \leftarrow_{r} s_1 \rightarrow_{r'} s_2' \Longrightarrow \exists s_3\ s_3'.\ s_2 \rightarrow^{=}_{r'} s_3 \approx s_3' \leftarrow^{=}_{r} s_2'.$$

Proof. If $r = r'$, then the conclusion follows trivially from Lemma 9.

For the $r \neq r'$ case, we perform a case analysis on the left step $s_2 \leftarrow_r s_1$. For the right step we then consider each possible rule that has *not* yet been considered as the left step: this is permitted since the cases on the left and right step are symmetric.

($join_\epsilon$) Suppose revision r joins a non-existent revision r'' in the left step. We distinguish two cases for the right step $s_1 \rightarrow_{r'} s_2'$, performed by revision r': it is either a ($join_\epsilon$) step, or not (denoted by ($\overline{join_\epsilon}$)). We discuss both cases in turn.

($join_\epsilon$) Suppose revision r' erroneously joins a non-existent revision r''' in the right step. Then the two steps meet directly in the error state ϵ (Figure 4.2, left).

($\overline{join_\epsilon}$) We have $s_2'\ r = s_1\ r$ and $s_2'\ r'' = \perp$ (r'' could not have been forked, since $r'' \in RID\ (s_1\ r)$). Thus, revision r can still erroneously join r'' in state s_2' and collapse the global state to ϵ (Figure 4.2, right).

$$\begin{tikzcd}
s_1 \arrow[rr, "r' : \mathrm{join}_\epsilon(r''')"] \arrow[d, "r : \mathrm{join}_\epsilon(r'')"] & & s_2' \arrow[d, equal] \\
s_2 \arrow[r, equal] & \epsilon \arrow[r, equal] & \epsilon
\end{tikzcd}
\qquad
\begin{tikzcd}
s_1 \arrow[rr, "r' : \overline{\mathrm{join}_\epsilon}"] \arrow[d, "r : \mathrm{join}_\epsilon(r'')"] & & s_2' \arrow[d, "r : \mathrm{join}_\epsilon(r'')"'] \\
s_2 \arrow[r, equal] & \epsilon \arrow[r, equal] & \epsilon
\end{tikzcd}$$

Figure 4.2: Strong local confluence: case ($join_\epsilon$).

($join$) Suppose revision r successfully joins a revision r'' in the left step. We again distinguish two cases for the right step $s_1 \rightarrow_{r'} s_2'$, performed by revision r': either r' also succesfully joins r'', or not (denoted by ($\overline{join(r'')}$)).

($join(r'')$) r' also joins revision r'' specifically, so that both $s_2\ r'' = \perp$ and $s_2'\ r'' = \perp$. Moreover, $s_2\ r' = s_1\ r'$ and $s_2'\ r = s_1\ r$. Hence $s_2 \rightarrow_{r'} \epsilon \leftarrow_{r} s_2'$ (Figure 4.3, left).

($\overline{join(r'')}$) r' does not join revision r''. In this case, the diverging steps commute, since the left step can still be performed in s_2', and the right step can still be performed in s_2 (Figure 4.3, right).

The details of this case involve reasoning about freshness. Namely, if the right step forks r''', then r''' can still be forked in s_2, since the left step did not introduce any revision identifiers (Proposition 1). A similar argument applies if the right step is a (*new*) step.

$$\begin{tikzcd}
s_1 \arrow[rr, "r' : \mathsf{join}(r'')"] \arrow[d, "r : \mathsf{join}(r'')"] & & s_2' \arrow[d, "r : \mathsf{join}_\epsilon(r'')"'] \\
s_2 \arrow[r, "r' : \mathsf{join}_\epsilon(r'')"'] & \epsilon \arrow[r, equal] & \epsilon
\end{tikzcd}
\qquad
\begin{tikzcd}
s_1 \arrow[rr, "r' : \overline{\mathsf{join}(r'')}"] \arrow[d, "r : \mathsf{join}(r'')"] & & s_2' \arrow[d, "r : \mathsf{join}(r'')"'] \\
s_2 \arrow[r, "r' : \overline{\mathsf{join}(r'')}"'] & s_3 \arrow[r, equal] & s_3'
\end{tikzcd}$$

Figure 4.3: Strong local confluence: case (*join*).

(*local*) Under a (*local*) step we here understand any step that is an (*apply*), (*ifTrue*), (*ifFalse*), (*get*) or (*set*) step: we exclude (*new*) for technical reasons. The right step is a (*) (*local*), (*new*) or (*fork*) step. It is easy to see that both steps commute (Figure 4.4). As in the previous case, Proposition 1 is used if the right step is a (*new*) or (*fork*) step.

$$\begin{tikzcd}
s_1 \arrow[rr, "r' : *"] \arrow[d, "r : \mathsf{local}"] & & s_2' \arrow[d, "r : \mathsf{local}"'] \\
s_2 \arrow[r, "r' : *"'] & s_3 \arrow[r, equal] & s_3'
\end{tikzcd}$$

Figure 4.4: Strong local confluence: case (*local*).

(*new*) We distinguish two cases for when the left step allocates a location identifier l. Either the right step also allocates l, or it does not (i.e., it allocates some $l' \neq l$ or is some (*fork*) step).

($new(l)$) The right step allocates l as well. By Lemma 10 and the assumption that the universe of location identifiers is infinite, there exists some $l'' \notin \mathit{LID}\ s_2 \cup \mathit{LID}\ s_2'$. Hence $s_2 \to_{r'} s_3$ and $s_2' \to_r s_3'$ can be chosen to allocate l''. For $\alpha = \mathit{id}$ and $\beta = \mathit{id}(l := l'', l'' := l)$, then, $s_3 \approx_{\alpha\beta} s_3'$ (Figure 4.5, left).

($\overline{new(l)}$) The right step allocates some location identifier $l' \neq l$ or is a (*fork*) step (Figure 4.5, right). In the first case, $l' \notin \mathit{LID}\ s_2$ and $l \notin \mathit{LID}\ s_2'$, so that the steps can commute. In the case of (*fork*), the steps can be shown to commute using Proposition 1.

$$\begin{tikzcd}
s_1 \arrow[rr, "r' : \mathsf{new}(l)"] \arrow[d, "r : \mathsf{new}(l)"] & & s_2' \arrow[d, "r : \mathsf{new}(l'')"'] \\
s_2 \arrow[r, "r' : \mathsf{new}(l'')"'] & s_3 \arrow[r, dash, "\approx" description] & s_3'
\end{tikzcd}
\qquad
\begin{tikzcd}
s_1 \arrow[rr, "r' : \overline{\mathsf{new}(l)}"] \arrow[d, "r : \mathsf{new}(l)"] & & s_2' \arrow[d, "r : \mathsf{new}(l)"'] \\
s_2 \arrow[r, "r' : \overline{\mathsf{new}(l)}"'] & s_3 \arrow[r, equal] & s_3'
\end{tikzcd}$$

Figure 4.5: Strong local confluence: case (*new*).

(*fork*) Finally, we consider the case where the left step is a (*fork*) step. For the right step, it only remains to consider the case where the right step is also a (*fork*) step. Either both steps allocate the same revision identifier r'' (Figure 4.6, left) or not (Figure 4.6, right). The more detailed arguments for these cases are analogous to those in case (*new*) above.

$$\begin{tikzcd}
s_1 \arrow[rr, "r' : \mathrm{fork}(r'')"] \arrow[d, "r : \mathrm{fork}(r'')"] & & s_2' \arrow[d, "r : \mathrm{fork}(r''')"'] \\
s_2 \arrow[r, "r' : \mathrm{fork}(r''')"'] & s_3 \arrow[r, dash, "\approx"] & s_3'
\end{tikzcd}
\qquad
\begin{tikzcd}
s_1 \arrow[rr, "r' : \mathrm{fork}(r''')"] \arrow[d, "r : \mathrm{fork}(r'')"] & & s_2' \arrow[d, "r : \mathrm{fork}(r'')"'] \\
s_2 \arrow[r, "r' : \mathrm{fork}(r''')"'] & s_3 \arrow[r, equal] & s_3'
\end{tikzcd}$$

Figure 4.6: Strong local confluence: case (*fork*), with $r'' \neq r'''$.

□

4.3 Confluence and determinacy

We are now ready to prove confluence (modulo renaming-equivalence) – and consequently determinacy – from strong local confluence. Like Burkchardt and Leijen, our proofs make use of the *diagram tiling* method, in which reduction diagrams are composed to construct new reduction diagrams [BKdV03]. We write out the steps of these proofs (which could be purely visual), so that no preliminary knowledge is required.

We first prove determinacy (Section 4.3.1), and then compare our proof to the one given by Burckhardt and Leijen (Section 4.3.2).

4.3.1 The proof

We first abstract away from the revision identifier indices in Lemma 11.

Lemma 12 (Strong local confluence (abstracted))**.** *If s_1 is reachable and the universes of location and revision identifiers are infinite, then*

$$\begin{tikzcd}
s_1 \arrow[rr] \arrow[d] & & s_2' \arrow[d, "="'] \\
s_2 \arrow[r, "="] & s_3 \arrow[r, dashed, dash, "\approx"] \arrow[u, phantom, "SLC"] & s_3'
\end{tikzcd}$$

Proof. From the assumption $s_2 \leftarrow s_1 \rightarrow s_2'$ and the definition of $\rightarrow$ we know that there exist r, r' such that $s_2 \leftarrow_r s_1 \rightarrow_{r'} s_2'$. By Lemma 11, there exist s_3, s_3' such that $s_2 \rightarrow_{r'}^{=} s_3 \approx s_3' \leftarrow_r^{=} s_2'$. Again using the definition of $\rightarrow$, then, $s_2 \rightarrow^{=} s_3 \approx s_3' \leftarrow^{=} s_2'$. □

Corollary 2 (Strong local confluence with reflexive step)**.** *If s_1 is reachable and the universes of location and revision identifiers are infinite, then*

$$\begin{tikzcd}
s_1 \arrow[rr, "="] \arrow[d] & SLC^= & s_2' \arrow[d, "="'] \\
s_2 \arrow[r, "="] & s_3 \arrow[r, dash, dashed, "\approx"] & s_3'
\end{tikzcd}$$

Proof. By a case distinction on the step $s_1 \to^= s_2'$. If $s_1 \to s_2$, then the result follows directly from Lemma 12. If $s_1 = s_2'$, then the diverging steps can trivially join in s_2. □

The following lemma states that the relations $\to$ and $\approx$ commute. Or stated in a more operational language: that equivalent global states can *mimic* each other's steps.

Lemma 13 (Mimicking)**.**

$$\begin{tikzcd}
s_1 \arrow[r, dash, dashed, "\approx"] \arrow[d] \arrow[dr, phantom, "\mathcal{M}"] & s_1' \arrow[d] \\
s_2 \arrow[r, dash, dashed, "\approx"] & s_2'
\end{tikzcd}$$

Proof. Assume $s_1 \to s_2$ and $s_1 \approx s_1'$. By the definition of $\to$ and $\approx$, there exist some r, α and β such that $s_1 \to_r s_2$ and $\alpha\ (\beta\ s_1) = s_1'$. In particular, then, $s_1'\ (\alpha\ r)$ is a renaming of local state $s_1\ r$. We perform a case distinction on the step $s_1 \to_r s_2$. By inspecting all of the deterministic rules (i.e., every rule except (*new*) and (*fork*)), it is easy to see that $s_1'\ (\alpha\ r)$ can perform the same rule as $s_1\ r$, giving a global state s_2' with $\alpha\ (\beta\ s_2) = s_2'$. For the non-deterministic rules (*new*) and (*fork*), $s_1\ r$ allocates a fresh identifier l or r'', respectively. By the properties of a permutation, $\beta\ l$ and $\alpha\ r''$ are fresh in s_1'. Thus, $s_1'(\alpha\ r)$ can allocate these, again giving a global state s_2' with $\alpha\ (\beta\ s_2) = s_2'$.

□

Corollary 3 (Mimicking generalized)**.**

$$\begin{tikzcd}
s_1 \arrow[r, dash, dashed, "\approx"] \arrow[d, two heads] \arrow[dr, phantom, "\mathcal{M}^*"] & s_1' \arrow[d, two heads] \\
s_2 \arrow[r, dash, dashed, "\approx"] & s_2'
\end{tikzcd}$$

Proof. By induction on the length of the left reduction sequence, using Lemma 13. □

Lemma 14 (Strip lemma)**.** *If s_1 is reachable and the universes of location and revision identifiers are infinite, then*

$$\begin{tikzcd}
s_1 \arrow[r, "="] \arrow[d, two heads] \arrow[dr, phantom, "\mathit{STRIP}"] & s_2' \arrow[d, two heads] \\
s_2 \arrow[r, two heads] & s_3 \approx s_3'
\end{tikzcd}$$

Proof. By induction on the length n of the left reduction sequence $s_1 \to^* s_2$. The base case $n = 0$ is trivial. For the inductive step, assume $s_1 \to^{n+1} s_2$. Thus, there exists some a such that $s_1 \to a \to^n s_2$. By $a \leftarrow s_1 \to^= s_2'$ and Corollary 2, there exists some b and c such that $a \to^= b \approx c \leftarrow s_2'$. By $s_2 \leftarrow^n a \to^= b$ and the induction hypothesis, there exists some s_3 and d such that $s_2 \to^* s_3 \approx d \leftarrow^* b$. By $d \leftarrow^* b \approx c$ and Lemma 3, finally, there exists an s_3' such that $d \approx s_3' \leftarrow^* c$. The joining reduction $s_2 \to^* s_3 \approx d \approx s_3'$ implies $s_2 \to^* s_3 \approx s_3'$ (using transitivity of $\approx$), and the joining reduction $s_2' \to c \to^* s_3'$ implies $s_2' \to^* s_3'$, as required. Pictorially:

$$\begin{tikzcd}
s_1 \arrow[rrr, "="] \arrow[d] & & & s_2' \arrow[d] \\
a \arrow[rr, "="] \arrow[d, "n"'] & \arrow[u, phantom, "\mathrm{SLC}^="] & b \arrow[r, dash, "\approx"] \arrow[d, two heads] & c \arrow[d, two heads] \\
s_2 \arrow[r, two heads] \arrow[ur, phantom, "\mathrm{IH}"] & s_3 \arrow[r, dash, dashed, "\approx"] & d \arrow[r, dash, "\approx"] \arrow[ur, phantom, "\mathcal{M}^*"] & s_3'
\end{tikzcd}$$

$\square$

Lemma 15 (Confluence modulo renaming-equivalence)**.** *If s_1 is reachable and the universes of location and revision identifiers are infinite, then*

$$\begin{tikzcd}
s_1 \arrow[r, dash, dashed, "\approx"] \arrow[d, two heads] & s_1' \arrow[r, two heads] \arrow[d, phantom, "\mathit{CR}^{\approx}"] & s_2' \arrow[d, two heads] \\
s_2 \arrow[r, two heads] & s_3 \arrow[r, dash, dashed, "\approx"] & s_3'
\end{tikzcd}$$

Proof. By induction on the length n of the left reduction sequence $s_1 \to^* s_2$. The base case $n = 0$ is trivial. For the inductive step, assume $s_1 \to^{n+1} s_2$. Thus, there exists an a such that $s_1 \to^n a \to s_2$. By $a \leftarrow^n s_1 \approx s_1' \to^* s_2$ and the induction hypothesis, there exist states b and c such that $a \to^* b \approx c \leftarrow^* s_2'$. By $s_2 \leftarrow a \to^* b$ and Lemma 14, there exist states s_3 and d such that $s_2 \to^* s_3 \approx d \leftarrow^* b$. By $d \leftarrow^* b \approx c$ and Lemma 3, finally, there exists an s_3' such that $d \approx s_3' \leftarrow^* c$. The joining reduction $s_2 \to^* s_3 \approx d \approx s_3'$ implies

$s_2 \rightarrow^* s_3 \approx s_3'$ (using transitivity of $\approx$), and the joining reduction $s_2' \rightarrow^* c \rightarrow^* s_3'$ implies $s_2' \rightarrow^* s_3'$, as required. Pictorially:

$$\begin{tikzcd}
s_1 \arrow[r, dashed, no head, "\approx"] \arrow[d, "n"'] & s_1' \arrow[rr, two heads] \arrow[drr, phantom, "\text{IH}"] & & s_2' \arrow[d, two heads] \\
a \arrow[rr, two heads] \arrow[d] \arrow[dr, phantom, "\text{STRIP}"] & & b \arrow[r, no head, "\approx"] \arrow[d, two heads] \arrow[dr, phantom, "\mathcal{M}^*"] & c \arrow[d, two heads] \\
s_2 \arrow[r, two heads] & s_3 \arrow[r, dashed, no head, "\approx"] & d \arrow[r, no head, "\approx"] & s_3'
\end{tikzcd}$$

□

Theorem 1 (Determinacy). *Let e be a program expression, and assume that the universes of location and revision identifiers are infinite. If $e \downarrow s$ and $e \downarrow s'$, then $s \approx s'$.*

Proof. By $e \downarrow s$, there exists some initialization $s_0 = \epsilon(r \mapsto \langle \epsilon, \epsilon, e \rangle)$ with $s_0 \rightarrow^* s$ maximal. Similarly, by $e \downarrow s'$, there exists some initialization $s_0' = \epsilon(r' \mapsto \langle \epsilon, \epsilon, e \rangle)$ with $s_0' \rightarrow^* s'$ maximal. Since e is a program expression, $\mathit{RID}\ e = \varnothing$, so clearly $s_0 \approx s_0'$, using the renaming $\alpha = (\mathit{id}(r := r'))$. By Lemma 15, there exist reductions $s \rightarrow^* s_3$ and $s' \rightarrow^* s_3'$ with $s_3 \approx s_3'$. By maximality of the reductions $s_0 \rightarrow^* s$ and $s_0' \rightarrow^* s'$ we have $s = s_3$ and $s' = s_3'$, and therefore $s \approx s'$. □

4.3.2 Comparison with the original proof

The mimicking diagram (Lemma 13) does not appear in Burckhardt and Leijen's account. Instead, they derive confluence modulo renaming-equivalence from strong local confluence by the following argument.

First, their strong local confluence lemma is stated as follows:[2]

$$s_2 \leftarrow_r s_1 \approx_{\alpha\beta} s_1' \rightarrow_{r'} s_2' \Longrightarrow \exists s_3\, s_3'.\, s_2 \rightarrow^{=}_{(\alpha^{-1}\ r')} s_3 \approx s_3' \leftarrow^{=}_{(\alpha\ r)} s_2'. \tag{4.1}$$

(Our version of this lemma follows as a corollary by choosing $s_1 = s_1'$ and $\alpha = \beta = \mathit{id}$.)

Next, they lift the relation $\rightarrow$ to equivalence classes $\mathcal{C}$ of states modulo renaming-equivalence, producing the relation $\rightarrow_{\mathcal{C}} \subseteq \mathcal{C} \times \mathcal{C}$. From the implication (4.1), it is then easy to show the strong local confluence lemma

$$C_2 \leftarrow_{\mathcal{C}} C_1 \rightarrow_{\mathcal{C}} C_2' \Longrightarrow \exists C_3.\, C_2 \rightarrow^{=}_{\mathcal{C}} C_3 \leftarrow^{=}_{\mathcal{C}} C_2'$$

[2] Burckhardt and Leijen actually write the strong local confluence lemma as

$$s_2 \leftarrow_r s_1 \approx s_1' \rightarrow_{r'} s_2' \Longrightarrow \exists s_3\, s_3'.\, s_2 \rightarrow^{=}_{r'} s_3 \approx s_3' \leftarrow^{=}_{r} s_2'.$$

This is slightly informal, since, e.g., revision r in s_1' might not in any way relate to revision r in s_1 at all.

for $C_1, C_2 \in \mathcal{C}$: unlifting the hypothesis $C_2 \leftarrow_{\mathcal{C}} C_1 \rightarrow_{\mathcal{C}} C_2'$ produces the hypothesis for (4.1), and the conclusion of (4.1) can be directly lifted to $\exists C_3.\, C_2 \rightarrow^{=}_{\mathcal{C}} C_3 \leftarrow^{=}_{\mathcal{C}} C_2'$. The diagram tiling technique can then be used to show confluence of $\rightarrow_{\mathcal{C}}$:

$$C_2 \leftarrow^*_{\mathcal{C}} C_1 \rightarrow^*_{\mathcal{C}} C_2' \Longrightarrow \exists C_3.\, C_2 \rightarrow^*_{\mathcal{C}} C_3 \leftarrow^*_{\mathcal{C}} C_2' \tag{4.2}$$

using more standard versions of the proofs to Lemmas 14 and 15.

For the final step, Burckhardt and Leijen only remark that the confluence modulo renaming-equivalence property follows from implication (4.2). Our attempt to fill in the details of this step proceeded as follows.

To begin, the confluence modulo renaming-equivalence hypothesis

$$s_2 \leftarrow^* s_1 \approx s_1' \rightarrow^* s_2'$$

can be lifted to $C_2 \leftarrow^*_{\mathcal{C}} C_1 \rightarrow^*_{\mathcal{C}} C_2'$, allowing us to derive $C_2 \rightarrow^*_{\mathcal{C}} C_3 \leftarrow^*_{\mathcal{C}} C_2'$ for some C_3, using (4.2). A subtle complication now arises when we attempt to unlift $C_2 \rightarrow^*_{\mathcal{C}} C_3 \leftarrow^*_{\mathcal{C}} C_2'$ to $s_2 \rightarrow^* s_3 \approx s_3' \leftarrow^* s_2'$ (for some s_3, s_3'): it must be shown that for any reduction sequence $C \rightarrow^*_{\mathcal{C}} C'$, there also exists a corresponding reduction sequence on the level of states $s \rightarrow^* s'$ with $s \in C$ and $s' \in C'$. From the black arrows in Figure 4.7, it is evident that this is not necessarily the case: the sequence $C_1 \rightarrow_{\mathcal{C}} C_2 \rightarrow_{\mathcal{C}} C_3 \rightarrow_{\mathcal{C}} C_4$ unlifts to $s_1 \rightarrow s_2 \neq s_3 \rightarrow s_4 \neq s_5 \rightarrow s_6$. If equivalent states can mimic each other's steps, however, then such a reduction sequence can always be constructed from an arbitrary unlifting (gray arrows): in this case, $s_1 \rightarrow s_2 \rightarrow s_1' \rightarrow s_2'$.

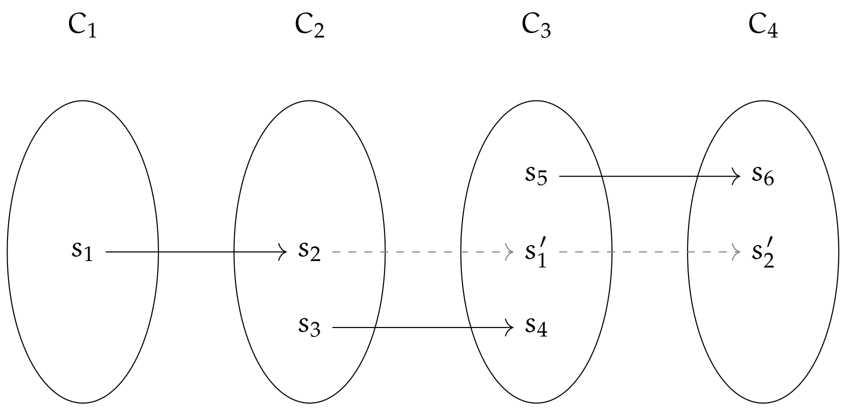

Figure 4.7: Unlifting a reduction sequence $C_1 \rightarrow_{\mathcal{C}} C_2 \rightarrow_{\mathcal{C}} C_3 \rightarrow_{\mathcal{C}} C_4$.

Thus, a proof of the mimicking diagram seems required. Our approach weaves the mimicking diagram directly into the diagram tiling proofs, allowing us to altogether circumvent the concepts of equivalence classes, lifting and unlifting. It also allows us to prove simpler statements of local determinism and strong local confluence, where the root of the divergence is now a single fixed state, rather than an equivalence. The advantage for the mechanical formalization is that we only have to reason about renamings (such as how they distribute over proof terms that represent global states) when they are actually required to establish equivalence.

Chapter 5

Isabelle/HOL

Isabelle/HOL is an *interactive theorem prover* (or *proof assistant*). The user of an interactive theorem prover specifies a formal theory, and formulates theorems. The machine generates the proof obligations for these theorems, which are then solved in an interactive process: the user suggests transformations of the proof state, which the machine processes if they are judged to be logically sound. This cycle continues until all obligations are solved.

'Isabelle' is the name of an interactive theorem prover designed by Paulson in 1986 [Pau89]. Its so-called 'meta-logic' is deliberately minimal, as it is meant to serve as a generic framework for the implementation of other, more expressive 'object logics'. 'HOL', which abbreviates *higher-order logic*, refers to such an object logic [NPW18]. Other available object logics include ZF (Zermelo-Fraenkel set theory) and FOL (first-order logic) [Pau18b]. Despite Isabelle's genericity, we note that the contemporary interest in Isabelle revolves almost exclusively around Isabelle/HOL.

Isabelle's design is a descendant of the so-called 'LCF approach' to theorem proving, formulated by Robert Milner in the 1970s. In this approach, theorems are values of a special data type `thm`, and inference rules are operations defined over `thm`. The combination of a small, trusted inference kernel on the one hand, and strict type-checking on the other, ensures that all values of type `thm` are indeed theorems. LCF's influence on Isabelle is more than conceptual: Milner invented the programming language ML to implement his LCF system, which is also Isabelle's implementation language.

Considered as a platform, the Isabelle/HOL has a rich variety of constructs, libraries and proofs methods, a highly expressive proof language (Isar), a dedicated editor (Isabelle/jEdit), a document preparation system, and more. In the remainder of this chapter, we cover the facets of Isabelle/HOL that are crucial for understanding the formalization described by this thesis.

5.1 Organization

Isabelle encourages a highly structured, hierarchical approach to formalization. Theory development takes place within a *theory* file, whose function can be likened to that of a module or library of a general-purpose programming language. A theory A can *import* another theory B (provided that their signatures are compatible), making the namespace of B available to A. Importing has a recursive effect, giving rise to a tree of dependencies.

A collection of related theories is called a *session*. At the root of each session is the theory `Pure`, which contains Isabelle's meta-logic. The HOL object logic is contained in theory `HOL`, which imports `Pure`. The denomination 'HOL' is also used more broadly to refer to a session which includes the theory `HOL` and associated theories for generic data structures and fundamental mathematical theories.

5.2 The logic

Isabelle's meta-logic is a polymorphic, intuitionistic higher order logic. The HOL object logic is classical and considerably more expressive. In this section we give an overview of both logics, and we explain how they relate. We choose to ignore technical subtleties that would only obfuscate the exposition. Useful technical accounts for respectively the meta-logic and the object logic are written by Paulson [Pau18c, Pau18a].

5.2.1 Isabelle's meta-logic

Isabelle's meta-logic exists for the implementation of object logics. Its main syntactic categories are *types* and *terms*.

Types

Types are defined inductively as follows:

- *Base types*. A base type is represented by some declared constant symbol. A base type is interpreted as a non-empty set. The only base type predefined in Isabelle's meta-logic is *prop*, which is interpreted as the set of meta-level truths.

- *Type variables*. A type variable can be regarded as a placeholder type. Isabelle uses ML-style syntax: an identifier prefixed by an apostrophe (′) (such as `'a`) denotes a type variable. A type variable can be schematic, in which case it is prefixed by `?`. The difference between schematic variables and non-schematic variables is that schematic variables can be instantiated in a proof search. Since free non-schematic variables in lemmas are automatically generalized to schematic variables after a lemma has been proven, it is usually unnecessary to write schematic variables explicitly.

- *Compound types*. Isabelle supports the introduction of compound types through the use of type constructors. Pure contains only one type constructor: *fun*. If σ and τ are types, then (σ, τ) *fun* is a type, which can be more conveniently written as $\sigma \Rightarrow \tau$. A type $\sigma \Rightarrow \tau$ is interpreted as the set of *total* functions from σ to τ. As is customary in functional programming languages, the infix notation $\Rightarrow$ is right-associative. Thus the expression $\sigma_1 \Rightarrow \sigma_2 \Rightarrow \sigma_3$ is parsed as $\sigma_1 \Rightarrow (\sigma_2 \Rightarrow \sigma_3)$.

Isabelle also supports Haskell-style type classes [HW06, Haf18]. Type classes can be likened to interfaces from object-oriented programming languages: they specify constants that any type class instance must implement, and optionally specify assumptions that the implementation must satisfy. Type classes are convenient because they enable operator overloading.

Terms

Every term t has a type σ. A term t can be constrained to have a type σ by writing $t :: \sigma$, which is interpreted as asserting set membership. The Isabelle framework adopts ML's type inference system, and convention dictates that type constraints are only written when type inference fails to infer the intended type.

Terms are constructed as in the simply typed lambda-calculus:

- Any variable x is a term. Like type variables, term variables can be schematic.
- Any constant term c is a term. Isabelle's meta-logic defines three constants, which are axiomatized to behave as meta-level logical connectives.[1] These constants are, for arbitrary types σ:
  - $\Longrightarrow :: prop \Rightarrow prop \Rightarrow prop$ ('implication')
  - $\bigwedge :: (\sigma \Rightarrow prop) \Rightarrow prop$ ('universal quantification')
  - $\equiv :: \sigma \Rightarrow \sigma \Rightarrow prop$ ('equivalence')

  Notice how $\bigwedge$ encodes the universal quantifier: it takes a function with type $\sigma \Rightarrow$ *prop* (intuitively, a 'predicate') and maps it to a *prop* (i.e., a judgement on whether that predicate holds universally). More in line with its semantics, expressions of the form $\bigwedge(\lambda x.\, P)$ are written as $\bigwedge x.\, P$ in Isabelle.
- *Application*. If t is a term of type $\sigma \Rightarrow \tau$ and t' is a term of type σ, then $t\ t'$ is a term of type τ.
- *Abstraction*. If x is variable of type σ and t is a term of type τ, then $\lambda x.\, t$ is a term of type $\sigma \Rightarrow \tau$.

[1] A 1989 paper by Paulson [Pau89] describes a core subset of the axioms.

5.2.2 The HOL object logic

The theory `HOL` introduces the type `bool`, which is the type of object-level truths. Terms of type `bool` are called *formulae*. The object logic introduces and axiomatizes many familiar constants over formulae, such as the connectives of first order logic with equality ($=$, $\neq$, $\neg$, $\vee$, $\wedge$, $\rightarrow$, $\forall$, $\exists$), and some functional programming constructs, such as if-then-else and case expressions. It also introduces a number of logical axioms, such as the law of excluded middle (which makes the logic classical) and the axiom of function extensionality.

The meta-logic and the object logic connect through the hidden function

```
Trueprop :: bool ⇒ prop
```

which lifts object-level truths to meta-level truths. It is used internally to coerce types from `bool` to `prop`, allowing us to reason about formulae on a meta-level. For instance, `p ∧ q ⟹ q ∧ p` is syntactic sugar for `Trueprop (p ∧ q) ⟹ Trueprop (q ∧ p)`.

The HOL session contains many libraries for generic data structures and fundamental mathematical theories, greatly simplifying the formalization effort. Examples of particularly valuable libraries for our formalization are the theories `Set`, `Nat`, `Map`, `Fun` and `Transitive_Closure`, which contain many useful definitions and lemmas for reasoning about respectively sets, natural numbers, partial functions, general function properties and relation closures.

5.3 Definitional mechanisms

Isabelle/HOL offers a variety of definitional mechanisms. A *definitional mechanism* allows the introduction of new types and constants, provided that the definitions satisfy certain soundness-preserving constraints. A general constraint is that the body of a definition should not contain free variables. By contrast, arbitrary axiomatizations (**axiomatization** in Isabelle/HOL) impose no constraints, meaning that one could accidentally introduce logical inconsistency when using them.

For the formalization of this thesis, we were able to safely limit ourselves to the use of definitional mechanisms. In this section we cover the major ones that we have used.

5.3.1 Type synonyms

Type synonyms can be defined through the **type_synonym** command. For instance, the declaration

```
type_synonym natFunction = "nat ⇒ nat"
```

allows us to write `NatFunction` to denote the type `nat ⇒ nat`.

Type synonyms are fully expanded in the internal logic of Isabelle/HOL. Thus, type synonyms are used purely for enhancing readability of theories.

5.3.2 Inductive data types

Inductive data types can be declared using the **datatype** command [BBB+17, BBD+18]. Declarations are allowed to be mutually inductive and argument types can be parameterized. A toy example of a mutually inductive data type with a single type parameter is

```
datatype α redStack = redBox α | extendBlue "α blueStack" α
and α blueStack = blueBox α | extendRed "α redStack" α
```

which defines the types α `redStack` and α `blueStack` for arbitrary types α. Intuitively, a data object with type α `redStack` represents an alternating stack of red and blue boxes (each box containing an element of type α), for which the top box is red.

A **datatype** declaration generates and proves many useful theorems, including distinction laws ($C_1\ x_1\ \ldots\ x_n \neq C_2\ y_1\ \ldots\ y_m$ for distinct constructors C_1 and C_2) and (mutual) induction principles. It also introduces a number of useful constants, such as a function that collects all α occurrences in some α-parameterized data type, and returns it as a set.

5.3.3 Inductive predicates

Inductive predicates and relations are defined using the **inductive** command. An example of a declaration is

```
inductive even :: "nat ⇒ bool" where
  zero: "even 0"
| step: "even n ⟹ even (Suc (Suc n))"
```

which allows us to derive `even n` for all even natural numbers `n`, using the automatically generated induction rule `even.induct`. An **inductive** declaration also implicitly defines negative cases: if we can show for some `n` that `even n` is not derivable from the specified rules, then we may conclude ¬`even n`.

5.3.4 Definitions and abbreviations

Constant definitions can be introduced through the command **definition**. An example is

```
definition double :: "nat ⇒ nat" where
  "double n = 2*n"
```

which adds the constant `double` and the equation `double ?n = 2*?n` (under the name `double_def`) to the internal logic.

The command **abbreviation** is the syntactic sugar analogue of **definition**. As a rule of thumb, **definition** is used to define complex concepts for which one would like to hide the actual definition as an 'implementation detail'. Doing so, one can maintain the right level of abstraction for automation tools, and prevent them from garbling proof states. If one would just like to introduce a name for something simple, however, **abbreviation** is more suitable, since it eliminates the need to make trivial definitional expansions.

One of the constraints on **definition** and **abbreviation** is that they do not allow recursive definitions. The next subsection describes two definitional mechanisms that do allow recursion.

5.3.5 Recursive functions

The command **fun** allows for recursive function definitions. Pattern matching can be used to distinguish cases, making the declaration style reminiscent of that of a functional programming language such as Haskell. An example is the declaration

```
fun fac :: "nat ⇒ nat" where
  "fac 0 = 1"
| "fac n = n * (fac (n - 1))"
```

which defines `fac` as the factorial function.

For soundness reasons, recursive functions must terminate. A **fun** declaration therefore includes a hidden automated attempt at a termination proof. If this attempt fails, then a **fun** declaration

```
fun f :: τ
where
  equations
  ⋮
```

can be expanded to the equivalent **function** declaration

```
function (sequential) f :: τ
where
  equations
  ⋮
by pat_completeness auto
termination by lexicographic_order
```

In this expansion, the command **termination** initiates the termination proof for `f`, which the proof **by** `lexicographic_order` attempts to solve. The proof can be replaced if it is not adequate.

We found that the shorthand **fun** usually suffices. However, in one case we were forced to use a **function** declaration in order to inspect and substitute a failed termination proof.

See Krauss [Kra] for more details on, e.g., the full semantics of **function** and the default termination proof.

5.4 Locales

A locale allows one to abstractly specify constants and assumptions about those constants. For instance, the locale declaration

```
locale fixed_point =
  fixes f :: "'a ⇒ 'a"
  assumes f_has_fixed_point: "∃x. f x = x"
```

fixes a function f that has a fixed point.

Unlike axiomatiziations, the constants and assumptions of locales are not visible in the global context. Rather, the assumptions of a locale are bound to a local context, which must be explicitly opened and closed:

```
context fixed_point
begin
  ⋮
end
```

Any theorems proven between commands **begin** and **end** will be visible to any context that includes the `fixed_point` locale.

Locales can be useful when one wants to reason abstractly about certain constants. In our case, this concerned the lambda calculus substitution operation: Burckhardt and Leijen do not commit to a specific implementation, so neither do we. A benefit is that the locale assumptions make explicit which properties the constant must specify.

Let c^t denote the type of a constant c. A locale declaration l that fixes constants $c_1,\ldots,c_n$ also generates a function $l :: c_1^t \Rightarrow \ldots \Rightarrow c_n^t \Rightarrow$ `bool` that tests whether its arguments satisfy the assumptions of locale l. This function can be used to show that the locale has a *model*, i.e., that one can actually define constants that satisfy the locale's specification. For the locale `fixed_point`, for instance, we could prove `fixed_point id`, where `id` is the identity function. Providing a model for a locale is recommended practice, since this guarantees that the locale describes a class of mathematical objects.

Locales have a much wider relevance than what we have described here. For more details, see Ballarin [Bal03].

5.5 Lemmas and proofs

In this section we explain how to state and prove lemmas.

5.5.1 Lemmas

Lemmas can be stated using the **lemma** command (or its equivalents: **theorem**, **proposition** and **corollary**). An example is

```
lemma positive_product:
  "(n :: nat) > 0 ⟹ (m :: nat) > 0 ⟹ n * m > 0"
```

which states that the product of two positive natural numbers `n` and `m` is positive, and stores the equation under the (optional) label `positive_product`. Instead of writing $A_1 \Longrightarrow A_2 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow C$, we can also group the assumptions of a statement by writing $[\![A_1 ; A_2 ; \ldots ; A_n]\!] \Longrightarrow C$.

The Isar proof language (explained in the subsection below) allows us to take a more structured approach and write

```
lemma positive_product:
  fixes n :: nat and m :: nat
  assumes
    n_positive: "n > 0" and
    m_positive: "m > 0"
  shows "n * m > 0"
```

to express the same lemma. An advantage of this formulation is that we can refer to the assumptions by name in proofs.

5.5.2 Proofs

A proof immediately follows some logical assertion. Proofs can be written in two ways: using apply-style scripts or using Isar proof scripts.

An apply-style script consists of sequence of *tactics*, each of which applies a transformation to the proof state, followed by a **done** command when all subgoals are solved. While the intermediary proof states were visible to the user writing the script, they are not explicit in the proof script itself.

The trivial conjecture $p \wedge q \Longrightarrow q \wedge p$, for instance, can be proven using an apply-style proof script as follows:

```
lemma "p ∧ q ⟹ q ∧ p"
  apply (erule conjE)
  apply (rule conjI)
    apply (assumption)
  apply (assumption)
  done
```

In this simple example, the conjunction elimination rule

$$?P \wedge ?Q \Longrightarrow (?P \Longrightarrow ?Q \Longrightarrow ?R) \Longrightarrow ?R \qquad (\texttt{conjE})$$

is first applied in a forward fashion, transforming the conjecture into the new subgoal

$$p \Longrightarrow q \Longrightarrow q \wedge p.$$

Next, the conjunction introduction rule

$$?P \Longrightarrow ?Q \Longrightarrow ?P \wedge ?Q \qquad (\texttt{conjI})$$

is applied in a backwards fashion, producing a proof state consisting of the two subgoals

$$p \Longrightarrow q \Longrightarrow q$$

and

$$p \Longrightarrow q \Longrightarrow p.$$

The conclusions of these subgoals are among the respective assumptions of these subgoals: two applications of the tactic **apply** (assumption) are used to solve them.

By comparison, an Isar [Wen18a] proof for the same theorem might look as follows:

```
lemma "p ∧ q ⟹ q ∧ p"
proof -
  assume p_and_q: "p ∧ q"
  have p: "p" by (rule conjunct1[OF p_and_q])
  have q: "q" by (rule conjunct2[OF p_and_q])
  show "q ∧ p" by (rule conjI[OF q p])
qed
```

Intermediary results are stated explicitly, and can be named, resulting in a proof script that is much better readable and maintainable, especially for large proofs. The syntax [OF ...] is a *theorem modifier* that is used to instantiate the conditions of the rules that are used.

More generally, a typical *simple* Isar proof script has the format

```
lemma $P_1$
proof $M_1$
  fixes $C_1$ and ...
  assume $P_2$ and ...
  have $P_3$ by $M_2$
  ⋮
  have $P_{n-1}$ by $M_{n-2}$
  show $P_n$ by $M_{n-1}$
qed
```

where the P_i represent formulae, the C_i represent constants, and the M_i represent proof methods. We make three observations:

- The singular argument to the **proof** command is called the *initial proof method*. The initial proof method applies an initial transformation to the conjecture. The method - leaves the proof state as it is. In the example above, the rule (erule conjE) could have been used as an initial proof method, rather than -.

- Out of the commands in the body of the **proof** block, only the command **show** is required, and the proof for the statement P_n must solve the active subgoal.
- Command **qed** closes the proof block, assuming all active subgoals has been solved.
- **proof** blocks can follow any statement, not just lemma statements. Thus, any of the proofs **by** M_i could be replaced with a suitable **proof** block.

Because of Isar's structured approach, Isar proofs are generally preferred over apply-style scripts. But apply-style scripts still have their merits. During proof development, apply-style scripts help quickly explore the proof space. And in finished proofs, we found that apply-style scripts can be useful to formalize trivial arguments over large proof terms that we do not want to repeat textually.

5.5.3 Proof automation

The simple proofs of the previous section relied on explicit rule applications, which is a very specific and low-level approach to theorem proving. Isabelle/HOL has powerful support for proof automation that makes writing proofs much easier. The central methods are *simp*, *auto*, *blast* and *metis* [BBN11]:

- The simplifier *simp* interprets equations `t = t'` (for terms `t` and `t'`) as rewrite rules $t \rightarrow t'$, and performs conditional, contextual rewriting on proof terms (with some additional tricks). Because rules are oriented from left to right, one should always write the 'simpler' term of an equation on the right-hand side.
- The *auto* tool interleaves simplification with a small amount of proof search. This helps clear obstructions for the simplifier, greatly increasing its effectiveness. The *auto* tool also splits up goals into subgoals, which *simp* does not do. Relatedly, *auto* works on *all* subgoals of a proof state, rather than only the first one.
- The *blast* tool is a tableau prover directly written in ML. It is very fast, since it bypasses the Isabelle kernel for proof search. This poses no threat to soundness, however, since any proof it finds is replayed through the Isabelle kernel. The *blast* tool does not perform any form of simplification.
- The *metis* tool is a superposition-based theorem prover. Similar to *blast*, it bypasses the Isabelle kernel for proof search, but replays any proofs it finds in the Isabelle kernel for increased trustworthiness.

metis is very powerful, but relative to the other methods, it is not user-friendly. The reason for this is that *simp*, *auto* and *blast* are all configured to have an implicit knowledge of lemmas in the database, while *metis* knows only about pure logic. Thus, any *metis* method call needs to be supplemented explicitly with relevant lemmas. For this reason, *metis* is typically not invoked by hand. However, *metis* is still very useful, since Sledgehammer (discussed in the next subsection) often returns *metis* proofs.

The set of lemmas that *simp*, *auto* and *blast* implicitly know about can be globally extended by annotating lemmas with *lemma attributes*. These attributes include `simp`, `intro` (introduction), `elim` (elimination), and `dest` (destruction):

- Giving attribute `simp` to a lemma $P_1 \Longrightarrow \ldots \Longrightarrow P_n \Longrightarrow t = t'$ (for $n \geqslant 0$) adds the corresponding conditional rewrite rule to the simplifier.
- Giving attribute `intro` to a lemma of the form $P_0 \Longrightarrow \ldots \Longrightarrow P_m$ (for $m \geqslant 1$) informs classical reasoners (which include *auto* and *blast*) that they may prove a goal of the form P_m by attempting proofs for $P_0, \ldots, P_{m-1}$.
- Giving attribute `elim` to a lemma of the form $P \Longrightarrow (P' \Longrightarrow Q) \Longrightarrow Q$ informs classical reasoners that they may prove a goal `Q` by replacing a hypothesis `P` with `P'` (`P` is *eliminated*).
- Giving attribute `dest` to a lemma of the form $P \Longrightarrow Q$ informs classical reasoners that they may deduce `Q` from `P`.

Attributes can be used locally. For instance,

```
by (auto simp add: X elim: Y)
```

invokes *auto*, with rule `X` added to the simplifier and rule `Y` added as an `elim` rule.

5.5.4 Sledgehammer and Nitpick

Sledgehammer and Nitpick are tools that aide in the proof development process [BBN11].

Sledgehammer aides in the *construction* of proofs. When trying to prove some conjecture, the user can invoke it by writing **sledgehammer**. Based on properties of the conjecture (such as which constants occur in it), Sledgehammer heuristically selects a few hundred facts that may be relevant for proving the conjecture. It then translates the conjecture and these facts into first-order logic, and delegates the result to a host of external resolution and SMT provers. From the output of these provers, an attempt is made to reconstruct an Isabelle proof (which might fail).

By contrast, Nitpick attempts to *disprove* conjectures using model finding techniques, and is invoked by writing **nitpick**. Why attempt to disprove a conjecture? Because it can be easy to make typos or logical errors when stating lemmas or defining concepts, and identifying such problems without tool support can be challenging.

Due to their nature, no **sledgehammer** and **nitpick** commands remain in the finished artifact. We found these tools extremely useful during the proof development process, however. This is especially true for Sledgehammer, which we frequently relied on for connecting the dots of tedious arguments, and for pointing us towards relevant lemmas from the vast amount of library material.

5.6 Isabelle/jEdit

Isabelle/jEdit is a dedicated editor for developing Isabelle theories [Wen12, Wen18b]. Its many useful features include building Isabelle sessions, keeping track of theory dependencies and changes, syntax highlighting, syntactic sugar for mathematical symbols (including subscript and superscript notations), asynchronous proof checking, sledgehammer integration (allowing the user to insert found proofs with a single click) and a search functionality for library theories.

Chapter 6

The formalization

With the exception of the results in Sections 3.1 and 3.2 (which discuss the program expression definition and the (*fork*) side condition, respectively), all of the concepts and results of this thesis have been formalized in Isabelle/HOL (version Isabelle2017). The formalization takes the form of a session composed of seven theory files: `Data` (Section 6.1), `Occurrences` (Section 6.2), `Renaming` (Section 6.3), `Substitution` (Section 6.4), `OperationalSemantics` (Section 6.5), `Executions` (Section 6.6) and `Determinacy` (Section 6.7). The dependencies between these theories are linear: theory `Data` imports `Main`, which is a session that aggregates the most common Isabelle/HOL libraries (see Nipkow [Nip18] for a quick overview), and every other theory imports only the theory that directly precedes it in the given enumeration. The total lines of Isabelle/HOL code is a little over 3000.

This chapter gives an overview of the formalization and documents important design decisions. Its outline mirrors the structure of the formalization. We do not mean to be exhaustive, and we will frequently drop type annotations and proofs.

6.1 Data

Theory `Data` defines some of the general function notations (Section 6.1.1), the elementary data types (values, expressions and contexts) (Section 6.1.2), every result related to plugging and decomposition (Section 6.1.3), and the definitions of stores and states (Section 6.1.4).

6.1.1 Function notations

Most of the general function notations from Section 2.2 already have a representation in Isabelle/HOL. Since functions must be total in Isabelle/HOL, we use the type constructor `map` (defined in theory `Map`) to model partial functions. The map type is based on option types (defined in the HOL theory `Option`), familiar from a programming language like Haskell:

```
datatype 'a option = None | Some 'a
```

A map is a function into some option type:

```
type_synonym ('a, 'b) "map" = "'a ⇒ 'b option" (infixr "⇀" 0)
```

Thus, we will write `'a ⇀ 'b` for a partial function from `'a` to `'b`, and `f x = None` for $f\,x = \bot$. Theory `Map` also defines `dom` and `ran` to denote respectively the domain and range of a partial function. An update of a total function is denoted by `f(x := y)`, and for maps `f`, `Map` introduces the notation `f(x ↦ y)` for `f(x := Some y)`. The function inverse f^{-1} can be denoted by `inv f`, which is defined in the HOL theory `Fun`. For a partial function constraint $f[\![x := y]\!]$ we will introduce no special notation: we will simply add $f\,x = y$ as an assumption on f.

That leaves the notations for the empty function ϵ and the combination $(f{::}g)$. We define these concepts ourselves in the section `FunctionNotations` of the `Data` theory:

```
abbreviation ϵ :: "'a ⇀ 'b" where
  "ϵ ≡ λx. None"

fun combine :: "('a ⇀ 'b) ⇒ ('a ⇀ 'b) ⇒ ('a ⇀ 'b)" ("_;;_" 20) where
  "(f ;; g) x = (if g x = None then f x else g x)"
```

We use the notation `f;;g` for combinations, rather than `f::g`, to avoid clashing with type constraints.

The section also contains an elementary result about combinations that has thus far been assumed:

```
lemma dom_combination_dom_union:
  "dom (τ;;τ') = dom τ ∪ dom τ'"
```

This is the kind of lemma that we will frequently gloss over in this chapter.

6.1.2 Values, expressions and contexts

Section `ValExprCntxt` contains the definitions of the elementary data types of the revision calculus: constants (`const`), values (`('r,'l,'v) val`), expressions (`('r,'l,'v) expr`) and contexts (`('r,'l,'v) cntxt`). The three type parameters `'r`, `'l` and `'v` denote respectively the types of revision identifiers, location identifiers and variables. The name `cntxt` is used since **context** is a reserved keyword.

The definitions are all rather straightforward **datatype** declarations. Values and expressions, for instance, are defined by the following mutually recursive declaration:

```
datatype (RID_V: 'r, LID_V: 'l,'v) val =
  CV const
```

```
| Var 'v
| Loc 'l
| Rid 'r
| Lambda 'v "('r,'l,'v) expr"
and (RID_E: 'r, LID_E: 'l,'v) expr =
  VE "('r,'l,'v) val"
| Apply "('r,'l,'v) expr" "('r,'l,'v) expr"
| Ite "('r,'l,'v) expr" "('r,'l,'v) expr" "('r,'l,'v) expr"
| Ref "('r,'l,'v) expr"
| Read "('r,'l,'v) expr"
| Assign "('r,'l,'v) expr" "('r,'l,'v) expr"
| Rfork "('r,'l,'v) expr"
| Rjoin "('r,'l,'v) expr"
```

The constructor `VE` ('value-expression') coerces a datum of type `('r,'l,'v) val` to type `('r,'l,'v) expr`.

The function $\texttt{RID}_\texttt{V}$ has type `('r,'l,'v) val ⇒ 'r set` and implements the revision identifier collector function *RID* for values *v*, etc. (The `set` type constructor is defined in the HOL theory `Set`.) We have similarly defined $\texttt{RID}_\texttt{C}$ and $\texttt{LID}_\texttt{C}$ for contexts. The **datatype** command also automatically proves all sorts of useful lemmas involving these functions.

6.1.3 Plugging and decomposing

The subsection `PluggingAndDecomposing` contains all definitions and results related to plugging and decomposing. The plug function is defined as follows:

```
fun plug :: "('r,'l,'v) cntxt ⇒ ('r,'l,'v) expr ⇒
  ('r,'l,'v) expr" (infix "◁" 60)
where
  "□ ◁ e = e"
| "ApplyL_ℰ ℰ e1 ◁ e = Apply (ℰ ◁ e) e1"
| "ApplyR_ℰ val ℰ ◁ e = Apply (VE val) (ℰ ◁ e)"
| "Ite_ℰ ℰ e1 e2 ◁ e = Ite (ℰ ◁ e) e1 e2"
| "Ref_ℰ ℰ ◁ e = Ref (ℰ ◁ e)"
| "Read_ℰ ℰ ◁ e = Read (ℰ ◁ e)"
| "AssignL_ℰ ℰ e1 ◁ e = Assign (ℰ ◁ e) e1"
| "AssignR_ℰ l ℰ ◁ e = Assign (VE (Loc l)) (ℰ ◁ e)"
| "Rjoin_ℰ ℰ ◁ e = Rjoin (ℰ ◁ e)"
```

for which we introduce the mixfix notation:

```
translations
  "ℰ[x]" ⇌ "ℰ ◁ x"
```

allowing us to write `ℰ[x]` for `plug ℰ x`. The inductive predicate `redex` defines the redexes, and the inductive predicate `decompose` defines the decomposition rules. The statements

```
inductive_simps redex_simps [simp]: "redex e"
inductive_cases redexE [elim]: "redex e"
inductive_cases decomposeE [elim]: "decompose e ℰ r"
```

automatically generate simplification and elimination rules for these two predicates, where `redex_simps`, `redexE` and `decomposE` denote the names of these rules.

The most important lemma of section `PluggingAndDecomposing` is

```
lemma completion_eq [simp]:
  assumes
    red_e: "redex r" and
    red_e': "redex r'"
  shows "(ℰ[r] = ℰ'[r']) = (ℰ = ℰ' ∧ r = r')"
```

formalizing Lemma 7. The simplifier can rewrite propositions `ℰ[r] = ℰ'[r']` to `ℰ = ℰ' ∧ r = r'` (provided `r` and `r'` are redexes), and `auto` can eliminate this proposition to `ℰ = ℰ'` and `r = r'`.

6.1.4 Stores and states

The closing section `StoresAndStates` defines the types of stores, local states and global states:

```
type_synonym ('r,'l,'v) store = "'l ⇀ ('r,'l,'v) val"
type_synonym ('r,'l,'v) local_state =
  "('r,'l,'v) store × ('r,'l,'v) store × ('r,'l,'v) expr"
type_synonym ('r,'l,'v) global_state = "'r ⇀ ('r,'l,'v) local_state"
```

A type `'a × 'b` is a product type, which is predefined in the HOL theory `Product_Type`. The × operator is right-associative. Thus, the type `local_state` is interpreted as the set containing tuples of the form $(\sigma, (\tau, e))$, with σ and τ stores, and e an expression. Unlike in Figure 2.5, we do not introduce type names for snapshots and local stores. This is because we will always want to prove results about stores generally.

The remainder of the Isabelle section introduces the straightforward definitions `doms ls`, $\mathtt{ls}_\sigma$, $\mathtt{ls}_\tau$ and $\mathtt{ls}_e$ for local states `ls`.

6.2 Occurrences

Theory `Occurrences` defines the *RID* and *LID* notations for all types containing revision and location identifiers (Section 6.2.1), and proves a number of inference rules that help automate reasoning about occurrences (Section 6.2.2).

6.2.1 Definitions

The *RID* and *LID* functions were already generated or values, expression and contexts (Section 6.1.2). The *RID* definitions for the remaining structures (i.e., stores, local states and global states) are as follows:

```
definition RIDS :: "('r,'l,'v) store ⇒ 'r set" where
  "RIDS σ ≡ ⋃ (RIDV ‘ ran σ)"

definition RIDL :: "('r,'l,'v) local_state ⇒ 'r set" where
  "RIDL s ≡ case s of (σ, τ, e) ⇒ RIDS σ ∪ RIDS τ ∪ RIDE e"

definition RIDG :: "('r,'l,'v) global_state ⇒ 'r set" where
  "RIDG s ≡ dom s ∪ ⋃ (RIDL ‘ ran s)"
```

Here, `f ‘ S` denotes `S` under the image of `f`, the same as our definition given in Section 2.2. The *LID* definitions are analogous to the *RID* definitions:

```
definition LIDS :: "('r,'l,'v) store ⇒ 'l set" where
  "LIDS σ ≡ dom σ ∪ ⋃ (LIDV ‘ ran σ)"

definition LIDL :: "('r,'l,'v) local_state ⇒ 'l set" where
  "LIDL s ≡ case s of (σ, τ, e) ⇒ LIDS σ ∪ LIDS τ ∪ LIDE e"

definition LIDG :: "('r,'l,'v) global_state ⇒ 'l set" where
  "LIDG s ≡ ⋃ (LIDL ‘ ran s)"
```

6.2.2 Inference rules

In Chapters 3 and 4, we have reasoned implicitly about occurrences. For instance, given $r'' \in \mathit{RID}\ v$, $r \neq r'$ and

$$s' = s(r \mapsto \langle \sigma, \tau(l \mapsto v), e\rangle, r' \mapsto \langle \sigma', \tau', e'\rangle)$$

it is immediately evident that $r'' \in \mathit{RID}\ s'$.

A more detailed chain of inferences underlying this conclusion would be:

$$\begin{array}{lll} & r'' \in \mathit{RID}\ v & \\ \Longrightarrow & r'' \in \mathit{RID}\ (\tau(l \mapsto v)) & \\ \Longrightarrow & r'' \in \mathit{RID}\ \langle \sigma, \tau(l \mapsto v), e\rangle & \\ \Longrightarrow & r'' \in \mathit{RID}\ (s(r \mapsto \langle \sigma, \tau(l \mapsto v), e\rangle)) & \\ \Longrightarrow & r'' \in s' & \text{(since } r \neq r'\text{)} \end{array}$$

which depends on trivial supporting lemmas such as

$$v \in \mathsf{ran}\ (\tau(l \mapsto v)).$$

To always make this type of argument explicit would be tedious, and it would obfuscate the proofs that we are actually interested in. For this reason we prove a number of lemmas that serve as inference rules for proof automation. Most of these lemmas were not formulated *a priori*: rather, they were introduced as automation got stuck on reasoning about occurrences.

We distinguish two main classes of inference rules. The first class contains pairs of introduction and elimination rules for each of the *RID* and *LID* definitions (subsection `IntroAndElimRules`). The purpose of these is mostly to eliminate the need to reason about some of the higher-order constructs (such as $\bigcup$) up or down the *RID* and *LID* definition hierarchies. For instance:

```
lemma RID_GI [intro]:
  "s r = Some v ⟹ r ∈ RID_G s"
  "s r' = Some ls ⟹ r ∈ RID_L ls ⟹ r ∈ RID_G s"
   apply (simp add: RID_G_def domI)
  by (metis (no_types, lifting) RID_G_def UN_I UnI2 ranI)
```

allows us to henceforth prove r $\in$ *RID* s with `auto` by showing that s maps r to some local state, or by providing some local state L such that r $\in$ *RID* L and s r$'$ = L (for some r$'$).

The second class revolves around proving distribution laws for the *RID* definitions (subsection `Distribution`). Consider for instance the following three lemmas:

```
lemma ID_distr_store [simp]:
  "RID_S (τ(l ↦ v)) = RID_S (τ(l := None)) ∪ RID_V v"

lemma ID_distr_local [simp]:
  "RID_L (σ,τ,e) = RID_S σ ∪ RID_S τ ∪ RID_E e"

lemma ID_distr_global [simp]:
  "RID_G (s(r ↦ ls)) = insert r (RID_G (s(r := None)) ∪ RID_L ls)"
```

Here, a term `f(x := None)` should be interpreted as a restriction of `f` (x is removed from `f`'s domain), and a term `insert x S` is a simplified way of writing `{x} ∪ S`. In each of the three lemmas, the set of revision identifiers of some complex structure is reformulated as the union of the sets of revision identifiers of its components. For instance, invoking `simp` on the proposition

```
r'' ∈ RID_G (s(r ↦ (σ, τ(l ↦ v), e)))
```

simplifies it to

```
r'' = r ∨ r'' ∈ RID_G (s(r := None)) ∨ r'' ∈ RID_S σ ∨ r'' ∈ RID_S (τ(l := None)) ∨
r'' ∈ RID_V v ∨ r'' ∈ RID_E e
```

and `auto` takes it apart even further, transforming disjunctions `p ∨ q` into `(p ⟹ False) ⟹ q`.

What about a term containing multiple updates, such as

```
r″ ∈ RID_G (s(r ↦ ls, r′ ↦ ls′))
```

with `r ≠ r′`? It is simplified to

```
r″ = r′ ∨ r″ ∈ RID_G (s(r := Some ls, r′ := None)) ∨ r″ ∈ RID_L ls′.
```

The desired deconstruction does not take place fully, since the restriction on the outside of the subterm `RID_G (s(r := Some ls, r′ := None))` blocks further applications of the `ID_distr_global` simplification rule. To solve this, we have also added the (clearly terminating) simplification lemma

```
lemma restrictions_inwards [simp]:
  "x ≠ x′ ⟹ f(x := Some y, x′ := None) = (f(x′ := None, x := Some y))"
```

which pushes all restrictions to the left of all proper updates, allowing the simplification rule `ID_distr_global` to fully take apart sequences of updates.

The closing subsection `Misc` contains a couple of miscellaneous lemmas related to occurrences. For instance, the lemma

```
lemma ID_distr_global_conditional:
  "s r = Some ls ⟹ RID_G s = insert r (RID_G (s(r := None)) ∪ RID_G ls)"
  "s r = Some ls ⟹ LID_G s = LID_G (s(r := None)) ∪ LID_L ls"
```

is useful in situations where a global state is not stated in some update form `f(x ↦ y)`, while we do have some knowledge about what it maps to (in the form of a condition). It also contains a number of inference rules related to combinations, for which we have no simplification laws.

6.3 Renaming

Theory `Renaming` introduces the renaming definitions (Section 6.3.1) and formalizes the notion of renaming-equivalence (Section 6.3.2). It also proves distributive laws (Section 6.3.3) and lemmas about a special class of permutations that we call *swaps* (Section 6.3.4), both of which aid in automation.

6.3.1 Definitions

For any parameterized type

$$(\alpha_1,\ \dots,\ \alpha_n)\ \kappa$$

introduced through the **datatype** command, a function

$$\texttt{map\_}\kappa :: (\alpha_1 \Rightarrow \alpha_1') \Rightarrow \dots \Rightarrow (\alpha_n \Rightarrow \alpha_n') \Rightarrow (\alpha_1,\ \dots,\ \alpha_n)\ \kappa \Rightarrow (\alpha_1',\ \dots,\ \alpha_n')\ \kappa$$

is generated. The term `map_κ f₁ ... fₙ x` denotes the term x with every element y at an α_i position in x replaced by $f_i\ y$. We reuse this function to implement $\alpha\beta$-renaming for the three elementary data types:

```
abbreviation rename_val ::
  "('r ⇒ 'r) ⇒ ('l ⇒ 'l) ⇒ ('r,'l,'v) val ⇒ ('r,'l,'v) val" ("ℛV")
where "ℛV α β v ≡ map_val α β id v"

abbreviation rename_expr ::
  "('r ⇒ 'r) ⇒ ('l ⇒ 'l) ⇒ ('r,'l,'v) expr ⇒ ('r,'l,'v) expr" ("ℛE")
where "ℛE α β e ≡ map_expr α β id e"

abbreviation rename_cntxt ::
  "('r ⇒ 'r) ⇒ ('l ⇒ 'l) ⇒ ('r,'l,'v) cntxt ⇒ ('r,'l,'v) cntxt" ("ℛC")
where "ℛC α β ℰ ≡ map_cntxt α β id ℰ"
```

Let σ' be an $\alpha\beta$-renaming of some store σ. How should σ' behave? If $\sigma\ l = \bot$, then we should have $\sigma'\ (\beta\ l) = \bot$, and if $\sigma\ l = v$, then we should have $\sigma'\ (\beta\ l) = v'$, with v' the $\alpha\beta$-renaming of v. This requirement is captured by the following definition:

```
definition is_store_renaming where
  "is_store_renaming α β σ σ' ≡ ∀l. case σ l of
  None ⇒ σ' (β l) = None | Some v ⇒ σ' (β l) = Some (ℛV α β v)"
```

We do not use this relational definition directly to capture renamings, however, since we found that it led to tedious proofs. We use the following equational definition instead, relying on the monadic bind operator »=, familiar from a programming language like Haskell:[1]

```
notation Option.bind (infixl "»=" 80)

fun ℛS :: "('r ⇒ 'r) ⇒ ('l ⇒ 'l) ⇒ ('r,'l,'v) store ⇒ ('r,'l,'v) store"
where "ℛS α β σ l = σ (inv β l) »= (λv. Some (ℛV α β v))"
```

The definition assumes that the permutation β has an inverse. Since we assume that β is a permutation, it is bijective (Section 2.2.3), and therefore has an inverse in all of our use cases. Where needed, the bijectivity assumption needs to be made explicit throughout the formalization, such as in the following lemma:

```
lemma ℛS_implements_renaming:  "bij β ⟹ is_store_renaming α β σ (ℛS α β σ)"
```

which is meant to convince us that the definition $\mathcal{R}_S$ is sound. The predicate `bij` is defined in theory `Fun`.

The renaming of a local state is obtained by renaming each of its components:

```
fun ℛL ::
  "('r ⇒ 'r) ⇒ ('l ⇒ 'l) ⇒ ('r,'l,'v) local_state ⇒ ('r,'l,'v) local_state"
where "ℛL α β (σ,τ,e) = (ℛS α β σ, ℛS α β τ, ℛE α β e)"
```

[1]The bind operator »= satisfies the monad laws `(None »= f) = f` and `(Some v »= f) = f v`.

and the renaming of a global state, finally, is analogous to the renaming of a store:

```
fun ℛG ::
  "('r ⇒ 'r) ⇒ ('l ⇒ 'l) ⇒ ('r,'l,'v) global_state ⇒ ('r,'l,'v) global_state"
where "ℛG α β s r = s (inv α r) »= (λls. Some (ℛL α β ls))"
```

We also experimented with similar renaming definitions where the α and β were decoupled. However, we found that this did not really simplify proofs, while it did lead to a lot of lemma duplicates. For this reason we simply choose to write, e.g., `ℛG α id s` when we only wish to apply a renaming α to the revision identifiers in some global state `s`.

6.3.2 Renaming-equivalence

Section `RenamingEquivalence` defines the notion of renaming-equivalence $\approx$:

```
definition eq_states :: ("_ ≈ _" [100, 100]) where
  "s ≈ s' ≡ ∃α β. bij α ∧ bij β ∧ ℛG α β s = s'"
```

Several identity, composition and inverse laws for renamings are proven to establish that $\approx$ is in fact an equivalence, culminating with the lemmas:

```
lemma αβ_refl: "s ≈ s"
lemma αβ_trans: "s ≈ s' ⟹ s' ≈ s'' ⟹ s ≈ s''"
lemma αβ_sym: "s ≈ s' ⟹ s' ≈ s"
```

The proof of equivalence relies on the following facts (for bijective α and β):

- $id\ (id\ s) = s$ for proving reflexivity.
- $\alpha\ (\beta\ s) = s' \Longrightarrow \alpha'\ (\beta'\ s') = s'' \Longrightarrow (\alpha' \circ \alpha)\ ((\beta' \circ \beta)\ s) = s''$ for proving transitivity, where $f \circ g$ denotes the composition of f and g ('f after g').
- $\alpha\ (\beta\ s) = s' \Longrightarrow \alpha^{-1}\ (\beta^{-1}\ s') = s$ for proving symmetry.

6.3.3 Distributive laws

We usually want to push renamings all the way down to the variables of a term. For instance

```
ℛG α β (s(r ↦ (σ(l ↦ v), τ;;τ', ℰ[e])))
```

should simplify to

```
ℛG α β s(α r ↦ (ℛS α β σ(β l ↦ ℛV α β v), ℛS α β τ;;ℛS α β τ',
ℛC α β ℰ[ℛE α β e]))
```

This allows renamed global states to match the source states of the rules of the operational semantics, and it helps in proving two global states are renaming-equivalent (see the next subsection).

We have proven distributive simplification laws for completions (`renaming_distr_completion`), combinations (`renaming_distr_combination`), store updates (`renaming_distr_store`) and global updates (`renaming_distr_global`): the example above illustrates their necessity. The definition for the renaming of a local state is itself a distributive simplification law: we need not prove a separate one.

6.3.4 Swaps

We call a permutation of the form $id(x := x', x' := x)$ a *swap*. It is easy to see that swaps are bijective (lemma `swap_bij` proves it).

Let α be revision identifier swap $id(r := r', r' := r)$ and x some structure containing revision identifiers. If $r \notin RID\ x$ and $r' \notin RID\ x$, then clearly, $\alpha\ x = x$. Section `Swaps` proves such laws for both location and revision identifier permutations, and for all structures containing identifiers. For instance, for revision identifier permutations applied to stores we have the rule

```
lemma eliminate_swap_store_rid [simp, intro]:
  "r ∉ RIDS σ ⟹ r' ∉ RIDS σ ⟹ ℛS (id(r := r', r' := r)) id σ = σ"
```

We illustrate the purpose of these laws by reconsidering the (*new*) case for the proof of local determinism (Lemma 9). In this case, the source state s_1 with

$$s_1\ r = \langle \sigma, \tau, \mathcal{E}[\mathsf{ref}\ v] \rangle$$

has two target states:

$$s_2 = s_1(r \mapsto \langle \sigma, \tau(l \mapsto v), \mathcal{E}[l] \rangle)$$

and

$$s_2' = s_1(r \mapsto \langle \sigma, \tau(l' \mapsto v), \mathcal{E}[l'] \rangle)$$

with $l \notin LID\ s_1$ and $l' \notin LID\ s_1$. The goal is to show $s_2 \approx s_2'$. Our claim was that for the revision identifier permutation $\alpha = id$ and the location identifier permutation $\beta = id(l := l', l' := l)$, $\alpha\ (\beta\ s_2) = s_2'$.

This can be automatically derived using `auto` as follows. First, from the simplifying distributive laws for renamings `auto` derives that

$$\alpha\ (\beta\ s_2)) = \alpha\ (\beta\ s_1)(\alpha\ r \mapsto \langle \alpha\ (\beta\ \sigma), \alpha\ (\beta\ \tau)(\beta\ l \mapsto \alpha\ (\beta\ v)), \alpha\ (\beta\ \mathcal{E})[\beta\ l] \rangle).$$

Second, from $l \notin LID\ s_1$, $l' \notin LID\ s_1$ and $s_1\ r = \langle \sigma, \tau, \mathcal{E}[\mathsf{ref}\ v] \rangle$, `auto` derives that l and l' do not occur in σ, τ, $\mathcal{E}$ and v, using the simplifying distributive laws for identifiers (together with lemma `ID_distr_global_conditional`). Finally, using the swap laws and the definitions of α and β,

$$\alpha\ (\beta\ s_1)(\alpha\ r \mapsto \langle \alpha\ (\beta\ \sigma), \alpha\ (\beta\ \tau)(\beta\ l \mapsto \alpha\ (\beta\ v)), \alpha\ (\beta\ \mathcal{E})[\beta\ l]\rangle)$$

can be further simplified to

$$s_1(r \mapsto \langle \sigma, \tau(l' \mapsto v), \mathcal{E}[l']\rangle)$$

which is the definition of s_2', completing the proof.

From a logical perspective, it would be nicer to generalize the swap laws (e.g., $\alpha\ x = x$ if $\forall r \in \mathit{RID}\ x.\ \alpha\ r = r$). However, the current formulation is sufficient for our purposes, and works better with pattern matching.

6.4 Substitution

The (*apply*) rule of the operational semantics assumes the existence of a substitution function over expressions. Theory `Substitution` introduces the constant `subst` (representing substitution) by means of a locale (Section 6.4.1). It also defines two models for the locale, demonstrating that its assumptions are satisfiable (Section 6.4.2).

6.4.1 Locale `substitution`

Our motivation for using a locale to introduce the `subst` function is covered by Section 5.4. The locale is defined as follows:

```
locale substitution =
  fixes subst :: "('r,'l,'v) expr ⇒ 'v ⇒ ('r,'l,'v) expr
    ⇒ ('r,'l,'v) expr"
  assumes
    renaming_distr_subst:
      "ℛ_E α β (subst e x e') = subst (ℛ_E α β e) x (ℛ_E α β e')"
  and
    subst_introduces_no_rids:
      "RID_E (subst e x e') ⊆ RID_E e ∪ RID_E e'"
  and
    subst_introduces_no_lids:
      "LID_E (subst e x e') ⊆ LID_E e ∪ LID_E e'"
```

The three locale assumptions were not specified a priori: rather, they were added whenever they were required in the formalization process. The first assumption is required to prove that steps can be mimicked by renaming-equivalent states. The relevance of the second assumption is more easily evident: we need it to show that an (*apply*) step $s_1 \to_r s_2$ and a (*fork*) step $s_1 \to_{r'} s_2'$ (forking some r'') commute (i.e., $\exists s_3.\ s_2 \to_{r'} s_3 \leftarrow_r s_2'$), since we will have to prove that r'' is still fresh in s_2. The need for the third assumption is analogous to the second.

6.4.2 Models for `substitution`

As explained in Section 5.4, it is good practice to demonstrate that the assumptions of locale `substitution` are satisfiable. The remainder of theory `Substitution` provides two models.

The first model is a function `constant_function` that always maps to unit:

```
fun constant_function where
  "constant_function e x e' = VE (CV Unit)"
```

It is very easy to show that it is a model:

```
lemma constant_function_models_substitution:
  "substitution constant_function"
  by (auto simp add: substitution_def)
```

This model is of some interest, since it demonstrates that the substitution function plays no interesting role in the proof of determinacy.

The second model is a more faithful implementation of a deterministic substitution function. We give a substitution definition for the pure lambda calculus that best illustrates the approach. Let $\mathcal{V}(t)$ denote the set of variables (free and bound) occurring in some lambda-term t, and let $t_{x\mapsto y}$ denote the term t in which every variable occurrence x has been renamed to y. Moreover, assume that natural numbers are used for variables. The substitution definition $[t/x]t'$ ('t for x in t'') is defined as follows:

$$\begin{array}{rcll} [t/x]x & = & t & \\ [t/x]y & = & y & (\textit{if } x \neq y) \\ [t/x](t'\ t'') & = & ([t/x]t')\ ([t/x]t'') & \\ [t/x](\lambda x.\, t') & = & \lambda x.\, t' & \\ [t/x](\lambda y.\, t') & = & \lambda z.\, [t/x](t'_{y\mapsto z}) & (\textit{if } x \neq y \textit{ and } z = \textit{max}(\mathcal{V}(t), \mathcal{V}(t')) + 1) \end{array}$$

The last case ensures capture-avoidance, since z's definition implies that it cannot occur free in t.

The approach to renaming is quite aggressive. First, if y is not free in t, then the renaming is not required, but for our purposes there is no point in adding an extra case distinction. Second, it would suffice to rename only *free* occurrences of y to z in t', rather than all occurrences. We nonetheless chose to rename all occurrences, since it allows us to reuse the generated function `map_expr` (described in Section 6.3.1) for renamings: the abbreviation

```
abbreviation rename_vars_expr ("ℛ𝒱_E") where
  "ℛ𝒱_E ζ ≡ map_expr id id ζ"
```

allows us to write `ℛ𝒱_E (id(x := y)) t` for $t_{x\mapsto y}$. By contrast, we found that renaming free variables using the substitution notion itself complicated proofs.

The function `nat_subst`$_E$ implements the analogous substitution definition for revision calculus expressions. It is defined through mutual recursion:

```
function
  nat_subst_V and
  nat_subst_E
  where
  ⋮
| "nat_subst_V e x (Lambda y e') = VE (
  if x = y then
    Lambda y e'
  else
    let z = Suc (Max (𝒱_E e' ∪ 𝒱_E e)) in
    Lambda z (nat_subst_E e x (ℛ𝒱_E (id(y := z)) e')))"
  ⋮
```

The command **function** is used since the termination proof attempted by **fun** fails (Section 5.3.5). The problem lies in the case included in the snippet above: the *size* of the third argument (the number of *Val* and *Expr* constructors) is strictly decreasing in the recursive function call, but this it is not automatically inferred. To remedy this, we prove the general lemma:

```
lemma var_renaming_preserves_size:
    "size (map_val α β ζ v) = size v"
    "size (map_expr α β ζ e) = size e"
```

which states that renamings do not change the size of a term.

Two mutually recursive functions are internally represented as a single sum type function. The termination proof

```
termination
apply (relation "measure (λx. case x of Inl (e,x,v) ⇒ size v |
  Inr (e,x,e') ⇒ size e')")
by (auto simp add: var_renaming_preserves_size(2))
```

states that the recursive calls are decreasing in the third argument of whichever option of the sum type is defined. It then automatically solves the subsequent goal by `auto`, strengthened with `size (map_expr α β ζ e) = size e` as a `simp` law.

The remainder of the `Substitution` theory proves that `nat_subst`$_E$ is a model, culminating in the lemma

```
lemma nat_subst_E_models_substitution: "substitution nat_subst_E"
```

6.5 Operational semantics

The operational semantics of the revision calculus is defined by the following inductive predicate contained in theory `OperationalSemantics`:

```
inductive revision_step ::
  "'r ⇒ ('r,'l,'v) global_state ⇒ ('r,'l,'v) global_state ⇒ bool"
where
  app: "s r = Some (σ, τ, ℰ[Apply (VE (Lambda x e)) (VE v)]) ⟹
    revision_step r s (s(r ↦ (σ, τ, ℰ[subst e x (VE v)])))"
| ifTrue: "s r = Some (σ, τ, ℰ[Ite (VE (CV T)) e1 e2]) ⟹
    revision_step r s (s(r ↦ (σ, τ, ℰ[e1])))"
| ifFalse: "s r = Some (σ, τ, ℰ[Ite (VE (CV F)) e1 e2]) ⟹
    revision_step r s (s(r ↦ (σ, τ, ℰ[e2])))"
| alloc: "s r = Some (σ, τ, ℰ[Ref (VE v)]) ⟹ l ∉ LID_G s ⟹
    revision_step r s (s(r ↦ (σ, τ(l ↦ v), ℰ[VE (Loc l)])))"
| get: "s r = Some (σ, τ, ℰ[Read (VE (Loc l))]) ⟹ l ∈ dom (σ;;τ) ⟹
    revision_step r s (s(r ↦ (σ, τ, ℰ[VE (the ((σ;;τ) l))])))"
| set: "s r = Some (σ, τ, ℰ[Assign (VE (Loc l)) (VE v)]) ⟹
    l ∈ dom (σ;;τ) ⟹ revision_step r s (s(r ↦ (σ, τ(l ↦ v), ℰ[VE (CV Unit)])))"
| fork: "s r = Some (σ, τ, ℰ[Rfork e]) ⟹ r' ∉ RID_G s ⟹
    revision_step r s (s(r ↦ (σ, τ, ℰ[VE (Rid r')]), r' ↦ (σ;;τ, ε, e)))"
| join: "s r = Some (σ, τ, ℰ[Rjoin (VE (Rid r'))]) ⟹
    s r' = Some (σ', τ', VE v) ⟹
    revision_step r s (s(r := Some (σ, (τ;;τ'), ℰ[VE (CV Unit)]), r' := None))"
| join_ε: "s r = Some (σ, τ, ℰ[Rjoin (VE (Rid r'))]) ⟹ s r' = None ⟹
    revision_step r s ε"
```

The formulations of the side conditions on (*new*) (i.e., `l ∉ LID_G s`) and (*fork*) (i.e., `r' ∉ RID_G s`) are as strict as possible, and the side conditions on (*get*) and (set) are explicit.

The declaration is followed by

```
inductive_cases revision_stepE [elim, consumes 1, case_names app ifTrue ifFalse
alloc get set fork join join_ε]:  "revision_step r s s'"
```

which generates an elimination rule of the form

```
⟦ revision_step ?r ?s ?s' ;
(⋀σ τ ℰ x e v. ?s' = ?s(?r ↦ (σ, τ, ℰ [subst (VE v) x e])) ⟹
?s ?r = Some (σ, τ, ℰ [Apply (VE (Lambda x e)) (VE v)]) ⟹ ?P) ;
... ;
(⋀σ τ ℰ r'. ?s' = ε ⟹ ?s ?r = Some (σ, τ, ℰ [Rjoin (VE (Rid r'))]) ⟹
?s r' = None ⟹ ?P) ⟧
⟹ ?P
```

where the '…' denotes the seven intermediate cases of the operational semantics. The command `consumes 1` in the **inductive_cases** declaration signifies that the first premiss `revision_step ?r ?s ?s'` is the target of elimination, and `case_names` provides names to all of the following cases. This allows one to write the readable Isar case analyses of revision steps that are used throughout the formalization. The readability could be further enhanced by also giving names to the case hypotheses and conclusions.

The remainder of `OperationalSemantics` is largely dedicated to the proof of Lemma 1 (including introducing any additional required notions for it). The lemma name is `step_preserves_`$\mathcal{S}_G$`_and_`$\mathcal{A}_G$. The proof is faithful to the proof given in Section 3. Transitive chains of reasoning, such as

$$\begin{aligned} e_1 &\subseteq e_2 \\ &\subseteq e_3 \\ &\vdots \\ &\subseteq e_n \end{aligned}$$

for proving $e_1 \subseteq e_n$, are represented in Isar by writing

```
have "e₁ ⊆ e₂"
also have "... ⊆ e₃"
⋮
also have "... ⊆ eₙ"
finally have "e₁ ⊆ eₙ"
```

and are used throughout the proof (here, '...' is part of the Isar syntax).

The theory closes with the inductive predicate `revision_step_relaxed`. It is the same predicate as `revision_step`, except that the (*new*) side condition is now `l ∉ ⋃ { doms ls | ls. ls ∈ ran s }`, and the side conditions on (*get*) and (*set*) are removed.

We cannot yet show that `revision_step` and `revision_step_relaxed` define the same transition system: for that we first need to formalize notions related to executions, which is the subject of the next section.

6.6 Executions

Theory `Executions` introduces the required concepts for reasoning about executions. To avoid overloading the symbol for logical implication $\rightarrow$, the set of steps is defined as a relation `[⇝]`:

```
definition steps :: "('r,'l,'v) global_state rel" ("[⇝]") where
  "steps = { (s,s') | s s'. ∃r. revision_step r s s' }"
```

where the type `'a rel` is a type synonym for `('a × 'a) set`. We also introduce an infix notation $s \leadsto s'$ for $(s, s') \in$ `[⇝]`, and infix notations $\leadsto^*$ and $\leadsto^=$ for respectively the reflexive transitive closure and reflexive closure of $\leadsto$. The closure operations are defined using definitions from the HOL theory `Transitive_Closure`.

The notions from Section 2.2.2 are faithfully defined:

```
abbreviation program_expr where
  "program_expr e ≡ LID_E e = {} ∧ RID_E e = {}"
```

```
abbreviation initializes where
  "initializes s e ≡ ∃r. s = (ε(r ↦ (ε,ε,e))) ∧ program_expr e"

abbreviation initial_state where
  "initial_state s ≡ ∃e. initializes s e"

definition execution where
  "execution e s s' ≡ initializes s e ∧ s ⇝* s'"

definition maximal_execution where
  "maximal_execution e s s' ≡ execution e s s' ∧ (∄s''. s' ⇝ s'')"

definition terminates_in where
  "e ↓ s' ≡ ∃s. maximal_execution e s s'"

definition reachable where
  "reachable s ≡ ∃e s'. execution e s' s"
```

We also define the notion of an inductive invariant:

```
definition inductive_invariant :: "(('r,'l,'v) global_state ⇒ bool) ⇒ bool"
where "inductive_invariant P ≡ (∀s. initial_state s ⟹ P s) ∧
  (∀s s'. s ⇝ s' ⟹ P s ⟹ P s')"
```

for which we prove:

```
lemma inductive_invariant_is_execution_invariant:
  "reachable s ⟹ inductive_invariant P ⟹ P s"
```

Inductive invariance is used to establish two facts. First, it is used to prove that the two revisions step definitions `revision_step` and `revision_step_relaxed` define the same transition relation on reachable states (Corollary 2):

```
lemma transition_relations_equivalent:
  "reachable s ⟹ revision_step r s s' = revision_step_relaxed r s s'"
```

We use the `revision_step` predicate as the default in the remainder of the formalization, since it makes more information explicit. Second, it is used to establish that reachable states contain finitely many revision and location identifiers (Lemma 10):

```
lemma reachable_imp_identifiers_finite:
  assumes reach: "reachable s"
  shows
    "finite (RID_G s)"
    "finite (LID_G s)"
```

This lemma is used to show that revisions can always allocate some revision or location identifier:

```
lemma reachable_imp_identifiers_available:
  assumes
    "reachable (s :: ('r,'l,'v) global_state)"
  shows
    "infinite (UNIV :: 'r set) ⟹ ∃r. r ∉ RID_G s"
    "infinite (UNIV :: 'l set) ⟹ ∃l. l ∉ LID_G s"
```

The predicates `finite` and `infinite` are defined in `Finite_Set`, and the set `UNIV` denotes the universal set of some type.

Finally, we show that reachability is closed under execution:

```
lemma reachability_closed_under_execution:
  "reachable s ⟹ s ⇝* s' ⟹ reachable s'"
```

In proofs, this allows us to extend a reachability assumption on s to any of its successor states s'.

6.7 Determinacy

The last theory of our formalization is theory `Determinacy`. This theory contains the proofs for rule determinism, local determinacy, strong local confluence, confluence modulo renaming-equivalence and, finally, determinacy.

6.7.1 Rule determinism

Rule determinism is represented by nine lemmas: one for each rule of the operational semantics. The rule for (*apply*), for instance, is

```
lemma app_deterministic [simp]:
  assumes
    s_r: "s r = Some (σ, τ, ℰ [Apply (VE (Lambda x e)) (VE v)])"
  shows
    "(revision_step r s s') =
    (s' = (s(r ↦ (σ, τ, ℰ [subst (VE v) x e]))))"
```

The rule determinism lemmas are all stated as simplification laws. These laws are useful when reasoning about peaks $s_2 \leftarrow_r s_1 \rightarrow_{r'} s_2'$. Performing a case distinction on, e.g., the left step $s_2 \leftarrow_r s_1$ generates some hypothesis $s_1\ r = L$. Any deducible information about the target state s_2' of the right step $s_1 \rightarrow_{r'} s_2'$ is then immediately derived: we need not first perform a case distinction on the right step, and then eliminate all the nonsensical cases.

The information also includes the required information on side conditions, if applicable:

```
lemma new_pseudodeterministic [simp]:
  assumes
    s_r: "s r = Some (σ, τ, ℰ [Ref (VE v)])"
  shows
    "(revision_step r s s') = (∃l. l ∉ LID_G s ∧ s' =
    (s(r ↦ (σ, τ(l ↦ v), ℰ [VE (Loc l)]))))"
```

6.7.2 Strong local confluence

Strong local confluence is represented by the following theorem:

```
theorem strong_local_confluence:
  assumes
    l: "revision_step r s1 s2" and
    r: "revision_step r' s1 s2'" and
    reach: "reachable (s1 :: ('r,'l,'v) global_state)" and
    lid_inf: "infinite (UNIV :: 'l set)" and
    rid_inf: "infinite (UNIV :: 'r set)"
  shows
    "∃s3 s3'. s3 ≈ s3' ∧ (revision_step r' s2 s3 ∨ s2 = s3) ∧
    (revision_step r s2' s3' ∨ s2' = s3')"
proof (cases "r = r'")
  case True
  thus ?thesis by (metis l local_determinism r)
next
  case neq: False
  thus ?thesis by (cases rule: revision_stepE[OF l]) (auto simp add:
    assms SLC_app SLC_ifTrue SLC_ifFalse SLC_new SLC_get SLC_set
    SLC_fork SLC_join SLC_join_ε)
qed
```

Like in the paper proof, a case distinction is made on $r = r'$, where the $r = r'$ case is the local determinism lemma:

```
lemma local_determinism:
  assumes
    left: "revision_step r s1 s2" and
    right: "revision_step r s1 s2'"
  shows "s2 ≈ s2'"
```

The automation aspects of the proof to lemma `local_determinism` are explained in Section 6.3.4, and also help in understanding the other Isabelle proofs for strong local confluence.

For the $r \neq r'$ case, each case of the case distinction on the left step $s_1 \to_r s_2$ is represented by a dedicated lemma `SLC_x`, with `x` the name of the case. This was done to make the large proof more manageable and more readable. The order of these lemmas is the same as in the proof to Lemma 11. The proofs contain more explicit tedious detail regarding, e.g., freshness of identifiers and swaps. This is especially true in case `SLC_fork`.

For proving each of the nine individual cases, we prove two general principles `SLC_sym` and `SLC_commute`:

```
lemma SLC_sym:
  "∃s3' s3. s3' ≈ s3 ∧ (revision_step r' s2 s3' ∨ s2 = s3') ∧
  (revision_step r s2' s3 ∨ s2' = s3) ⟹
  ∃s3 s3'. s3 ≈ s3' ∧ (revision_step r s2' s3 ∨ s2' = s3) ∧
  (revision_step r' s2 s3' ∨ s2 = s3')"

lemma SLC_commute:
  "⟦ s3 = s3'; revision_step r' s2 s3; revision_step r s2' s3' ⟧ ⟹
  s3 ≈ s3' ∧ (revision_step r' s2 s3 ∨ s2 = s3) ∧
  (revision_step r s2' s3' ∨ s2' = s3')"
```

Lemma `SLC_sym` is used for solving the symmetric cases that were omitted in the proof to Lemma 11. When applied in a case (*rule1*) vs. (*rule2*), it transforms the conclusion into its symmetric version, effectively reducing the case to (*rule2*) vs. (*rule1*), which at that point already has a proof.

Lemma `SLC_commute` is a general proof principle that is used in many cases in which the diverging steps commute. By applying the rule, the proof obligation is made more specific, which helps both in guiding proof automation and in writing understandable Isar proofs. The lemmas `join_and_local_commute`, `local_steps_commute` and `local_and_rfork_commute` have similar roles, refining the proof obligation even further for the commuting pairs (*join*) vs. (*local*), (*local*) vs. (*local*) and (*local*) vs. (*fork*), respectively.

6.7.3 Confluence and determinacy

The remainder of theory `Determinacy` revolves around proving the diagram tiling proofs of Section 4.3, culminating in the proof of confluence modulo renaming-equivalence:

```
lemma confluence_modulo_equivalence:
  assumes
    s1s2: "s1 ↝* s2" and
    s1s2': "s1' ↝* s2'" and
    equiv: "s1 ≈ s1'" and
    reach: "reachable (s1 :: ('r,'l,'v) global_state)" and
    lid_inf: "infinite (UNIV :: 'l set)" and
    rid_inf: "infinite (UNIV :: 'r set)"
  shows "∃s3 s3'. s3 ≈ s3' ∧ s2 ↝* s3 ∧ s2' ↝* s3'"
```

and finally, determinacy:

```
theorem determinacy:
 assumes
  prog_expr: "program_expr e" and
  e_terminates_in_s: "e ↓ s" and
  e_terminates_in_s': "e ↓ s'" and
  lid_inf: "infinite (UNIV :: 'l set)" and
  rid_inf: "infinite (UNIV :: 'r set)"
 shows "s ≈ s'"
```

All formal diagram proofs are faithful to the paper proofs. The only proof that required considerable extra work is the proof for the mimicking diagram. Namely, the claims

$$\mathrm{r} \notin \mathit{RID}\ \mathrm{s} \Longleftrightarrow \alpha\ \mathrm{r} \notin \mathit{RID}\ (\alpha\ (\beta\ \mathrm{s}))$$

and

$$\mathrm{l} \notin \mathit{LID}\ \mathrm{s} \Longleftrightarrow \beta\ \mathrm{l} \notin \mathit{LID}\ (\alpha\ (\beta\ \mathrm{s})),$$

assumed in Lemma 13, had to be proven formally. This required us to show similar properties for all the lower concepts in the hierarchy of types containing identifiers (values, expressions, stores and local states).

Chapter 7

Discussion

The formalization lead to the identification of a number of ambiguities in the specification of the formal semantics, and their resolution lead to changes in the side conditions of the operational semantics. The formalization also showed that the proof of determinacy could be simplified.

What is the significance of these findings? We may interpret this question pragmatically, and ask what the implications are for existing implementations. We may also take more of a conceptual approach, by taking the semantics at face value, and asking what sorts of issues the formalization exposed.

Let us first of all address the pragmatic interpretation: we do not think that our findings map to bugs in the C# and Haskell implementations. The short explanation for this is that the logic related to revision and location identifiers—which caused trouble in the formal semantics—is handled by the solid runtimes of these programming languages. For a more detailed explanation, consider the scenario described in Section 3.2, in which a revision `r` is about to join a deleted revision `r′`, and indeterminacy results from the fact that the identifier `r′` may or may not be reallocated by a concurrent fork operation. Based on the C# fragments and explanations contained in Burckhardt and Leijen's original paper [BBL10], it seems like a 'revision identifier' is simply a reference to an object instance of a `Revision` class. Thus, as long as a revision references such an object, C#'s garbage collector will not remove it, and a concurrent fork cannot replace it. Experimentation within an official online environment[1] corroborates this conjecture: a `Revision` object's hash code (accessed through the method `.GetHashCode()`) is unaffected by a join operation, and a second join attempt even returns a special exception stating that revisions cannot be joined twice. The Haskell implementation, discussed in a paper by Leijen [LFB11], seems to have similar characteristics, and the associated publication explicitly describes replacing a revision's data with an exception when it is subject to a join. We

[1]See `https://rise4fun.com/Revisions/`.

think this discrepancy between the semantics and the implementations is unfortunate, since it seems like determinacy is not preserved in the implementation setting when two revisions race to join a third. Presumably, however, the intent for the semantics was to prove determinacy *assuming that revisions are not joined twice*, meaning that rule (*join*$_\epsilon$) was merely added as a simplifying assumption for proofs.

Let us now take the semantics at face value. What sort of value can a formalization add? The result that the condition on (*fork*) has to be strengthened (Section 3.2) essentially exposes a bug that breaks an important desired safety property: the merit of this result is self-evident. The results that show that the side condition on (*new*) can be weakened, and the side conditions on (*get*) and (*set*) removed (Section 3.3), by contrast, do not expose any incorrect behaviour. Rather, they show that any faithful implementations of the transition rules could be aggressively optimized. Without the assurance provided by the rather involved proof to Lemma 1 and its mechanical verification, such optimizations may be deemed too risky to actually apply in a hypothetical system implementation. These two classes of advantages are also stressed and amply illustrated by Newcombe et al. [NRZ$^+$15], who describe the use of formal methods at Amazon Web Services.

Another advantage of the formalization is that it made us highly aware of design decisions. This shows not only in the major results, but also in some of the minor observations that we have made throughout the thesis. One may consider, for instance, the 'curious asymmetry' discussed in the remark in Section 3.2, which could lead to a subtle change in the definition of execution contexts; and the observation that extending the calculus with custom merge functions may as not be as trivial as Burckhardt and Leijen seem to suggest (Section 2.2.4).

Our simplification of the proof of determinacy, finally, is probably not very consequential, even though it may be appreciated for theoretical reasons. However, if someone ever decides to extend the calculus, resulting in more complex diagram tiling proofs, it may help make those proofs more manageable. It also makes clear that such extensions should take care to preserve the validity of the mimicking diagram.

Since the thesis was intended as a general case study of the formalization of a concurrency model, we would like to dedicate a few words to our personal experience of the overall formalization process, and the use of Isabelle/HOL in particular.

In general, we were positively surprised by just how much the relatively simple act of *specification* alone uncovered. The insights of Chapter 3, for instance, were largely a consequence of having to specify the formal semantics. Nonetheless, it was the more laborious act of verification that helped us get the details right on multiple occasions. For instance, it was the act of verification that helped us formulate the right inductive invariant for the proof to Lemma 1, and which lead us to the realization that the mimicking diagram (Lemma 13) was required. Moreover, the confidence provided by a mechanical verification cannot be underestimated.

Our overall experience of Isabelle/HOL in particular has been very positive. Our frame of reference is limited to the interactive theorem prover Coq [BC13], the model checker for the mCRL2 specfication language [GMR+07], and both the model checker (TLC) and theorem prover (TLAPS) for the TLA+ specification language [Lam02]. While we are no expert on any of these tools, it seems to us that the extent of tool support (e.g., Isabelle/jEdit has many useful features), automation support and proof language expressiveness for Isabelle/HOL is unrivalled. In some ways, this does translate to a relatively steep learning curve. In particular, since most automation methods (such as *auto*) exhibit black box behavior to an average user (and understandably so), it really is experience that helps one work effectively with these tools. Moreover, it sometimes proved challenging to find a solution to a problem, since the documentation on Isabelle/HOL is distributed over a large number of publications.

We see at least three ways in which future work could meaningfully extend the work presented in this thesis. First, the other results in Burckhardt and Leijen's original paper [BL11] could also be formalized. In particular, we think that the theorem stating that there exists a closest common ancestor for every pair of states in revision diagram be would interesting to formalize, since the property is important, and its paper proof relatively involved. Second, rule (*join*) could be generalized to support custom merge functions, as discussed in the original paper and in Section 2.2.4. Since the use of custom merge functions is a defining feature of concurrent revisions, it would be useful to clarify which general constraints such functions should satisfy exactly. Third, the calculus could be extended with features that are part of the concurrent revisions project, but are not yet formulated in any formal semantics, such as support for exceptions [LFB11] and (more substantially) incremental computation [BLS+11].

We think all of these potential extensions can leverage our formalization in two ways. First, all of the elementary definitions and the associated results can be directly reused, such as the completion equivalence lemma (Lemma 7), the result that $\approx$ is indeed an equivalence (Section 6.3), and all of the necessary (but uninteresting) lemmas required for reasoning about occurrences (Section 6.2) and renamings (Section 6.3). This eliminates a lot of tedium from future formalization efforts. Second, since most of our proofs are written using the structured Isar proof language, it should be quite easy to modify these proofs when, for instance, additional rules are introduced to the calculus: any newly generated cases can be straightforwardly integrated into the existing proofs. We consider this high degree of maintainability another great benefit of using Isabelle/HOL.

Bibliography

[ABHI08] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. *ACM SIGPLAN Notices*, 43(1):63–74, 2008.

[Bal03] Clemens Ballarin. Locales and locale expressions in Isabelle/Isar. In *International Workshop on Types for Proofs and Programs*, pages 34–50. Springer, 2003.

[BBB$^+$17] Julian Biendarra, Jasmin Christian Blanchette, Aymeric Bouzy, Martin Desharnais, Mathias Fleury, Johannes Hölzl, Ondřej Kunčar, Andreas Lochbihler, Fabian Meier, Lorenz Panny, Andrei Popescu, Christian Sternagel, René Thiemann, and Dmitriy Traytel. Foundational (co)datatypes and (co)recursion for higher-order logic. In *Frontiers of Combining Systems - 11th International Symposium, FroCoS 2017, Brasília, Brazil, September 27-29, 2017, Proceedings*, pages 3–21, 2017.

[BBD$^+$18] Julian Biendarra, Jasmin Christian Blanchette, Martin Desharnais, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. Defining (co)datatypes and primitively (co)recursive functions in Isabelle/HOL, 2018. http://isabelle.in.tum.de/dist/Isabelle2018/doc/datatypes.pdf (accessed: 24 November 2018).

[BBL10] Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. Concurrent programming with revisions and isolation types. In *ACM Sigplan Notices*, volume 45, pages 691–707. ACM, 2010.

[BBN11] Jasmin Christian Blanchette, Lukas Bulwahn, and Tobias Nipkow. Automatic proof and disproof in Isabelle/HOL. In *International Symposium on Frontiers of Combining Systems*, pages 12–27. Springer, 2011.

[BC13] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[BKdV03] Marc Bezem, Jan Willem Klop, and Roel de Vrijer, editors. *Term rewriting systems*. Cambridge University Press, 2003.

[BL11] Sebastian Burckhardt and Daan Leijen. Semantics of concurrent revisions. In *European Symposium on Programming*, pages 116–135. Springer, 2011.

[BLS$^+$11] Sebastian Burckhardt, Daan Leijen, Caitlin Sadowski, Jaeheon Yi, and Thomas Ball. Two for the price of one: a model for parallel and incremental computation. *ACM SIGPLAN Notices*, 46(10):427–444, 2011.

[CPZ08] Ariel Cohen, Amir Pnueli, and Lenore D. Zuck. Mechanical verification of transactional memories with non-transactional memory accesses. In *International Conference on Computer Aided Verification*, pages 121–134. Springer, 2008.

[DDD$^+$17] Simon Doherty, Brijesh Dongol, John Derrick, Gerhard Schellhorn, and Heike Wehrheim. Proving opacity of a pessimistic STM. In Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone, editors, *20th International Conference on Principles of Distributed Systems (OPODIS 2016)*, volume 70 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 35:1–35:17, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[DGLM13] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, 25(5):769–799, 2013.

[GMR$^+$07] Jan Friso Groote, Aad Mathijssen, Michel Reniers, Yaroslav Usenko, and Muck Van Weerdenburg. The formal specification language mCRL2. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

[GN08] Patrice Godefroid and Nachiappan Nagappan. Concurrency at Microsoft: An exploratory survey. In *CAV Workshop on Exploiting Concurrency Efficiently and Correctly*, 2008.

[Haf18] Florian Haftmann. Haskell-style type classes with Isabelle/Isar, 2018. `https://isabelle.in.tum.de/dist/Isabelle2018/doc/classes.pdf` (accessed: 24 November 2018).

[Har16] Robert Harper. *Practical foundations for programming languages*. Cambridge University Press, 2016.

[HLR10] Tim Harris, James Larus, and Ravi Rajwar. Transactional memory. *Synthesis Lectures on Computer Architecture*, 5(1):1–263, 2010.

[HMPJH05] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60. ACM, 2005.

[HPST06] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. *ACM SIGPLAN Notices*, 41(6):14–25, 2006.

[HW06] Florian Haftmann and Makarius Wenzel. Constructive type classes in isabelle. In *International Workshop on Types for Proofs and Programs*, pages 160–174. Springer, 2006.

[HWC$^+$04] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. *SIGARCH Comput. Archit. News*, 32(2):102–, March 2004.

[KL13] Andrei A. Kirilenko and Andrew W. Lo. Moore's law versus Murphy's law: Algorithmic trading and its discontents. *Journal of Economic Perspectives*, 27(2):51–72, 2013.

[KM66] Richard M. Karp and Rayamond E. Miller. Properties of a model for parallel computations: determinacy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411, 1966.

[Kra] Alexander Krauss. Defining recursive functions in Isabelle/HOL. `https://isabelle.in.tum.de/doc/functions.pdf` (accessed: 24 November 2018).

[Lam02] Leslie Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[LFB11] Daan Leijen, Manuel Fahndrich, and Sebastian Burckhardt. Prettier concurrency: purely functional concurrent revisions. In *ACM SIGPLAN Notices*, volume 46, pages 83–94. ACM, 2011.

[LPSZ08] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *ACM SIGOPS Operating Systems Review*, 42(2):329–339, 2008.

[MHC+06] Chaiyasit Manovit, Sudheendra Hangal, Hassan Chafi, Austen McDonald, Christos Kozyrakis, and Kunle Olukotun. Testing implementations of transactional memory. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 134–143. ACM, 2006.

[Nip18] Tobias Nipkow. What's in Main, 2018. https://isabelle.in.tum.de/dist/Isabelle2018/doc/main.pdf (accessed: 16 November 2018).

[NPW18] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, 2018.

[NRZ+15] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services uses formal methods. *Communications of the ACM*, 58(4):66–73, 2015.

[Pau89] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

[Pau18a] Lawrence C. Paulson. Isabelle's logics, 2018. https://isabelle.in.tum.de/dist/Isabelle2018/doc/logics.pdf (accessed: 16 November 2018).

[Pau18b] Lawrence C. Paulson. Isabelle's logics: FOL and ZF, 2018. https://isabelle.in.tum.de/dist/Isabelle2018/doc/logics-ZF.pdf (accessed: 16 November 2018).

[Pau18c] Lawrence C. Paulson. Old introduction to Isabelle, 2018. https://isabelle.in.tum.de/dist/Isabelle2018/doc/intro.pdf (accessed: 16 November 2018).

[ST97] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[Wen12] Makarius Wenzel. Isabelle/jedit–a prover IDE within the PIDE framework. In *International Conference on Intelligent Computer Mathematics*, pages 468–471. Springer, 2012.

[Wen18a] Makarius Wenzel. The Isabelle/Isar reference manual, 2018. https://isabelle.in.tum.de/dist/Isabelle2018/doc/isar-ref.pdf (accessed: 16 November 2018).

[Wen18b] Makarius Wenzel. Isabelle/jEdit, 2018. https://isabelle.in.tum.de/dist/Isabelle2018/doc/jedit.pdf (accessed: 16 November 2018).