Compactness via Residual Stepping

Matias Scharager
Carnegie Mellon University, USA

Abstract

The compactness lemma in programming language theory states that any use of a recursive function can be simulated using a finite unrolling of the function. It is a cornerstone of the *logical relations* proof technique for proving liveness properties of typed programs. However, the proof of the compactness lemma is often messy because the multiple copies made of the recursive function during execution can be unrolled an inconsistent number of times, and the overhead cost of this bookkeeping becomes even more apparent when this proof is mechanized. Known proof techniques that aim to simplify this bookkeeping do not generalize to languages with non-local control flow, hence the need for a new proof approach that maintains generality while minimizing bookkeeping.

We present a novel proof technique by utilizing an intermediate *residual* language to mechanize bookkeeping, simplifying the overall compactness lemma proof and extending it to a wider range of programming languages. We demonstrate this approach by formally verifying the compactness lemma within the Rocq theorem prover for a language featuring non-local control flow and polymorphism.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases Compactness Lemma, Residual Language, Formal Verification Proof Technique, Logical Relations

1 Introduction

The name "compactness" within the field of programming language theory was first used for domain-theoretic topics dealing with infinite objects. When we encode a function in a program, we define an output for each possible input to the function, even if the collection of all inputs is infinite. The key idea of compactness is that we are only capable of using a finite amount of this information during any program execution that eventually terminates.

The compactness lemma of this paper is this pattern of reasoning applied to the recursive depth of a function. This helps bring several important properties of denotational semantics and domain theory into the realm of operational semantics, bridging the gap between the representations of programming languages [13].

The central principle behind compactness is how the recursive depth of the function changes during execution. Within a terminating program, the function had to have recursively called itself up to a finite recursive depth, so replacing the function with a finite unrolling with the same behavior up to that recursive depth should result in the same outcome.

While the compactness lemma is easy to believe informally, the need for substantial bookkeeping on the depth of all the instances of the recursive function during execution makes it difficult to break down the lemma into easily verified components.

1.1 Practical Application of Compactness

In practice, the compactness lemma is utilized primarily for proving important properties of logical relations. Logical relations are equivalences built inductively on type structure that allows for reasoning about various behavioral properties of programs. The theorem that enables the use of these logical relations is known as the fundamental theorem of logical relations (FTLR) which states that all well-typed terms belong to the logical relation. Recursive functions create circular reasoning within the proof of FTLR, and compactness

resolves this circular reasoning by allowing us to induct on the finite recursive depth of the function.

A detailed example of how compactness (referred to as the unwinding theorem) is used to verify FTLR can be found in Pitts [16]. We will briefly summarize this approach to showcase the usefulness of compactness.

Suppose that we have some logical relation R over closed values of the language, defined recursively on the types, where the case for recursive functions appears as some variation of

$$(\texttt{fun}\ f(x : A) : B\ \texttt{is}\ e) \in R[A \to B] =$$
$$\forall v \in R[A].\ ([\texttt{fun}\ f(x : A) : B\ \texttt{is}\ e, v/f, x]e) \in R[B].$$

To prove that $F \in R[A \to B]$, we need to show that a substitution of $[F/f]$ is a related substitution under R, which requires showing that $F \in R[A \to B]$, hence creating a circular argument.

Now suppose that F_n is the nth unwinding of the recursive function where it exactly matches the behavior of F up to a recursive depth of n. We can break free of the circular argument by inducting on this unwinding through the admissibility theorem.

▶ **Theorem 1** (Admissibility). *If for all n, $F_n \in R[A \to B]$, then we have $F \in R[A \to B]$.*

Finally, to prove admissibility, we need to show the compactness lemma: any instance of F within a whole program can be computationally modeled by a specific unrolling F_i. Since admissibility holds for all n, then the specific i we care about is covered by the assumption, and admissibility holds.

1.2 A Road Map to Formalization

At its base, the compactness lemma is a proof by induction, but one that does not necessarily lend itself well to be formalized in a proof assistant. This induction requires bookkeeping not only all the locations where the function in question occurs, but also the depth of unrolling at each one of these locations, and then maintaining all this bookkeeping as the program executes. While this bookkeeping might be feasible on paper, it can get out of hand rather quickly in a mechanization where the overhead cost becomes more evident.

Proper effort in constructing definitions before diving into a proof often results in a smoother formal verification experience overall. To this end, we find that defining the residual language explained in Section 3 allows us to break down the compactness lemma into several smaller intermediate lemmas that are all straightforward to mechanize.

This paper starts by presenting the proof method in generalized way without specializing to any particular language definition. Section 2 explains the definitions needed to formally express the compactness lemma along with denoting the base assumptions we make of the language in question in order to construct our proof, which as it turns out is a reasonable set of assumptions for any programming language. We express the residual language in Section 3 and motivate why we would want such a definition. This finally leads us to Section 4 where express the proof sketch of the compactness lemma through a series of helper lemmas and corollaries.

Some previous mechanized verifications of the compactness lemma [9, 8] also manage to provide a succinct rendition of bookkeeping, but they do so at the cost of generality. Importantly, these approaches fail in the setting of non-local control flow, necessitating the development of a more generalized approach, hence our result. We highlight the generality of our residual stepping proof procedure in Section 5. There, we pick out a few key language

$$\tau := \cdots \mid \tau \to \tau$$
$$e := \cdots \mid \texttt{fun}\ f(x:\tau):\tau\ \texttt{is}\ e \mid e\ e$$

$$\frac{\Gamma, f:\tau_1\to\tau_2, x:\tau_1 \vdash e:\tau_2}{\Gamma \vdash \texttt{fun}\ f(x:\tau_1):\tau_2\ \texttt{is}\ e : \tau_1\to\tau_2}\ {\to}I \qquad \frac{\Gamma\vdash e_1:\tau_2\to\tau \qquad \Gamma\vdash e_2:\tau_2}{\Gamma\vdash e_1\ e_2:\tau}\ {\to}E$$

$$\frac{}{\texttt{fun}\ f(x:\tau):\tau'\ \texttt{is}\ e'\ \texttt{val}}\ \texttt{fun val}$$

$$\frac{e\ \texttt{val}}{(\texttt{fun}\ f(x:\tau):\tau'\ \texttt{is}\ e')(e) \longmapsto [\texttt{fun}\ f(x:\tau):\tau'\ \texttt{is}\ e', e/f, x]e'}\ \texttt{app}\longmapsto$$

Figure 1 Relevant components of the programming language

features that can make the compactness lemma harder to formally verify through other proof procedures, and proceed to provide a mechanization for that setting in Section 6 as a proof of concept. No generative AI was used in the development of this mechanization.

2 Formally Expressing Compactness

To properly define the compactness lemma, we must first define a programming language with recursive functions to work with. The relevant syntax we use can be found in Figure 1 and we utilize the standard call-by-value dynamics for this generalized language. This proof strategy does not rely on any specific definition of a programming language other than the basic facts that substitution is properly defined and the following two properties of the reduction semantics hold true.

▶ **Lemma 2** (Determinism).
1. *If* $e \longmapsto e_1$ *and* $e \longmapsto e_2$ *then* $e_1 = e_2$
2. *Not both of* e $\texttt{val}$ *and* $e \longmapsto e'$

▶ **Lemma 3** (Safety). *For all well-formed programs* e*, either there exists some well-formed program* e' *such that* $e \longmapsto e'$ *or* e $\texttt{val}$*.*

The specific definition of "well-formed" we utilize is that of a typing judgment, and it is standard practice to prove the type safety of typed languages. However, this formalization of compactness does not directly rely on the types themselves and can be extended to an un-typed setting, as long as there is a meaningful property of "well-formed" that shows this is a reduction system.

Fix an arbitrary closed function $F = \texttt{fun}\ f(x:\tau_1):\tau_2\ \texttt{is}\ e_f$ that we wish to represent by its unrolling via compactness. This parameter is passed along to the rest of the formalization as input arguments so that the formalization can abstract over the function in question.

We also create more convenient syntax for expressing multiple steps taken. The judgment $e \longmapsto^n e'$ signifies that e steps to e' in n number of steps for a natural number n. The judgment $e\downarrow$ signifies that e terminates, or more formally that there exists a program v and natural number n such that $e \longmapsto^n v$ and v $\texttt{val}$.

Much like prior works, we define function unrollings with the syntax of Figure 2. We use F_ω as a convenient notation for the full recursive function F and use F_n to represent

$$
\begin{aligned}
F_0 &= \texttt{fun}\ g(y : \tau_1) : \tau_2\ \texttt{is}\ (g)(y) \\
F_{n+1} &= \texttt{fun}\ \_(y : \tau_1) : \tau_2\ \texttt{is}\ [F_n, y/f, x]e_f \\
F_\omega = F &= \texttt{fun}\ f(x : \tau_1) : \tau_2\ \texttt{is}\ e_f
\end{aligned}
$$

Figure 2 Function unrollings for $F = \texttt{fun}\ f(x : \tau_1) : \tau_2\ \texttt{is}\ e_f$

$$
\begin{aligned}
\tau &:= \cdots \mid \tau \to \tau \\
e &:= \cdots \mid \textbf{fun}\ f\ (x : \tau) : \tau.\ e \mid e\ e \mid \omega
\end{aligned}
$$

$$
\frac{}{\Gamma \vdash \omega : \tau_1 \to \tau_2}\ \omega I
\qquad
\frac{}{\omega\ \texttt{val}^\omega}\ \omega\ \texttt{val}^\omega
\qquad
\frac{e\ \texttt{val}^\omega}{(\omega)(e) \longmapsto_\omega [\omega, e/f, x]e_f}\ \omega \longmapsto_\omega
$$

Figure 3 Residual stepping extension with respect to $F = \texttt{fun}\ f(x : \tau_1) : \tau_2\ \texttt{is}\ e_f$

an unrolling that matches the full recursive function up to a depth of n. The base case F_0, commonly referred to as $\lambda.\bot$, is the simplest way to express a function that will loop infinitely when called, allowing for an easy way of expressing non-termination. We are able to prove the non-termination of calling F_0 directly as a lemma.

▶ **Lemma 4** (Bottom). *$(F_0)(v)$ does not terminate for any v $\texttt{val}$.*

Lemma 4 is the only place in this formalization reliant on determinism, Lemma 2. This means this method of proving compactness can extend to any non-deterministic language that admits Lemma 4, assuming some reasonable definition of termination in a non-deterministic setting.

With all this in place, we can express our end goal of the compactness lemma for our chosen function.

▶ **Lemma 5** (Compactness). *$[F_\omega/x]e \downarrow$ iff $\exists n.\ [F_n/x]e \downarrow$.*

This is strong enough to prove the admissibility of a logical relation, which is what the compactness lemma is normally used for in practice.

3 Residual Language Definition

If we observe the two programs $[F_\omega/x]e$ and $[F_n/x]e$, we note that they syntactically coincide with each other precisely at e with open variable x before substitution occurs. Assuming both programs terminate, we can also observe that the chain of transitions matches syntactically in exactly the places where the function F is not present. We would like to model this matching shape of execution, so we need a language that models the behavior of the open term e without accounting for the functions being substituted in, essentially executing an open term. We do this by adding a placeholder ω in place of the function F, and augment the language with dynamics for ω that simulates the function F.

Abstracting over the function we care about, we define a residual language that extends our original language as defined in Figure 3. All other dynamic rules are recursively defined in the same way, replacing all instances of $\texttt{val}$ and $\longmapsto$ with $\texttt{val}^\omega$ and $\longmapsto_\omega$ respectively.

The ω is a reserved variable name that is global in scope, and unlike other variables that occur locally in Γ, it has associated dynamics derived from F. A "closed" program within this language can contain the free variable ω because the dynamics define how to handle the execution of this variable. We refer to programs in this language as "residuals" because they what is left of the original terms after the occurrences of the F in question have been replaced with ω.

3.1 Relating Terms to Residuals

We now have an alternative means of representing the program $[F_\omega/x]e$ utilizing the residual language, that is by converting e into its corresponding residual p where the open variable x is replaced with the special variable ω. We likewise have a means of representing $[F_n/x]e$ with the same residual p, but in this translation, we lose track of the depth of the unrolling. We wish to express this correspondence by defining a relation between terms and residuals, and indexing the relation so we can keep track of the depth of unrolling.

We define e $\mathbf{of}^n$ p to be the least compatible relation between terms and residuals admitting F_ω $\mathbf{of}^n$ ω and for all $i \geq n$, F_i $\mathbf{of}^n$ ω. As such, e $\mathbf{of}^n$ p signifies that term e exactly syntactically matches residual p except in the specific places that ω is in p, which represents the holes left behind after removing occurrences of F_i and F_ω. We choose the name **of** because we see that the term is made "of" the residual by filling the gaps in the residual with the desired function unrolling.

Moreover, e $\mathbf{of}^n$ p specifies that e must have unrollings of F of a depth of at least n, specifying the minimum possible unrolling depth. The benefit of this definition is that it condenses bookkeeping all instances of the unrolling into maintaining a single number representing the minimum possible unrolling that we can use for induction in later proofs.

Maintaining only the minimum presents an underestimate of the present function unrollings. Our proofs must always assume the worst case of the minimum function unrolling, thus often requiring a greater depth of recursion than is actually needed. However, the burden of this inefficiency affects only the mathematical formalization rather than the program execution itself, and the compactness lemma cares only about the existence of a depth of unrolling without caring about the minimum such depth, hence there's no issue.

This flexibility allows us to arbitrarily decrease the minimum counter whenever we deem it necessary, and we express this property explicitly via Lemma 6. This creates an explicit ordering on the $\mathbf{of}^n$ relation based on the natural number n where the weakest such relation is $\mathbf{of}^0$ where any level of function unrollings is allowed.

▶ **Lemma 6** (Decrement Residual). *e $\mathbf{of}^{n+1}$ p implies e $\mathbf{of}^n$ p.*

Lemma 6 is used mainly for inductive cases of the proof where the function in question is not called. These cases do not require a decrement of the index, but we force it to decrement anyway for ubiquity of bookkeeping between all cases, hence the underestimation.

4 Proof Strategy

The main principle of the proof strategy for compactness is an indexed relation between term and residual dynamic judgments expressed by the $\mathbf{of}^n$ relation. Under certain assumptions expressed in the lemmas, we can convert between `val` and $\texttt{val}^\omega$ and between $\longmapsto$ and $\longmapsto_\omega$. We can think of this process as removing all instances of the function F in question to leave us with the residual program, then refilling the residual program with the new instances of the function F at potentially different unrolling depths.

$$\begin{array}{ccc} e & \longmapsto & e' \\ \mathbf{of}^0 & & \mathbf{of}^0 \\ p & \longmapsto_\omega & p' \\ \mathbf{of}^{n+1} & & \mathbf{of}^n \\ d & \longmapsto & d' \end{array}$$

Figure 4 Combination of Lemma 8 and Lemma 10 assuming $e \downarrow$

We start first by relating `val` and $\texttt{val}^\omega$. This is necessary for the overall proof of compactness and also within the later lemmas if the definition of $\longmapsto$ is dependent on `val` as in the call-by-value setting. If we have $e\ \mathbf{of}^n\ p$ and focus specifically on the places where e and p are distinct, we end up with F_i on the left and ω on the right, both of which are values by `fun val` and $\omega\ \texttt{val}^\omega$, making the two judgments match up.

▶ **Lemma 7** (Value Residual). *Given* $v\ \mathbf{of}^n\ v_p$, *then* v `val` *iff* $v_p\ \texttt{val}^\omega$.

Proof. The forward direction is proved by induction on v `val` and inversion on $v\ \mathbf{of}^n\ v_p$. Similarly, the backwards direction is proved by induction on $v_p\ \texttt{val}^\omega$ and inversion of $v\ \mathbf{of}^n\ v_p$ ◀

The rest of the lemmas are highlighted in Figure 4. We start with the givens $e \longmapsto e'$, $e\ \mathbf{of}^0\ p$, and $d\ \mathbf{of}^{n+1}\ p$, then we acquire $p \longmapsto_\omega p'$ and $e'\ \mathbf{of}^0\ p'$ from Lemma 8, then we conclude $d \longmapsto d'$ and $d'\ \mathbf{of}^n\ p'$ from Lemma 10. These steps require the main bulk of the formalization proof effort as they will involve the specific cases of interest of calling the function in question.

▶ **Lemma 8** ($\longmapsto$ to $\longmapsto_\omega$). *If* $e \downarrow$ *and* $e\ \mathbf{of}^0\ p$ *and* $e \longmapsto e'$ *then there exists* p' *such that* $p \longmapsto_\omega p'$ *and* $e'\ \mathbf{of}^0\ p'$.

Proof. By induction on $e \longmapsto e'$ and inversion on $e\ \mathbf{of}^n\ p$. Most of the cases are trivial as $\longmapsto_\omega$ contains all the same rules as $\longmapsto$, and we can utilize Lemma 7 and the induction hypothesis to convert between them.

For the case of $e = (F_0)(v)$, we utilize Lemma 4 to show this case contradicts $e \downarrow$.

For the case of $e = (F_{i+1})(v)$ and $p = (\omega)(v)$, we know that $(F_{i+1})(v) \longmapsto [F_i, v/f, x]e_f$ through the `app` $\longmapsto_\omega$ rule, and we know $(\omega)(v) \longmapsto_\omega [\omega, v/f, x]e_f$ through the $\omega \longmapsto_\omega$ rule, so we show $[F_i, v/f, x]e_f\ \mathbf{of}^0\ [\omega, v/f, x]e_f$ by the definition of it being the least compatible relation.

For the case of $e = (F_\omega)(v)$ and $p = (\omega)(v)$, we know that $(F_\omega)(v) \longmapsto [F_\omega, v/f, x]e_f$ through the `app` $\longmapsto_\omega$ rule, and we know $(\omega)(v) \longmapsto_\omega [\omega, v/f, x]e_f$ through the $\omega \longmapsto_\omega$ rule, so we show $[F_\omega, v/f, x]e_f\ \mathbf{of}^0\ [\omega, v/f, x]e_f$ by the definition of it being the least compatible relation. ◀

Note that Lemma 8 relies on the assumption of $e\ \mathbf{of}^0\ p$ and results in $e'\ \mathbf{of}^0\ p'$ in the conclusion. We may strengthen the conclusion to $e'\ \mathbf{of}^n\ p'$ for any natural number n desired, as long as we strengthen the assumption to be that of $e\ \mathbf{of}^{n+1}\ p$, thus removing the need for the $e \downarrow$ judgment. However, this generalization is not sufficient for the backward direction of compactness $(\exists n.\ [F_n/x]e \downarrow) \implies [F_\omega/x]e \downarrow$ since it fails several edge cases involving the possibility where $n = 0$ and the function is never called during execution. It is easier to consolidate these cases into this single lemma as stated since the weaker conclusion of $e'\ \mathbf{of}^0\ p'$ is all that is required in this overall proof strategy.

▶ **Corollary 9** ($\longmapsto^n$ to $\longmapsto_\omega^n$). *If* e $\mathbf{of}^0$ p *and* $e \longmapsto^n e'$ *and* e' `val` *then there exists* p' *such that* $p \longmapsto_\omega^n p'$ *and* e' $\mathbf{of}^0$ p'.

Having made all the effort to convert into $\longmapsto_\omega$, we now reap the benefits of having these empty holes in the residual which we can fill in with whatever unrolling is desired. The preservation of the residual structure for this arbitrary filling is expressed in the following lemma.

▶ **Lemma 10** ($\longmapsto_\omega$ to $\longmapsto$). *If* d $\mathbf{of}^{n+1}$ p *and* $p \longmapsto_\omega p'$ *then there exists* d' *such that* $d \longmapsto d'$ *and* d' $\mathbf{of}^n$ p'.

Proof. By induction on $p \longmapsto_\omega p'$ and inversion on d $\mathbf{of}^{n+1}$ p. Once again, most of the cases are trivial as we can utilize Lemma 7 and the induction hypothesis to convert between $\longmapsto_\omega$ and $\longmapsto$. We utilize Lemma 6 to decrease the minimum required depth of unrollings by one whenever applicable.

The case of $d = (F_i)(v)$ and $p = (\omega)(v)$ for $i < n+1$ contradicts d $\mathbf{of}^{n+1}$ p as the depth of unrolling is less than the stated minimum of $n+1$.

For the case of $d = (F_i)(v)$ and $p = (\omega)(v)$ for $i \geq n+1$, we know $(\omega)(v) \longmapsto_\omega [\omega, v/f, x]e_f$ through the $\omega \longmapsto_\omega$ rule, and we know that $(F_i)(v) \longmapsto [F_{i-1}, v/f, x]e_f$ through the $\texttt{app} \longmapsto_\omega$ rule (since $i > 0$), so we show $[F_{i-1}, v/f, x]e_f$ $\mathbf{of}^n$ $[\omega, v/f, x]e_f$ by the definition of it being the least compatible relation since $i - 1 \geq n$.

For the case of $d = (F_\omega)(v)$ and $p = (\omega)(v)$, we know $(\omega)(v) \longmapsto_\omega [\omega, v/f, x]e_f$ through the $\omega \longmapsto_\omega$ rule, and we know that $(F_\omega)(v) \longmapsto [F_\omega, v/f, x]e_f$ through the $\texttt{app} \longmapsto_\omega$ rule, so we show $[F_\omega, v/f, x]e_f$ $\mathbf{of}^n$ $[\omega, v/f, x]e_f$ by the definition of it being the least compatible relation. ◀

▶ **Corollary 11** ($\longmapsto_\omega^n$ to $\longmapsto^n$). *If* d $\mathbf{of}^n$ p *and* $p \longmapsto_\omega^n p'$ *then there exists* d' *such that* $d \longmapsto^n d'$ *and* d' $\mathbf{of}^0$ p'.

Note that this d $\mathbf{of}^n$ p assumption for Corollary 11 is dependent on the total number n of steps taken. In most cases, it is possible to decrease the strength of the assumption to be lower than n, but proving such a lemma would require the excessive bookkeeping we are trying to avoid and is unnecessary to prove our desired version of compactness.

We are now able to prove a generalized notion of compactness that follows very naturally by combining all the lemmas we have in place. This lemma is strictly stronger than our desired definition of compactness in Lemma 5 as we know by definition that $[F_i/x]e$ $\mathbf{of}^i$ $[\omega/x]e$ from the definition of it being the least compatible relation.

▶ **Lemma 12** (Generalized Compactness). *If* $e \downarrow$ *in* n *steps, then for all* p *and* d *such that* e $\mathbf{of}^0$ p *and* d $\mathbf{of}^n$ p, *we know* $d \downarrow$ *in* n *steps.*

Proof. First, decompose $e \downarrow$ into $e \longmapsto^n e'$ and e' `val`. Then, apply Corollary 9 to get $p \longmapsto_\omega^n p'$ where e' $\mathbf{of}^0$ p'. From here, we can apply Lemma 7 to get that p' $\texttt{val}^\omega$. Next, we apply Corollary 11 to get $d \longmapsto^n d'$ and d' $\mathbf{of}^0$ p'. We finish by showing d' `val` via Lemma 7. ◀

5 Generality of Residual Stepping

We consider two metrics to demonstrate the efficacy of the residual stepping proof strategy. One goal is to minimize bookkeeping, and through the various definitions and lemmas we have expressed, we have managed to minimize the amount of bookkeeping sufficiently to

make the process of formally verifying the compactness lemma feasible. However, some previous existing approaches [9, 8] also do well under this metric: this is not the first formally verified proof of the compactness lemma.

In turn, these previous approaches sacrifice the metric of generality by relying on more assumptions about the programming language in question. As more language features are added to the language, these assumptions could become invalidated, necessitating a complete reconstruction of the proof overall. On the other hand, the residual stepping approach is driven almost purely by the dynamics of the language, making almost no assumptions about the type theory of the language, or any of the technical components and constructors of the language, as all we care about are reduction semantics. As such, this proof technique can be applied to any compactness lemma with almost no modification by simply handling the extra trivial compatibility cases.

To showcase the generality of residual stepping, we will consider various different language features and explain how the proof approach can be adapted to that setting. We select some of these to include in our proof of concept formulation in Section 6.

This proof strategy does cleanly extend to the dependently typed setting and polymorphic typed setting without any modification. The reason for this is that the dynamics of the term are not dependent on the type, as the type is only used for static checking before execution. Our Rocq implementation demonstrates this method works for polymorphic types.

Bekić's theorem [5] allows us to extend the analysis of a single recursive function to that of mutually recursive functions. However, we find that our proof framework still applies to the mutual-recursion setting with a small modification. Specifically, whereas before we only had one recursive function F_ω and one residual variable ω, we now have n recursive functions $F_{\omega,n}$ and n residual variables ω_n to keep track of. The minimum depth of unrolling can be shared among all of these functions, so the interface of the $\mathbf{of}^n$ judgment remains unchanged in all proofs, but we must add these extra cases into the definition of the $\mathbf{of}^n$ judgment. This adds a small amount of overhead to the substitution lemmas for $\mathbf{of}^n$ and adds some base cases to Lemmas 8 and 10, but these changes are straightforward to make.

For the sake of expressing a language where the residual stepping technique does not hold, we can consider the extreme case where the language can read and branch on its syntax during execution. In this setting, program equivalence is trivialized to syntactic equivalence, making it generally uninteresting to reason about and removing the need for the compactness lemma in the first place. The compactness lemma itself does not hold true in such a setting, due to programs being able to syntactically distinguish between a recursive function and the finite unrolling of it.

This esoteric case does not disqualify the generality of the residual stepping approach. On the contrary, this suggests a deeper if and only if connection between the residual stepping approach and the compactness lemma, suggesting that the compactness lemma is only true precisely when it can be proven through the residual stepping method.

5.1 Non-Local Control Flow Through Continuations

An important language feature to consider is that of non-local control flow, such as through the use of continuations. In such a setting, a program can choose to switch its environment of execution, or its evaluation context, by simply passing a value into a continuation, switching into the environment stored within that continuation.

The expressive nature of non-local control flow invalidates an important lemma often used for analyzing the computation of programs: contextual equivalence is no longer adequate

with respect to evaluation. Previous proof approaches for the compactness lemma that rely on this assumption cannot be generalized to the setting of non-local control flow.

We state that a relation R is adequate with respect to evaluation if for any e_1 and e_2 such that $e_1 \longmapsto e_2$, we know that $e_1\ R\ e_2$. Previous formally verified approaches to the compactness lemma [8] relied on this property holding for contextual equivalence, specifically that if $e_1 \longmapsto e_2$, then for any context C, $C[e_1] \downarrow$ if and only if $C[e_2] \downarrow$. Adequacy greatly simplifies the compactness lemma proof because we can relate $[F_i/x]e$ and $[F_\omega/x]e$ through the compatibility argument that comes directly from the definition of contextual approximation, and then utilize adequacy to maintain this relation throughout the execution of the programs.

Adequacy for contextual equivalence breaks due to the mechanism that allows non-local control flow to discard the current executing environment. We explain this in more detail through a counterexample once we specify the language in Section 6, but for now we can interpret it as whenever we attempt to analyze the result of a computation e by using a context C, e can through its own execution refuse to be analyzed by the context C by escaping to a different context.

The residual stepping approach, unlike previous formal verification approaches, can be generalized to this setting of non-local control flow as it only relies on the minimal set of properties expressed in Section 2, not including adequacy. The **of** relation naturally extends to program contexts, and analyzing the whole program allowing us to maintain the desired invariants in all possible environments the program may choose to switch to.

6 Rocq Formalization

As a proof of concept, we have chosen to formally verify compactness in an expressive language containing non-local control flow effects and polymorphism. In the remainder of the paper, we go into deeper detail about this specific application of residual stepping and how it can be formally verified in Rocq. All proofs were verified in version 9.0 of Rocq. The formulation is linked here: https://github.com/pi314mm/compactness. No generative AI was used in the development of this mechanization.

Reasoning about programs with non-local control effects adds an extra layer of difficulty to the formalization in every aspect. We shall closely examine how several lemmas throughout the formalization adapt to handle this extended case.

6.1 Language Specification

The first design decision we had to make was choosing between different representations of stack frames and evaluation contexts. Harper [11] presents one plausible formulation that is very explicit in how values are returned to stack frames by making use of two different judgments: $k \triangleright e$ to represent that term e has computations to be done, and $k \triangleleft v$ to represent that term v is a value and is returning up the stack. This formulation has the benefit that the continuation component is kept separate from the term component, so the next step to take in execution is always immediately obvious, making dynamic stepping rules require no premises. However, we opted against this formulation due to the additional machinery involved in making multiple judgments, and also due to the extra steps needed to convert between the two judgments explicitly.

We instead choose to utilize evaluation contexts $E[e]$ for continuation E and term e with a slight twist on the definition of the dynamics. Normally with evaluation contexts, the inductive cases of the stepping relation are made implicit, utilizing a two-part relation $E[e] \longmapsto e'$ where e is a redex. Instead, we utilize a three-part relation $E; e \longmapsto e'$ and

$$
\begin{aligned}
\tau &:= 0 \mid 1 \mid \tau \to \tau \mid \tau \times \tau \mid \forall \alpha.\tau \mid \exists \alpha.\tau \mid \texttt{cont}(\tau) \\
e &:= \texttt{let}\ e = (x)\ \texttt{in}\ e \mid \texttt{absurd}\{\tau\}(e) \mid () \\
&\quad \mid \texttt{fun}\ f(x:\tau):\tau\ \texttt{is}\ e \mid e\ e \mid \langle e, e\rangle \mid \pi_1(e) \mid \pi_2(e) \\
&\quad \mid \Lambda x.e \mid (e)[\tau] \mid \texttt{pack}\ (\tau, e)\ \texttt{as}\ \exists x.\tau \\
&\quad \mid \texttt{open}\ (e)\ \texttt{as}\ (\alpha, x)\ \texttt{in}\ (e) \mid \texttt{letcc}\{\tau\}(x.e) \\
&\quad \mid \texttt{throw}(e; e) \mid \texttt{cont}(e) \\
E &:= [\cdot]
\end{aligned}
$$

Figure 5 Full grammar of language

$$
\frac{\Gamma, x : \texttt{cont}(\tau) \vdash e : \tau}{\Gamma \vdash \texttt{letcc}\{\tau\}(x.e) : \tau}
\qquad
\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \texttt{cont}(\tau)}{\Gamma \vdash \texttt{throw}(e_1; e_2) : 0}
\qquad
\frac{E : (\cdot \vdash \tau) \Rightarrow (\cdot \vdash 1)}{\Gamma \vdash \texttt{cont}(E) : \texttt{cont}(\tau)}
$$

$$
\frac{}{\texttt{cont}(E)\ \texttt{val}}
\qquad
\frac{}{E; \texttt{letcc}\{\tau\}(x.e) \longmapsto E[[\texttt{cont}(E)/x]e]}
\qquad
\frac{E[\texttt{throw}([\cdot]; e_2)]; e_1 \longmapsto e'}{E; \texttt{throw}(e_1; e_2) \longmapsto e'}
$$

$$
\frac{e_1\ \texttt{val} \qquad E[\texttt{throw}(e_1; [\cdot])]; e_2 \longmapsto e'}{E; \texttt{throw}(e_1; e_2) \longmapsto e'}
\qquad
\frac{e\ \texttt{val}}{E; \texttt{throw}(e; \texttt{cont}(E')) \longmapsto E'[e]}
$$

Figure 6 Selected statics and dynamics

explicitly define the inductive cases of the stepping relation by transferring stack frames from e to E. This gives us the best of both approaches where we have better control over the inductive cases through stack frames, but we don't have to deal with the secondary judgment of returning a value to a continuation.

The complete grammar of our language is shown in Figure 5. This syntax was defined as one inductive datatype within Rocq which meant we could reuse the syntax from term to define our evaluation context stack frames, hence only needing the one extra base case, and we only need to implement substitution once to cover substitutions into all three constructs. The syntax can be found in the SyntaX.v file of the Rocq supplement.

Variables in the language are implemented using De Bruijn indices. We utilized the explicit substitution framework implemented by Crary [8] with some modifications for ease of use. This framework gives us various tools for substitution in a generalized setting which we use to simplify several substitution goals throughout our formalization. The files related to this can be found in the EasySubst folder of the Rocq supplement.

Figure 6 illustrates the static and dynamic typing judgments associated with non-local control flow. For the static typing judgments, the important things to note are that $\texttt{throw}(e_1; e_2)$ is of type 0 and that $\texttt{cont}(E)$ requires that E be an evaluation context converting from τ in the empty context to 1 in the empty context. The static and dynamic typing judgments can be found in the Rules.v file of the Rocq supplement.

We opted to restrict our $\texttt{throw}(e_1; e_2)$ to the type 0 since we have $\texttt{absurd}\{\tau\}(e)$ in our language capable of changing the type 0 into whatever we want. Marking $\texttt{throw}(e_1; e_2)$ as the type 0 helps us identify at a type level that the computation will never return a value to

the current evaluation context, allowing for certain lemmas outside the scope of compactness to hold vacuously true.

The reason for the restriction on the evaluation context having a return type of 1 is how it affects the safety proof of the language. We restrict our definition of closed, whole programs to being closed terms of the answer type 1, meaning that all evaluation contexts produced during evaluation have the return type of 1. Enforcing this via the statics eliminates the need to maintain this as an invariant in the proof of type safety, making it a lot easier to formally verify within Rocq.

The small-step semantics for $\texttt{letcc}\{\tau\}(x.e)$ and the last case of $\texttt{throw}(e; \texttt{cont}(E'))$ are standard rules when dealing with continuations, as the three-part relation version doesn't look much different from the two-part relation version. The two other inductive rules for $\texttt{throw}(e_1; e_2)$ demonstrate how stack frames from the computation are transferred over to the evaluation context explicitly, allowing us greater control over these inductive cases when we induct over the stepping relation.

6.2 Language Properties

It is now easy to devise the counterexample that violates adequacy of contextual equivalence. Consider the single-step evaluation of $[\cdot]; \texttt{throw}([\cdot]; ()) \longmapsto ()$, where the program escapes the current environment. Adequacy of contextual equivalence would state that for all contexts C, $C[\texttt{throw}([\cdot]; ())] \downarrow$ if and only if $C[()] \downarrow$. However, the context $\texttt{let}\ x = ([\cdot])\ \texttt{in}\ \bot$ contradicts this claim by distinguishing between these two terms since $\texttt{let}\ x = (\texttt{throw}([\cdot]; ()))\ \texttt{in}\ \bot$ terminates but $\texttt{let}\ x = (())\ \texttt{in}\ \bot$ does not. Thankfully, our proof does not depend on this result, and we only need to prove the minimal properties delineated in Section 2.

The next step we take is to prove several lemmas about substitutions in the typing judgments as a means of defining substitution. The generalized substitution framework provides some tools for reasoning about substitutions, but we need to adapt it to the setting of non-local control flow as well. These substitution-related lemmas can be found in the Substitution.v file of the Rocq supplement.

With everything in place, we are finally able to tackle the type safety theorem. However, in the setting of non-local control flow, the theorem looks slightly different from one would normally expect, reflecting the non-local transfer of control.

▶ **Theorem 13** (Progress). *If* $\cdot \vdash e : \tau$ *then either* $e\ \texttt{val}$ *or* $\forall E : (\cdot \vdash \tau) \Rightarrow (\cdot \vdash 1).\ \exists e'.\ E; e \longmapsto e'$

▶ **Theorem 14** (Preservation). *If* $\cdot \vdash e : \tau$ *and* $E : (\cdot \vdash \tau) \Rightarrow (\cdot \vdash 1)$ *and* $E; e \longmapsto e'$ *then* $\cdot \vdash e' : 1$

For progress, instead of just stating there is a next step, we must consider all possible enclosing evaluation contexts that could enclose this term and show that there exists a next step for each one of them. This strengthens the inductive hypothesis enough to carry out the proof.

For preservation, we are given the additional assumption about the evaluation context having a valid type on the left, and we conclude that the right side is a closed program of type 1 in accordance with the three-part relation.

Combining these two theorems together gives us the type safety theorem, fulfilling the requirement expressed by Lemma 3. These theorems can be found in the Saftey.v file of the Rocq supplement.

The other requirement of determinism, Lemma 2, is fairly straightforward to prove for this language. This is proved in the Rules.v file of the Rocq supplement. This determinism

```
Inductive Of (A B f : term) (n : nat) :
    term -> term -> Prop :=
...
| Of_tm_fun : forall e1 e2 e3 p1 p2 p3,
  Of A B f n e1 p1 ->
  Of A B f n e2 p2 ->
  Of A B f n e3 p3 ->
  Of A B f n (tm_fun e1 e2 e3) (tm_fun p1 p2 p3)
...
| Of_pat_inf :
    Of A B f n (tm_fun A B f) pat
| Of_pat_fin : forall i, i >= n ->
    Of A B f n (unroll i A B f) pat
```

Figure 7 Select cases of the **of** judgment

lemma is only utilized for Lemma 4, which can be found in the Kleene.v file of the Rocq supplement.

6.3 Formalizing Residual Language

The remainder of the proofs covered in this paper can all be found within the Compactness.v file of the Rocq supplement.

The biggest design decision involved in creating the residual language associated with our chosen language is how to implement the open global variable ω. Do we want to handle substitution into ω in the same way as local variables but with a global scope or express it as a different syntactic construct?

We originally tried to express ω as the 0th variable per our De Bruijn indices interpretation of variables. The benefit of this is that we did not need to implement any new instrumentation to handle substitution into the ω variable as we could just utilize the same framework we use for local variables. The disadvantage is that we need to consider the index of the variable changing when going underneath binders within the term. As such, the **of** judgment needs to carry an extra counter indicating what variable index is associated with ω, and this counter changes within the inductive definition of the **of** judgment whenever entering into a bound scope. The substitution lemmas associated with this setup proved to be rather bulky as we essentially had to reprove the substitution lemma in a way that singled out the 0th variable, and it unnecessarily complicated the definition of the **of** judgment, although this method does work well with good enough support for variable bindings.

We opted instead to define the "pat" constant in our language to represent the variable ω. We still had to prove substitution-related theorems for this setting, but it was a lot simpler as we only had to consider a single isolated variable instead of needing to reindex De Bruijn indices.

Beyond the necessary substitution lemmas, we also needed to implement a lot of intermediate lemmas dealing with $\mathbf{of}^n$ including the following lemmas.

▶ **Lemma 15** (Compose $\mathbf{of}^n$). *If* E $\mathbf{of}^n$ P *and* e $\mathbf{of}^n$ p *then* $E[e]$ $\mathbf{of}^n$ $P[p]$.

▶ **Lemma 16** (Reflexivity for **of**).
$\Gamma \vdash \tau$ implies τ $\mathbf{of}^n$ τ

- $\Gamma \vdash e : \tau$ implies e $\mathbf{of}^n$ e
- $E : (\cdot \vdash \tau) \Rightarrow (\cdot \vdash 1)$ implies E $\mathbf{of}^n$ E

6.4 Compactness

In order to prove compactness with the extended three-part relation for continuations, several intermediate steps need to be extended to account for open evaluation contexts with variable ω.

We need to make the following modifications to Lemmas 8 and 10 by generalizing over their evaluation context.

▶ **Lemma 17** ($\longmapsto$ to $\longmapsto_\omega$ with Continuations). *If* $E[e] \downarrow$ *and* E $\mathbf{of}^0$ P *and* e $\mathbf{of}^0$ p *and* $E; e \longmapsto e'$ *then there exists* p' *such that* $P; p \longmapsto_\omega p'$ *and* e' $\mathbf{of}^0$ p'.

▶ **Lemma 18** ($\longmapsto_\omega$ to $\longmapsto$ with Continuations). *If* D $\mathbf{of}^{n+1}$ P *and* d $\mathbf{of}^{n+1}$ p *and* $P; p \longmapsto_\omega p'$ *then there exists* d' *such that* $D; d \longmapsto d'$ *and* d' $\mathbf{of}^n$ p'.

The overall proof strategy for proving these lemmas remains the same, we simply need to manage the overhead of evaluation contexts using Lemma 15 whenever needed.

Lemma 7 for relating values does not change at all. The multi-step Corollaries 9 and 11 and even the generalized compactness Lemma 12 also do not change since the multi-step relation for continuations is defined by chaining single steps of the form $[\cdot]; e \longmapsto e'$ for the empty evaluation context $[\cdot]$, and we can show that $[\cdot]$ $\mathbf{of}^n$ $[\cdot]$ by definition. The final desired lemma of compactness does not change significantly: we simply add evaluation contexts to the definition. This is where Lemma 16 comes into play by granting us that E $\mathbf{of}^0$ E.

▶ **Lemma 19** (Compactness with Continuations). $E[[F_\omega/x]e] \downarrow$ iff $\exists n.\ E[[F_n/x]e] \downarrow$.

As we can see, even when dealing with this modified setting with control flow effects, the overall proof strategy remains largely the same. Adding more features into the language even beyond continuations does not present a significant impact on the compactness lemma, as the general shape of execution remains the same upon replacing instances of the recursive function.

7 Related Work

The use of the compactness lemma has early roots in the literature as a means of converting domain-theoretic verification techniques of denotational semantics into the setting of operational semantics. Through personal correspondence, we learn that the lemma was first coined as the "compactness" property by Harper as early as 1995. Mason, Smith, and Talcott were the first to utilize the compactness lemma for operational semantics and proved the lemma by focusing on the specific one-step reduction relevant to the function approximation [13]. The name "compactness" was adopted by Pitts in his development of a proof strategy for it using cofinal sets [15] in the realm of natural semantics. Soon later, this strategy was adapted to an operational semantics setting by Birkedal and Harper [6].

The Pitts strategy involves formalizing a program context $C\{F_{n_1}, F_{n_2}, \cdots, F_{n_k}\}$ which takes in as input a vector that consists of various different levels of unrollings of the function in question. The strategy relies on bookkeeping all these unrollings and future expansions to the unrolling vector during the overall computation process.

The important thing to note is that Pitts suggests that the best way to facilitate compactness is to reason about the whole program during execution, and does this via

the program context. There is nothing in this proof that would prevent it from extending to the setting of non-local control flow, but the overhead cost of bookkeeping unrolling vectors becomes evident when considering formal verification. Our formalization also relies on reasoning about the whole program but does so by maintaining invariants over the whole program through the **of** judgment to simplify bookkeeping.

Around the same time, Crary devised a different strategy to prove compactness utilizing applicative approximation [7]. Given closed computations e_1 and e_2 of the same type, the applicative approximation $e_1 \preceq e_2$ means that if e_1 reduces to a value v_1, then e_2 must also reduce to a value v_2 where $v_1 \preceq_{val} v_2$ for some type-specific definition of $\preceq_{val}$. With this, it is easy to see that $\bot$ approximates everything, so by extension, lower depths of unrollings approximate higher depths of unrollings.

Crary simulates any chain of reductions $[F_\omega/x]e_1 \longmapsto \cdots \longmapsto [F_\omega/x]e_n$ with a related chain of applicative approximations: $[F_k/x]e_1 \succeq \cdots \succeq [F_{k-j}/x]e_n$. By the definition of applicative approximation, $[F_{k-j}/x]e_n \downarrow$ implies that $[F_k/x]e_1 \downarrow$, which is sufficient to establish compactness. The chain's use of approximation instead of evaluation allows it to maintain a consistent level of approximation, since if an appearance of x is filled by F truncated too little, truncating that appearance more will result in an approximation.

Unfortunately, applicative approximation only works for negative connectives and does not generalize to positive connectives. To handle positive connectives, Crary extends the approach utilizing contextual approximation [8]. Unlike applicative approximation, it is not obvious that evaluation implies contextual approximation nor is it obvious that $\bot$ contextually approximates everything, so proving these theorems requires extra work. Once these are proven for the specific language in question, the proof of compactness proceeds in the same way through a simulation lemma.

A large disadvantage of this proof strategy through contextual approximation is that it is not immediately obvious how it can be extended to the case of non-local control flow. In expressing continuations through `letcc` and `throw` operations, it is no longer the case that evaluation implies contextual approximation: we have the single-step evaluation of $\texttt{throw}((); [\cdot]) \longmapsto ()$, but the context $\texttt{let}\ x = ([\cdot])\ \texttt{in}\ \bot$ distinguishes between these two terms. However, as shown by the Rocq formalization, the residual stepping approach can cleanly extend to the setting of non-local control flow as there is no reliance on contextual approximation.

7.1 Step-Indexed Logical Relations

An alternative to compactness for FTLR of logical relations is using step indexing to formalize the definition of logical relations. The logical relation is then not only inductively defined on the type structure but also on an extra counter based on the occurrences of effectful operations (such as non-termination) in computation. This counter strengthens the inductive hypothesis of FTLR, allowing us to ignore the issues presented by analyzing recursive functions.

The step-indexed model was first invented by Appel and McAllester [4], but was soon after adapted into the realm of logical relations for various settings by Ahmed and coworkers [1, 2, 3, 14].

However, using step indexing has many drawbacks as it makes the logical relation less elegant to use after proving FTLR, as instead of just using the type of the term, it forces us to keep track of an additional counter. This has more than just aesthetic consequences: step-indexed logical relations can only handle saftey properties, not liveness properties, indicating that they are inherently weaker than logical relations defined through the compactness lemma.

There are additionally several variations of step-indexed logical relations that avoid the traditional presentation by instead incorporating a future modality [10]. We will still categorize these as step-indexed logical relations because, while they greatly help make step-indexing more practical, they likewise cannot handle liveness properties.

7.2 Program Capsules

The residual stepping strategy was in part inspired by the idea of a program capsule [12] where closed programs feature the pair $\langle e, \sigma \rangle$ where e is an open term and σ is a substitution which provides meaning to the variables in e. In this setting, the open term e can continue executing normally as if it were closed by reading the values stored for each free variable on the substitution. This framework enables a nice separation between the computation components of a term e and the value components located in the substitution σ.

For our purposes, we only allow the one free variable ω, and the function F in question is closed and global, meaning the substitution σ is simply of the form $\omega \hookrightarrow F$. The only difference between the overall execution of $\langle e, \omega \hookrightarrow F_\omega \rangle$ and $\langle e, \omega \hookrightarrow F_i \rangle$ is found within the substitution σ, therefore we can isolate out the substitution to enable an easy comparison of through the execution of e, hence residual stepping.

8 Conclusion

As Pitts claimed, the best way to deal with the compactness lemma is by analyzing the program as a whole during its execution. The merit of this paper is not so much the ideas surrounding compactness as we draw upon many years of previously known insight, but the novelty is in the proof structure, the simplification of bookkeeping, and generalizability to create a proof pearl of the compactness lemma.

When comparing $[F_\omega/x]e$ to $[F_n/x]e$, what we truly care about are the computable components as described by the residual program e, abstracting over the specific depth of the unrolling. The residual language helps us describe precisely that, and the **of** relation further provides all the structure needed for bookkeeping the unrollings in a formal verification setting.

When translating these proofs into a proof assistant like Rocq, several implicit details that could have been hand-waved in paper proofs become non-trivial, formally verified proofs. Unlike some of the earliest work on compactness, this proof strategy was developed with formal verification in mind, and thus the overall proof strategy is clean in both the paper version of lemmas and the machinery utilized to formally verify them.

In future work, we may want to consider the compactness lemma under the condition that the function F in question is an open term. We suspect that the full generality of program capsules with properly scoped substitutions are needed for this case rather than the specified residual language with a globally scoped ω representing function F.

References

1 Umut A Acar, Amal Ahmed, and Matthias Blume. Imperative self-adjusting computation. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 309–322, 2008.

2 Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *European Symposium on Programming*, pages 69–83. Springer, 2006.

3 Amal Ahmed, Derek Dreyer, and Andreas Rossberg. State-dependent representation independence. *ACM SIGPLAN Notices*, 44(1):340–353, 2009.

4 Andrew W Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(5):657–683, 2001.

5 Hans Bekić. Definable operations in general algebras, and the theory of automata and flowcharts. *Programming Languages and Their Definition: H. Bekič (1936–1982)*, pages 30–55, 2005.

6 Lars Birkedal and Robert Harper. Relational interpretations of recursive types in an operational setting. *Information and computation*, 155(1-2):3–63, 1999.

7 Karl Crary. *Type-theoretic methodology for practical programming languages*. Cornell University, 1998.

8 Karl Crary. Fully abstract module compilation. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

9 Karl Crary and Robert Harper. Syntactic logical relations for polymorphic and recursive types. *Electronic notes in theoretical computer science*, 172:259–299, 2007.

10 Derek Dreyer, Amal Ahmed, and Lars Birkedal. Logical step-indexed logical relations. *Logical Methods in Computer Science*, 7, 2011.

11 Robert Harper. *Practical foundations for programming languages*. Cambridge University Press, 2016.

12 Jean-Baptiste Jeannin and Dexter Kozen. Computing with capsules. In *Descriptional Complexity of Formal Systems: 14th International Workshop, DCFS 2012, Braga, Portugal, July 23-25, 2012. Proceedings 14*, pages 1–19. Springer, 2012.

13 Ian A Mason, Scott F Smith, and Carolyn L Talcott. From operational semantics to domain theory. *Information and Computation*, 128(1):26–47, 1996.

14 Georg Neis, Derek Dreyer, and Andreas Rossberg. Non-parametric parametricity. *ACM Sigplan Notices*, 44(9):135–148, 2009.

15 Andrew M Pitts. Operationally-based theories of program equivalence. *Semantics and Logics of Computation*, 14:241, 1997.

16 Andrew M Pitts. Typed operational reasoning. *Advanced Topics in Types and Programming Languages*, pages 245–289, 2005.