The Power of Parameterization in Coinductive Proof

Chung-Kil Hur
Microsoft Research
gil@microsoft.com

Georg Neis
MPI-SWS & Saarland University
neis@mpi-sws.org

Derek Dreyer
MPI-SWS
dreyer@mpi-sws.org

Viktor Vafeiadis
MPI-SWS
viktor@mpi-sws.org

Abstract

Coinduction is one of the most basic concepts in computer science. It is therefore surprising that the commonly-known lattice-theoretic accounts of the principles underlying coinductive proofs are lacking in two key respects: they do not support *compositional* reasoning (*i.e.,* breaking proofs into separate pieces that can be developed in isolation), and they do not support *incremental* reasoning (*i.e.,* developing proofs interactively by starting from the goal and generalizing the coinduction hypothesis repeatedly as necessary).

In this paper, we show how to support coinductive proofs that are both compositional and incremental, using a dead simple construction we call the *parameterized greatest fixed point*. The basic idea is to parameterize the greatest fixed point of interest over the *accumulated knowledge* of "the proof so far". While this idea has been proposed before, by Winskel in 1989 and by Moss in 2001, neither of the previous accounts suggests its general applicability to improving the state of the art in interactive coinductive proof.

In addition to presenting the lattice-theoretic foundations of parameterized coinduction, demonstrating its utility on representative examples, and studying its composition with "up-to" techniques, we also explore its mechanization in proof assistants like Coq and Isabelle. Unlike traditional approaches to mechanizing coinduction (*e.g.,* Coq's `cofix`), which employ *syntactic* "guardedness checking", parameterized coinduction offers a *semantic* account of guardedness. This leads to faster and more robust proof development, as we demonstrate using our new Coq library, *Paco*.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic

Keywords Coinduction, simulation, parameterized greatest fixed point, compositionality, lattice theory, interactive theorem proving

1. Introduction

Coinduction is one of the most basic concepts in computer science. Coinductive proofs, especially those based on *simulation* arguments, are relevant in many settings where one wishes to model infinitary properties or recursive behaviors [13, 18]. It is therefore surprising that the commonly-known lattice-theoretic accounts of the principles underlying coinductive proofs are lacking in two key respects: they do not support *compositional* reasoning (*i.e.,* breaking proofs into separate pieces that can be developed in isolation), and they do not support *incremental* reasoning (*i.e.,* developing proofs interactively by starting from the goal and generalizing the coinduction hypothesis repeatedly as necessary).

The Trouble with Compositionality. Consider, for instance, two possibly non-terminating, mutually recursive but loosely coupled functions, f and g, which we would like to show equivalent to another pair of recursive functions, f' and g'. Ideally, we should be able to reason about each function separately. That is, we should be able to prove that $f \simeq f'$ entails $g \simeq g'$, and similarly that $g \simeq g'$ entails $f \simeq f'$, and somehow derive from those two entailments that $(f, g) \simeq (f', g')$. The problem, however, is that this type of circular reasoning (aka "rely-guarantee"-style reasoning) is unsound in general. For example, if $f = g$ and $f' = g'$, then we would be able to derive $f \simeq f'$ from a tautology.

One way to avoid this kind of unsound circularity is by placing a syntactic *guardedness* restriction on the proofs of each of the entailments, ensuring that their hypotheses are only used after the definitions of the functions in their conclusions have been unfolded. This is the approach taken, for instance, by Coq's `cofix` tactic for coinduction [3]. Unfortunately, the limitation of using a *syntactic* criterion is that it is inherently non-compositional: it requires one to have access to the *proof* of each of the component entailments in order to determine whether the whole coinductive argument is valid. Moreover, as we explain in more detail in Section 7.3, the syntactic nature of guardedness checking makes proof checking severely inefficient and interacts poorly with other tactics. What we would really like instead is a more *semantic* account of guardedness checking, by which the guardedness condition is reflected directly in the statement of the entailment being proved, rather than being relegated to a syntactic property of the proof itself.

The Desire for Incrementality. Consider the transition system shown in Figure 1, and suppose we want to show that there is an infinite path starting from node a.

Let us first try to imitate a model checker and explore all paths starting from a in a depth-first fashion. For the first step, there is only one choice: the edge $a \to b$. Then at b, we have two choices. Perhaps we can try $b \to c$, but this will soon lead to a dead end, at which point we will have to backtrack. So let us follow the edge $b \to d$ instead. Then, we follow $d \to b$ and we are back to a node we have already visited. We have discovered a cycle, and thus an infinite path, reachable from a.

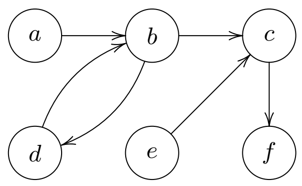
Now let us try to do the same proof formally. The set of nodes from which infinite paths emanate can be defined as the greatest fixed point $inf \overset{\text{def}}{=} \nu\, step$ of the following monotone function:

$$step(X) \overset{\text{def}}{=} \{x \in \mathsf{Node} \mid \exists y \in X.\ x \to y\}$$

Our goal then is to show that $a \in inf$. Of course, in this small example, we could just compute inf directly by iteration and then

This research was carried out primarily while the first author was a post-doctoral researcher at MPI-SWS. The second author is currently funded by a Google European Doctoral Fellowship.

POPL'13, January 23–25, 2013, Rome, Italy.
Copyright © 2013 ACM 978-1-4503-1832-7/13/01…$10.00



Node($=x$)	$\mathrm{G}_{step}(\{x\})$
a	$\{a, b, d\}$
b	$\{a, b, d\}$
c	$\{a, b, d, e\}$
d	$\{a, b, d\}$
e	$\{a, b, d\}$
f	$\{a, b, c, d, e\}$

Figure 1. A simple transition system and tabulation of G_{step}.

check if a is in it, but suppose we do not want to do this because in practice the transition system may be huge or even infinite.

Instead, we may employ Tarski's fixed-point theorem [22], which says that, to show $a \in inf$, it suffices to find a set of nodes X such that $a \in X$ and $\forall x \in X.\ \exists y \in X.\ x \to y$. For the given transition system, a possible such set is $X = \{a, b, d\}$, which corresponds to the set of nodes that we followed to exhibit the cycle earlier. The problem, however, is that this proof is rather different from the "model checking" one, and actually more difficult because it forces us to figure out what X is up front. What we would really like is a way to prove our goal by incrementally expanding the coinduction hypothesis from $\{a\}$ to $\{a, b\}$ to $\{a, b, d\}$ as we explore the transition system and see what nodes are reachable from a. The validity of such an approach is intuitively obvious, but what is the general lattice-theoretic proof principle that justifies it?

Contributions. In this paper, we show how to support coinductive proofs that are *both compositional and incremental*, using a dead simple construction we call the *parameterized greatest fixed point*. The basic idea is to parameterize the greatest fixed point of interest over the *accumulated knowledge* of "the proof so far".

Neither the idea nor the construction behind it is an original invention of ours *per se*. In 1989, Winskel [23] proposed the same idea for supporting "local model checking" in the modal μ-calculus. (His construction, which is slightly different from ours, supports incrementality but not compositionality—in our sense of the word—but it is straightforward to repurpose his core "reduction lemma" to derive a compositional version of his construction.) Independently, in 2001, Moss [14] presented a construction that *is* essentially the same as ours, albeit in a more abstract categorical setting. However, neither of these prior accounts suggests the general applicability of the parameterized greatest fixed point to improving the state of the art in interactive coinductive proof.

Our goal in the present paper is to popularize the idea of parameterized coinduction and explore its potential as a practically useful tool. More specifically, we make the following contributions:

- We present the parameterized greatest fixed point in simple lattice-theoretic terms, and show that it validates several useful principles for compositional, incremental proofs (Section 2). We give representative examples to illustrate the utility of these proof principles (Sections 2 and 3).
- We show how parameterized coinduction is complementary to the traditional approach to simplifying simulation proofs via *"up-to" techniques*, and we develop the basic theory of how these approaches compose (Section 4).
- We explore the issues that arise in the *mechanization* of parameterized coinduction in existing interactive theorem provers like Coq and Isabelle (Section 5). Fortunately, several of these issues can be resolved through variations on a somewhat esoteric technique called *Mendler-style recursion* (Section 6).
- We describe *Paco* (pronounced "pah-ko"), a new Coq library we have developed for parameterized coinduction. Compared to Coq's existing `cofix` tactic, Paco enables faster and more robust proof development, thanks to its support for *semantic*, rather than syntactic, guardedness checking (Section 7).

Finally, we conclude the paper in Section 8 with a detailed discussion of related work.

The technical development of this paper has been formalized in the Coq proof assistant. That formalization, together with a tutorial for our Coq library, Paco, is available from the Paco website:

`http://plv.mpi-sws.org/paco/`

2. Parameterized Coinduction

Let us begin by reviewing the basic lattice theory underlying coinductive definitions and their associated standard proof principles.

Consider a complete lattice $(C, \sqsubseteq, \sqcap, \sqcup, \top, \bot)$, and a monotone (*i.e.,* order-preserving) function $f \in C \xrightarrow{\text{mon}} C$. Strictly speaking, for generality, we do not require $\sqsubseteq$ to be antisymmetric, and we write $\equiv$ for the intersection of $\sqsubseteq$ and $\sqsupseteq$ (its inverse), which corresponds to $=$ if we have antisymmetry. We say that r is a *prefixed point* of f if $f(r) \sqsubseteq r$, and r is a *postfixed point* of f if $r \sqsubseteq f(r)$. Further, we write μf for f's least fixed point and νf for its greatest fixed point, which by Tarski's fixed-point theorem [22] is equal to the join of all postfixed points of f:

$$\nu f \ \equiv\ \bigsqcup\{r \in C \mid r \sqsubseteq f(r)\}$$

Tarski's Principle. We are concerned with proving statements of the form $x \sqsubseteq \nu f$. From Tarski's theorem we directly get that postfixed points are included in the greatest fixed point:

$$x \sqsubseteq f(x) \implies x \sqsubseteq \nu f \qquad\qquad (\textsc{Tarski})$$

To prove that $x \sqsubseteq \nu f$ when $x \not\sqsubseteq f(x)$ using this principle, we have to determine a postfixed point of f larger than x up front:

$$x \sqsubseteq \nu f \iff \exists r.\ x \sqsubseteq r \wedge r \sqsubseteq f(r)$$

This is clearly inconvenient for doing interactive proofs, as it forces one to construct the coinduction hypothesis r up front, instead of allowing r to be generated naturally in the course of the proof. Recall that in the example of the introduction, although we were only interested in showing that $a \in \nu\, step$, we had to pick $r := \{a, b, d\}$ up front. In large proofs, this quickly becomes a big problem. For example, the ClightTSO-Csharpminor simulation proof in the CompCertTSO verified compiler [20] requires a simulation relation r that comprises 69 cases, most of which tediously relate intermediate execution states.

Strong Coinduction. Second, there is a slight variant of (Tarski), sometimes called the *strong coinduction principle* [4]:

Lemma 1 (Strong coinduction)**.** $x \sqsubseteq \nu f \iff x \sqsubseteq f(x \sqcup \nu f)$.

Proof. First, we have $\nu f \equiv f(\nu f) \sqsubseteq f(x \sqcup \nu f)$ (†). The ($\Longrightarrow$) direction follows directly from (†). For the ($\Longleftarrow$) direction: from $x \sqsubseteq f(x \sqcup \nu f)$ and (†), we get $x \sqcup \nu f \sqsubseteq f(x \sqcup \nu f)$, *i.e.,* $x \sqcup \nu f$ is a postfixed point of f. So, from (Tarski), $x \sqsubseteq x \sqcup \nu f \sqsubseteq \nu f$. □

This principle is "strong" in the sense that it is complete, but it still does not offer us a very useful interactive proof technique. The problem arises if in the course of proving that $x \sqsubseteq f(x \sqcup \nu f)$, we ever need to generalize the coinduction hypothesis by adding some y to it. The only recourse the strong coinduction principle gives us at this point (if we want to continue interactively with the proof) is to show that $y \sqsubseteq \nu f$. But of course the proof of that may cycle around, forcing us to prove that $x \sqsubseteq \nu f$, in which case we are stuck. We are therefore forced to restart the proof, generalizing the coinduction hypothesis to $x \sqcup y$, *i.e.,* showing that $x \sqcup y \sqsubseteq \nu f$.

Parameterized Coinduction. Our *parameterized coinduction* principle gives us a way to avoid restarting the proof by making explicit the idea of *accumulated knowledge*. In the course of the proof, we remember the things that we have already claimed are $\sqsubseteq \nu f$ and we can treat those as assumed knowledge in *guarded* subproofs (where guardedness is enforced semantically, not syntactically).

Formally, the idea is that instead of dealing with νf directly, we deal instead with some G_f that is parameterized by accumulated knowledge. That is, $\mathrm{G}_f \in C \xrightarrow{\text{mon}} C$, and intuitively, $\mathrm{G}_f(x)$ represents our "goal" of proving that something is in νf, under accumulated (or assumed) knowledge x.

Definition 1 (Parameterized greatest fixed point)**.**
We define $\mathrm{G} \in (C \xrightarrow{\text{mon}} C) \xrightarrow{\text{mon}} (C \xrightarrow{\text{mon}} C)$:

$$\mathrm{G}_f(x) \overset{\text{def}}{=} \nu y.\ f(x \sqcup y)$$

Here, and elsewhere, we write $\nu y.a$ for $\nu(\lambda y.a)$. Note that the fixed point in the definition exists because $\lambda y.\ f(x \sqcup y)$ is monotone for monotone f. The monotonicity of G and G_f is easy to check.

One way of understanding G is pictorially. In the transition system from the introduction, $\mathrm{G}_{step}(X)$ describes the set of nodes that either have an infinite path (*i.e.,* are in $\nu\ step$), or else have a *non-empty* path leading to a node in X.[1] To illustrate, Figure 1 lists the set $\mathrm{G}_{step}(\{x\})$ for each node x.

Incrementality of Parameterized Coinduction. We begin with the trivial observation that the parameterized greatest fixed point coincides with the standard one if no knowledge has been accumulated.

Lemma 2 (Initialize)**.** $\nu f \equiv \mathrm{G}_f(\bot)$.

Further, by simply unfolding the fixed point, we obtain the following analogue of the strong coinduction principle mentioned previously (Lemma 1), except that this version can be stated directly as an equality on the parameterized greatest fixed point:

Lemma 3 (Unfold)**.** $\mathrm{G}_f(x) \equiv f(x \sqcup \mathrm{G}_f(x))$.

What G_f allows us to do in addition, that νf does not support, is to accumulate knowledge in the following sense:

Theorem 4 (Accumulate)**.** $y \sqsubseteq \mathrm{G}_f(x) \iff y \sqsubseteq \mathrm{G}_f(x \sqcup y)$.

Proof. The $(\Longrightarrow)$ direction follows straight from the monotonicity of G_f. In the $(\Longleftarrow)$ direction, assume $y \sqsubseteq \mathrm{G}_f(x \sqcup y)$ (*). Then,

$$\begin{array}{rll}
\mathrm{G}_f(x \sqcup y) & \equiv f(x \sqcup y \sqcup \mathrm{G}_f(x \sqcup y)) & \text{fixed point equation} \\
& \sqsubseteq f(x \sqcup \mathrm{G}_f(x \sqcup y)) & f \text{ monotone and (*)} \\
& \equiv (\lambda z.\ f(x \sqcup z))\,(\mathrm{G}_f(x \sqcup y)) &
\end{array}$$

Therefore, as $\mathrm{G}_f(x \sqcup y)$ is a postfixed point of $\lambda y.f(x \sqcup y)$, we obtain from (TARSKI) that $\mathrm{G}_f(x \sqcup y) \sqsubseteq \mathrm{G}_f(x)$, which together with (*) entails $y \sqsubseteq \mathrm{G}_f(x)$, as required. □

Compositionality of Parameterized Coinduction. Our construction also admits a clean compositional rule for combining proofs in the circular rely-guarantee style [8]:

$$\frac{\begin{array}{ll} g_1 \sqsubseteq \mathrm{G}_f(r_1) & r_1 \sqsubseteq r \sqcup g_2 \\ g_2 \sqsubseteq \mathrm{G}_f(r_2) & r_2 \sqsubseteq r \sqcup g_1 \end{array}}{g_1 \sqcup g_2 \sqsubseteq \mathrm{G}_f(r)} \qquad (\text{COMPOSE})$$

The rule says that we can prove g_1 and g_2 are "correct" under assumptions r by proving that g_1 is correct under the additional assumption that g_2 is correct and similarly that g_2 is correct under the additional assumption that g_1 is. In essence, this rule is sound because $\mathrm{G}_f(r)$ allows the use of the assumptions r only within a guarded context.

Although we have motivated the desire for compositionality separately from the desire for incrementality, it turns out that this (COMPOSE) rule is equivalent to the accumulation theorem (Theorem 4), which we have already proved, and in fact the two principles are *interderivable* under no assumptions about how G_f is defined (aside from the fact that it is monotone).

To see this, let us first derive (COMPOSE) from Theorem 4. Assuming the premises of (COMPOSE) hold, it is clear by monotonicity of G_f that $g_1 \sqsubseteq \mathrm{G}_f(r \sqcup g_1 \sqcup g_2)$ and $g_2 \sqsubseteq \mathrm{G}_f(r \sqcup g_1 \sqcup g_2)$, so $g_1 \sqcup g_2 \sqsubseteq \mathrm{G}_f(r \sqcup g_1 \sqcup g_2)$. By Theorem 4, $g_1 \sqcup g_2 \sqsubseteq \mathrm{G}_f(r)$ as desired. Conversely, we can derive Theorem 4 from (COMPOSE). As before, the $(\Longrightarrow)$ direction follows straight from the monotonicity of G_f. In the $(\Longleftarrow)$ direction, assume $y \sqsubseteq \mathrm{G}_f(x \sqcup y)$. Then, instantiate (COMPOSE) with $g_1 = g_2 = y$, $r = x$, and $r_1 = r_2 = x \sqcup y$. The conclusion yields $y \equiv y \sqcup y \sqsubseteq \mathrm{G}_f(x)$ as desired.

A Simple Application of Parameterized Coinduction. We now return to the "model checking" example presented in the introduction, and show how to use parameterized coinduction to prove $a \in inf$ incrementally. Here, we instantiate the principle with the powerset lattice on states of the transition system. Thus, showing $x \in S$ is equivalent to showing $\{x\} \sqsubseteq S$ in this lattice.

$$\begin{array}{lll}
& a \in inf = \nu\ step & \\
\Longleftrightarrow & a \in \mathrm{G}_{step}(\emptyset) & \text{initialize} \\
\Longleftrightarrow & \exists y \in \mathrm{G}_{step}(\emptyset).\ a \to y & \text{unfold} \\
\Longleftarrow & b \in \mathrm{G}_{step}(\emptyset) & \text{pick } y := b \\
\Longleftrightarrow & b \in \mathrm{G}_{step}(\{b\}) & \text{accumulate} \\
\Longleftrightarrow & \exists y \in \{b\} \cup \mathrm{G}_{step}(\{b\}).\ b \to y & \text{unfold} \\
\Longleftarrow & d \in \{b\} \cup \mathrm{G}_{step}(\{b\}) & \text{pick } y := d \\
\Longleftrightarrow & d \in \mathrm{G}_{step}(\{b\}) & \text{since } d \neq b \\
\Longleftrightarrow & \exists y \in \{b\} \cup \mathrm{G}_{step}(\{b\}).\ d \to y & \text{unfold} \\
\Longleftarrow & b \in \{b\} \cup \mathrm{G}_{step}(\{b\}) & \text{pick } y := b \qquad \square
\end{array}$$

As you can see, we can perform the same incremental proof as when "model checking" the example. Note that in the proof we do not necessarily have to accumulate the visited nodes at every step, but rather only when we think that their addition to the accumulated knowledge will be useful in the remainder of the proof.

With the same example, we can also illustrate a simple use of compositionality. We can easily establish:

$$\begin{array}{ll}
b \in \mathrm{G}_{step}(\{d\}) & \text{by unfolding \& picking } y := d \\
d \in \mathrm{G}_{step}(\{b\}) & \text{by unfolding \& picking } y := b
\end{array}$$

Thus, by (COMPOSE), $\{b, d\} \subseteq \mathrm{G}_{step}(\emptyset) = inf$.

Full Characterization of G_f**.** Finally, we observe that Lemma 3 and Theorem 4 uniquely determine G_f up to $\equiv$.

Proposition 5. For any G' such that (i) $\forall x.\ \mathrm{G}'(x) \equiv f(x \sqcup \mathrm{G}'(x))$ and (ii) $\forall x, y.\ y \sqsubseteq \mathrm{G}'(x \sqcup y) \implies y \sqsubseteq \mathrm{G}'(x)$, we have $\mathrm{G}' \equiv \mathrm{G}_f$.

Proof. Given x, $\mathrm{G}'(x) \sqsubseteq \mathrm{G}_f(x)$ follows by (TARSKI) from assumption (i). To show $\mathrm{G}_f(x) \sqsubseteq \mathrm{G}'(x)$, it suffices by (ii) to show $\mathrm{G}_f(x) \sqsubseteq \mathrm{G}'(x \sqcup \mathrm{G}_f(x))$, which, after unfolding on the left and rewriting using (i) on the right, follows by monotonicity of f. □

3. A Simulation Example

In this section, we illustrate parameterized coinduction on a slightly larger example that demonstrates the practical motivation for a compositional, incremental coinduction principle.

Consider the two recursive programs f and g shown in Figure 2. These programs continually poll the user for a new (numerical) input, compute the double of the sum of all inputs seen so far,

[1] The fact that the paths to nodes in x must be non-empty is what ensures that x can be used as an assumption only within guarded contexts.

$$\begin{array}{l}
\mathsf{restart} \overset{\text{def}}{=} \mathsf{fix\,restart(f).}\ \lambda\mathsf{n}. \\
\qquad \mathsf{if\ n} > \underline{0}\ \mathsf{then\ (output\ n; restart\ f\ (n} - \underline{1}\mathsf{))\ else\ f}\ \underline{0} \\
\quad \mathsf{f} \overset{\text{def}}{=} \mathsf{fix\,f(n).} \\
\qquad \mathsf{let\ v} = \mathsf{(output\ n; input())} * \underline{2}\ \mathsf{in} \\
\qquad \mathsf{(if\ v} \neq \underline{0}\ \mathsf{then\ f\ else\ restart\ f)\ (v + n)} \\
\quad \mathsf{g} \overset{\text{def}}{=} \mathsf{fix\,g(m).} \\
\qquad \mathsf{output}\ (\underline{2} * \mathsf{m}); \\
\qquad \mathsf{let\ v} = \mathsf{input()\ in} \\
\qquad \mathsf{if\ v} = \underline{0}\ \mathsf{then\ restart\ g}\ (\underline{2} * \mathsf{m})\ \mathsf{else\ g\ (v + m)}
\end{array}$$

Figure 2. Recursive programs in the simulation example.

$$\begin{array}{ll}
v\ ::= \underline{n} \mid \mathsf{fix}\, f(x).\, e & \oplus ::= + \mid - \mid * \mid = \mid \neq \mid > \mid \geq \\
e\ ::= x \mid v \mid e_1 \oplus e_2 \mid e_1\ e_2 \mid \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \mathsf{input}() \mid \mathsf{output}\ e & \\
K ::= e \oplus \bullet \mid \bullet \oplus v \mid e\ \bullet \mid \bullet\ v \mid \mathsf{if}\ \bullet\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \mathsf{output}\ \bullet & \\
q\ ::= \tau \mid \mathsf{in}\ n \mid \mathsf{out}\ n &
\end{array}$$

$$\begin{array}{rcll}
\lambda x.\, e & \overset{\text{def}}{=} & \mathsf{fix}\, f(x).\, e & \text{where } f \notin \text{fv}(e) \\
\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 & \overset{\text{def}}{=} & (\lambda x.\, e_2)\ e_1 & \\
e_1; e_2 & \overset{\text{def}}{=} & \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 & \text{where } x \notin \text{fv}(e_2)
\end{array}$$

$$\begin{array}{rcll}
\underline{n_1} \oplus \underline{n_2} & \rightarrow & \underline{n_1 \oplus n_2} & \\
\mathsf{if}\ \underline{n}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 & \rightarrow & e_1 & \text{where } n \neq 0 \\
\mathsf{if}\ \underline{0}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 & \rightarrow & e_2 & \\
(\mathsf{fix}\, f(x).\, e)\ v & \rightarrow & e[(\mathsf{fix}\, f(x).\, e)/f, v/x] & \\
\mathsf{input}() & \xrightarrow{\mathsf{in}\ n} & \underline{n} & \\
\mathsf{output}\ \underline{n} & \xrightarrow{\mathsf{out}\ n} & \underline{0} & \\
K[e] & \xrightarrow{q} & K[e'] & \text{where } e \xrightarrow{q} e'
\end{array}$$

Figure 3. Syntax and semantics of a tiny programming language.

and report the current value of the sum. If at any point, the user provides zero as an input, then the programs count down to zero, and start again. The two programs differ in the representation of the double of the sum, as well as in their programming style (as the programmer responsible for f followed a somewhat peculiar coding style). Nevertheless, it should be relatively straightforward to see that $\mathsf{g}(\underline{m})$ is equivalent to $\mathsf{f}(\underline{2m})$.

Formally, these programs are written in the minimal programming language defined in Figure 3. This is a standard call-by-value λ-calculus with integers, recursion, primitives for inputting and outputting integer values, and a right-to-left evaluation order for applications and arithmetic operations. We give its operational semantics as a labelled transition system, $e \xrightarrow{q} e'$, with the labels recording numerical inputs and outputs. We follow the standard convention of writing $\rightarrow$ instead of $\xrightarrow{\tau}$ for internal transitions.

For this language, we can define (weak) similarity as the greatest fixed point of the following function on expression relations:

$$\begin{array}{l}
\mathit{sim}(R) \overset{\text{def}}{=} \{(e_1, e_2) \mid \\
\quad (\forall v.\ e_1 = v \implies e_2 \overset{\tau}{\rightsquigarrow} v) \\
\quad \wedge\ (\forall q, e_1'.\ e_1 \xrightarrow{q} e_1' \implies \exists e_2'.\ e_2 \overset{q}{\rightsquigarrow} e_2' \wedge R(e_1', e_2'))\}
\end{array}$$

where $\overset{\tau}{\rightsquigarrow}$ is equal to $\rightarrow^*$ (the reflexive-transitive closure of $\rightarrow$), and $\overset{q}{\rightsquigarrow}$ is equal to $\rightarrow^* \xrightarrow{q}$ for $q \neq \tau$. If e_1 is a value, we require e_2 to evaluate to the same value; otherwise, if e_1 reduces to some e_1', we require that e_2 can match that execution step and end up in a R-related state.[2] Note that we are working with the powerset lattice $\mathcal{P}(\mathsf{exp} \times \mathsf{exp})$ where $\sqsubseteq\ =\ \subseteq$, $\sqcup = \cup$, $\sqcap = \cap$, and $\bot = \emptyset$.

Our goal is to show that $\mathsf{g}(\underline{m})$ simulates $\mathsf{f}(\underline{2m})$, that is:

$$R_0 := \{(\mathsf{f}(\underline{2m}), \mathsf{g}(\underline{m})) \mid m \in \mathbb{Z}\} \subseteq \nu \mathit{sim}$$

While this might appear trivial, proving this formally using Tarski's coinduction principle is extemely laborious. We have to come up with a "simulation relation" R containing all the intermediate execution steps of f and g appropriately matched: just before the output, just after the output, just before the input, just after the input, just before the evaluation of the condition, just after the evaluation of the condition, just after the choice of the condition, and so on. In total, there are 16 cases. This relation must be fully defined before we can even start the proof.

We remark that up-to techniques [18, 17], the standard toolbox for simplifying simulation proofs, while very helpful in many cases, are of limited use in this example. As the programs strictly alternate internal and external execution steps, the "up to reduction" technique can reduce the cases only by a factor of two. Similarly, the "up to context" technique does not help us relate the intermediate states of the calls to f and g because the two functions have rather different internal structure. (It does, however, help in reasoning about restart, as we will see in Section 4.)

3.1 Proof Sketch using Parameterized Coinduction

We now show how to apply parameterized coinduction in order to avoid the explicit, manual generalization of the coinduction hypothesis that is required with Tarski's principle. First, let us introduce the following shorthand:

$$\begin{array}{r}
\mathit{ff}(v) \overset{\text{def}}{=} (\mathsf{if}\ v \neq \underline{0}\ \mathsf{then\ f\ else\ restart\ f})\ (v + \underline{2m}) \\
\mathit{gg}(v) \overset{\text{def}}{=} \mathsf{if}\ v = \underline{0}\ \mathsf{then\ restart\ g}\ (\underline{2} * \underline{m})\ \mathsf{else\ g}(v + \underline{m})
\end{array}$$

We will now prove $R_0 \subseteq \mathrm{G}_{\mathit{sim}}(\emptyset)$, which by Lemma 2, $\equiv \nu \mathit{sim}$.
Step 1. By applying Theorem 4 followed by Lemma 3, we get as our goal:

$$R_0 \subseteq \mathit{sim}(\emptyset \cup R_0 \cup \mathrm{G}_{\mathit{sim}}(\emptyset \cup R_0)) = \mathit{sim}(R_0 \cup \mathrm{G}_{\mathit{sim}}(R_0))$$

This reduces to showing

$$\exists e_2'.\ \mathsf{g}(\underline{m}) \rightarrow^* e_2' \wedge (e_1', e_2') \in R_0 \cup \mathrm{G}_{\mathit{sim}}(R_0)$$

where $e_1' = (\mathsf{let\ v} = (\mathsf{output}\ \underline{2m};\ \mathsf{input}()) * \underline{2}\ \mathsf{in}\ \mathit{ff}(\mathsf{v}))$.
We pick $e_2' = (\mathsf{output}\ \underline{2m};\ \mathsf{let\ v} = \mathsf{input}()\ \mathsf{in}\ \mathit{gg}(\mathsf{v}))$ and proceed to show $(e_1', e_2') \in \mathrm{G}_{\mathit{sim}}(R_0)$.
Step 2. We now unfold the fixed point (Lemma 3) to get as our goal $(e_1', e_2') \in \mathit{sim}(R_0 \cup \mathrm{G}_{\mathit{sim}}(R_0))$, which means showing

$$\exists e_2''.\ e_2' \rightarrow^* \xrightarrow{\mathsf{out}\ 2m} e_2'' \wedge (e_1'', e_2'') \in R_0 \cup \mathrm{G}_{\mathit{sim}}(R_0)$$

where $e_1'' = (\mathsf{let\ v} = (\underline{0};\ \mathsf{input}()) * \underline{2}\ \mathsf{in}\ \mathit{ff}(\mathsf{v}))$. Now, we pick $e_2'' = (\underline{0};\ \mathsf{let\ v} = \mathsf{input}()\ \mathsf{in}\ \mathit{gg}(\mathsf{v}))$ and proceed to show $(e_1'', e_2'') \in \mathrm{G}_{\mathit{sim}}(R_0)$.
Next Steps. Further steps using Lemma 3 eventually lead us to a point where we have to prove the following two inclusions:

$$(\mathsf{f}(\underline{2v + 2m}), \mathsf{g}(\underline{v + m})) \in R_0 \cup \mathrm{G}_{\mathit{sim}}(R_0)$$
$$(\mathsf{restart\ f}\ \underline{2m},\ \mathsf{restart\ g}\ \underline{2m}) \in R_0 \cup \mathrm{G}_{\mathit{sim}}(R_0).$$

Regarding the former, we observe that the terms are related by R_0 and so we are done. Regarding the latter, we proceed as follows to show that they are related by $\mathrm{G}_{\mathit{sim}}(R_0)$.
Accumulation Step. We realize that (i) we have to increase the knowledge R_0 because restart calls itself recursively, and (ii) that before doing so we should generalize the goal to:

$$R_1 := \{(\mathsf{restart\ f}\ \underline{n},\ \mathsf{restart\ g}\ \underline{n}) \mid n \in \mathbb{Z}\} \subseteq \mathrm{G}_{\mathit{sim}}(R_0)$$

To this, we now apply the accumulating principle (Theorem 4) and get as our new goal:

$$R_1 \subseteq \mathrm{G}_{\mathit{sim}}(R_0 \cup R_1)$$

[2] We chose this simulation definition for simplicity. We can also handle more elaborate definitions that result in coarser equivalences for values of function type.

Final Steps. We proceed in the same manner as earlier, using Lemma 3 to step through the code of $\mathsf{restart}$. When arriving at the recursive call in the then-branch, we use the new R_1 part of the coinduction hypothesis and are done. When reasoning about the else-branch, on the other hand, we conclude by appeal to the original R_0 part, which fortunately is still around. □

The benefit of our approach over the traditional Tarski approach here is that the simulation relation does not have to be defined up front. Instead, the intermediate goals and the quantifier instantiations can be generated automatically by an interactive theorem prover using simple tactics. The details for achieving this will be presented in Section 5.

3.2 Decomposing the Proof

Since parameterized coinduction also supports compositionality, we can factor out the reasoning about $\mathsf{restart}$ from the previous proof into a separate generic lemma that may then be reused in other proofs.

Lemma 6 (Restart)**.** For all values f_1 and f_2, we have:

$$\{(\mathsf{restart}\ f_1\ \underline{n},\ \mathsf{restart}\ f_2\ \underline{n}) \mid n \in \mathbb{Z}\} \subseteq \mathrm{G}_{sim}(\{(f_1\ \underline{0}, f_2\ \underline{0})\})$$

Its proof follows straightforwardly from the fact that $\mathsf{restart}$ applies its function argument only to $\underline{0}$.

Instead of proving the previous goal $R_0 \subseteq \mathrm{G}_{sim}(\emptyset)$ directly, it now suffices to prove the following:

Lemma 7. $R_0 \subseteq \mathrm{G}_{sim}(R_1)$

Then $R_0 \subseteq \mathrm{G}_{sim}(\emptyset)$ is obtained from Lemmas 6 and 7 by rule (COMPOSE) (using empty initial assumptions). The proof of Lemma 7 follows the same structure as the one sketched for $R_0 \subseteq \mathrm{G}_{sim}(\emptyset)$, except that we are done after performing what were called above the "Next Steps".

Remark. Without G_{sim} we cannot split the proof into separate lemmas about $\mathsf{restart}$ and about f and g. For example, while we can prove the following statements:

$$R_0 \subseteq \nu\, sim \implies R_1 \subseteq \nu\, sim$$
$$R_1 \subseteq \nu\, sim \implies R_0 \subseteq \nu\, sim$$

we cannot combine the two to derive $R_0 \subseteq \nu\, sim$. This requires a sound form of circular reasoning, such as the one provided by the (COMPOSE) rule.

4. Combination with Up-To Techniques

The incremental proof sketched in the previous section, despite being a huge improvement over the one based on (TARSKI), is still quite tedious to do (at least on paper). Most of the proof involved trivally stepping through f and g. Doing so for f and g is arguably necessary, because their structure is quite different, but for the proof of Lemma 6 about $\mathsf{restart}$, where the structure is identical, it seems unnecessary. In this section, we will see that with a bit of additional theory, this tedium can be avoided as well.

In traditional (bi-)simulation proofs, people often employ simplification techniques known as *up-to functions* [19, 17]. Intuitively, an up-to function maps an element $r \in C$ to an element r^* (typically larger than r) that is still valid as a coinduction hypothesis for proving $r \sqsubseteq \nu f$.

Definition 2. $(-)^* \in C \xrightarrow{\text{mon}} C$ is a *sound up-to function* for $f \in C \xrightarrow{\text{mon}} C$ iff $r \sqsubseteq f(r^*)$ implies $r \sqsubseteq \nu f$ for all $r \in C$.

The standard definition (*e.g.,* [17]) does not require $(-)^*$ to be monotone, but this is a very natural condition, which holds of all up-to functions in the literature that we are aware of, and is needed for taking fixed points involving $(-)^*$.

This raises the question: is parameterized coinduction compatible with the use of up-to functions? Fortunately, the answer is yes. To see how up-to functions and parameterized coinduction can be combined, we observe that sound up-to functions respect the greatest fixed point:

Lemma 8. If $(-)^*$ is a sound up-to function for $f \in C \xrightarrow{\text{mon}} C$, then:

$$\nu f^* \sqsubseteq \nu f \quad \text{where} \quad f^* \overset{\text{def}}{=} \lambda r.\ f(r^*)$$

Proof. We have $\nu f^* \sqsubseteq f^*(\nu f^*) = f((\nu f^*)^*)$, which by Definition 2 implies $\nu f^* \sqsubseteq \nu f$. □

Thus, to prove $x \sqsubseteq \nu f$ using parameterized coinduction, we can instead show $x \sqsubseteq \mathrm{G}_{f^*}(\bot)$ for any sound up-to function $(-)^*$.

How does this interact with compositionality? Suppose we have two separate proofs using two up-to functions $\star$ and $\circ$:

$$x \sqsubseteq \mathrm{G}_{f^\star}(y \sqcup r) \quad \text{and} \quad y \sqsubseteq \mathrm{G}_{f^\circ}(x \sqcup r)$$

If $f^\star$ and f° happen to be equal, then we can combine the proofs using (COMPOSE) as expected. However, in general $(-)^\star$ and $(-)^\circ$ might be two arbitrarily different up-to functions, and thus we may not be able to apply the composition rule.

Fortunately, there is a solution to this dilemma if we confine ourselves to *respectful* up-to functions:

Definition 3. $(-)^* \in C \xrightarrow{\text{mon}} C$ is a *respectful up-to function* for $f \in C \xrightarrow{\text{mon}} C$ iff for any $r, s \in C$ the following holds:[3]

$$r \sqsubseteq s \ \wedge\ r \sqsubseteq f(s) \implies r^* \sqsubseteq f(s^*)$$

Lemma 9. If $(-)^*$ is a respectful up-to function for f, then it is also a sound one. (The proof follows that in [17].)

Respectfulness, although stronger than soundness, is still satisfied by most up-to functions of interest, and, crucially, has better compositionality properties. In particular:

Proposition 10.

1. $\mathrm{id} \overset{\text{def}}{=} (\lambda x.\ x)$ is a respectful up-to function for any f.
2. If u and u' are respectful up-to functions for f, then so is $u \circ u'$.
3. If each element of a set X is a respectful up-to function for f, then so is $\bigsqcup X$.[4]

We can thus define the greatest respectful up-to function for a given $f \in C \xrightarrow{\text{mon}} C$:

$$f^\dagger \overset{\text{def}}{=} \bigsqcup \{u \in C \xrightarrow{\text{mon}} C \mid u \text{ is a respectful up-to function for } f\}$$

Using Proposition 10.3, it is easy to show that $(-)^\dagger$ is a respectful up-to function, and that it is the greatest such. Moreover, we have:

Lemma 11. If $(-)^*$ is a respectful up-to function for f, then:

$$\mathrm{G}_{f^*} \sqsubseteq \mathrm{G}_{f^\dagger}$$

Proof. Follows from monotonicity of G and $(-)^* \sqsubseteq (-)^\dagger$. □

Now, using Lemma 11, we can bring the two proofs from above to a "common denominator",

$$x \sqsubseteq \mathrm{G}_{f^\dagger}(y \sqcup r) \quad \text{and} \quad y \sqsubseteq \mathrm{G}_{f^\dagger}(x \sqcup r),$$

and then compose them to get $x \sqcup y \sqsubseteq \mathrm{G}_{f^\dagger}(r)$.

[3] As with the definition of a sound up-to function, Sangiorgi [17] does not assume monotonicity of $(-)^*$, but here he requires a slightly weaker property: $r \sqsubseteq s \wedge r \sqsubseteq f(s) \implies r^* \sqsubseteq s^*$.

[4] Here and elsewhere, we treat a function space $A \to C$ as the complete lattice obtained by canonically lifting C pointwise.

$$\frac{(e, e') \in r}{(e, e') \in r^{\mathrm{ctx}}} \quad (\textsc{incl}) \qquad\qquad \frac{}{(e, e) \in r^{\mathrm{ctx}}} \quad (\textsc{refl})$$

$$\frac{(e, e') \in r \qquad (e_1, e_1') \in r \qquad (e_2, e_2') \in r}{(\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2, \mathsf{if}\ e'\ \mathsf{then}\ e_1'\ \mathsf{else}\ e_2') \in r^{\mathrm{ctx}}} \quad (\textsc{if})$$

$$\frac{\forall v.\, \begin{pmatrix} e_1[v/x, \mathsf{fix}\, f(x).\, e_1/f], \\ e_1'[v/x', \mathsf{fix}\, f'(x').\, e_1'/f'] \end{pmatrix} \in r \qquad (e_2, e_2') \in r}{((\mathsf{fix}\, f(x).\, e_1)\ e_2, (\mathsf{fix}\, f'(x').\, e_1')\ e_2') \in r^{\mathrm{ctx}}} \quad (\textsc{appv})$$

Figure 4. A respectful up-to function.

Actually, since the greatest respectful up-to function is so powerful, we see no point in ever stating a proof component's contribution involving a different respectful up-to function. In other words: state your goal in terms of the greatest one. The following properties enable the use of zero or more particular respectful up-to functions inside the proof of such a goal:

Lemma 12. If $(-)^*$ is a respectful up-to function for f, then for any $r \in C$:

1. $r \sqsubseteq r^\dagger$
2. $(r^\dagger)^* \sqsubseteq r^\dagger$

Proof. Follows from Proposition 10.1 and 10.2, respectively. □

We remark that the greatest respectful up-to function also allows us to use up-to reasoning at any point in a proof by parameterized coinduction (not just after unfolding).

Theorem 13. $\mathrm{G}_{f^\dagger}(r) \equiv (\mathrm{G}_{f^\dagger}(r))^\dagger$

Proof. The ($\sqsubseteq$) direction holds by Lemma 12.1. The ($\sqsupseteq$) direction is more complicated. By (TARSKI) it suffices to show $\mathrm{G}_{f^\dagger}(r)^\dagger \sqsubseteq f((r \sqcup \mathrm{G}_{f^\dagger}(r)^\dagger)^\dagger)$. This follows from Lemma 12.2 if we can show $\mathrm{G}_{f^\dagger}(r)^\dagger \sqsubseteq f((r \sqcup \mathrm{G}_{f^\dagger}(r)^\dagger)^{\dagger\dagger})$. By respectfulness in turn it suffices to show $\mathrm{G}_{f^\dagger}(r) \sqsubseteq (r \sqcup \mathrm{G}_{f^\dagger}(r)^\dagger)^\dagger$ and $\mathrm{G}_{f^\dagger}(r) \sqsubseteq f((r \sqcup \mathrm{G}_{f^\dagger}(r)^\dagger)^\dagger)$. Both are not hard to show using Lemma 12.1 and, for the second, monotonicity of f and $(-)^\dagger$. □

Simulation Example. We return to the simulation example presented in Section 3, and show how to apply an "up to context" technique in order to simplify the proof.

To reason up to contexts, one usually defines a context closure operation. We could do this here as well, but, in combination with $(-)^\dagger$, we can get away with something simpler: Figure 4 defines a function $(-)^{\mathrm{ctx}} \in \mathcal{P}(\mathsf{exp}^2) \to \mathcal{P}(\mathsf{exp}^2)$. It is straightforward to verify that this is a respectful up-to function for *sim* (the proof can be found in our Coq formalization). Observe, however, that the definition is not recursive and thus $(-)^{\mathrm{ctx}}$ only adds *atomic* contexts (*i.e.,* contexts whose definition does not involve any form of recursion); we will see in a moment how we can nevertheless get the full power of a proper context closure operation.

The final rule (APPV) is the case for function application. As $\nu\, sim$ relates values only if they are identical, the simpler rule—$\{$if $(e_1, e_1') \in r$ and $(e_2, e_2') \in r$ then $(e_1\, e_2,\ e_1'\, e_2') \in r^{\mathrm{ctx}}\}$—while sound, is useless because the first assumption requires e_1 and e_1' to evaluate to syntactically the same function. Therefore, APPV requires the functions to be already values, and checks that their bodies are related whenever the functions are applied to the same arguments. Finally, note that the following rule can be derived from APPV:

$$\frac{(e_1, e_1') \in r \qquad (e_2, e_2') \in r}{(e_1; e_2,\ e_1'; e_2') \in r^{\mathrm{ctx}}} \quad (\textsc{seq})$$

With the help of this up-to function we will now prove the same two lemmas as before (Lemmas 6 and 7), except that we replace G_{sim} by $\mathrm{G}_{sim^\dagger}$.

Lemma 14. For all values f_1 and f_2, we have:

$$\{(\mathsf{restart}\ f_1\ \underline{n},\ \mathsf{restart}\ f_2\ \underline{n}) \mid n \in \mathbb{Z}\} \subseteq \mathrm{G}_{sim^\dagger}(\{(f_1\ \underline{0}, f_2\ \underline{0})\})$$

Lemma 15. $R_0 \subseteq \mathrm{G}_{sim^\dagger}(R_1)$.

Proof of Lemma 14. As in Section 3, we apply the accumulation principle and reason about the first two steps of execution with the help of Lemma 12.1. Then we obtain the following proof obligation:

$$(e_1,\ e_2) \in (R \cup \mathrm{G}_{sim^\dagger}(R))^\dagger$$

where:

$$e_i := \mathsf{if}\ \underline{n} > 0\ \mathsf{then}\ (\mathsf{output}\ \underline{n}; \mathsf{restart}\ f_i\ (\underline{n} - \underline{1}))\ \mathsf{else}\ f_i\ \underline{0}$$
$$R := \{(\mathsf{restart}\ f_1\ \underline{n},\ \mathsf{restart}\ f_2\ \underline{n}) \mid n \in \mathbb{Z}\} \cup \{(f_1\ \underline{0}, f_2\ \underline{0})\}$$

At this point, since $(-^\dagger)^{\mathrm{ctx}} \sqsubseteq (-)^\dagger$ by Lemma 12.2, we may apply one of the rules from Figure 4 in order to simplify this goal. We pick IF and hence it remains to show the following three:

1. $(\underline{n} > 0,\ \underline{n} > 0) \in (R \cup \mathrm{G}_{sim^\dagger}(R))^\dagger$
2. $((\mathsf{output}\ \underline{n}; \mathsf{restart}\ f_1\ (\underline{n} - \underline{1})),$
   $(\mathsf{output}\ \underline{n}; \mathsf{restart}\ f_2\ (\underline{n} - \underline{1}))) \in (R \cup \mathrm{G}_{sim^\dagger}(R))^\dagger$
3. $(f_1\ \underline{0}, f_2\ \underline{0}) \in (R \cup \mathrm{G}_{sim^\dagger}(R))^\dagger$

Now, by the *same argument*, we may apply another of these rules in each case. So, effectively, we can apply an arbitrary number of rules and thus did not lose any power by defining $(-)^{\mathrm{ctx}}$ in terms of atomic contexts only.

(1) is solved by rule REFL, and (3) is solved by Lemma 12.1 since the terms are related by R. To show (2), we first apply rules SEQ and REFL, and then use Lemma 12.1 to reduce the goal to:

$$(\mathsf{restart}\ f_1\ (\underline{n} - \underline{1}),\ \mathsf{restart}\ f_2\ (\underline{n} - \underline{1})) \in \mathrm{G}_{sim^\dagger}(R)$$

This is easily shown by unfolding (Lemma 3), performing a step of computation, which converts the two instances of $\underline{n} - \underline{1}$ to $\underline{n-1}$, and then concluding once again with Lemma 12.1.

The attentive reader may have noticed that we never seem to need rule INCL. Indeed, this is a side effect of reasoning via Lemma 12. Nevertheless, the rule is necessary: without it, $(-)^{\mathrm{ctx}}$ would not be respectful. □

Proof of Lemma 15. Here, reasoning up to contexts seems not to buy us anything, so we just derive the goal from our old proof (Lemma 7) via Lemma 11. □

Finally, we can, as before, use (COMPOSE) to deduce that $R_0 \subseteq \mathrm{G}_{f^\dagger}(\emptyset)$. By Lemmas 2, 8 and 9, this implies $R_0 \subseteq \nu f$.

5. Mechanizing Parameterized Coinduction

In this section, we discuss at a high level the issues raised by formalizing our parameterized coinduction principle from Section 2 in a proof assistant such as Isabelle/HOL or Coq. Details about our Coq implementation, as well as examples using it, follow in Section 7.

To establish some common terminology, we say that a *predicate* of arity n is a (dependent) function of type

$$\Pi a_1{:}A_1.\ \Pi a_2{:}A_2(a_1).\ \ldots \Pi a_n{:}A_n(a_1, \ldots, a_{n-1}).\ S$$

where the sort S is impredicative (`Prop` in Coq, `bool` in Isabelle). If instead the sort S is predicative (`Type` in Coq, or `Set` in Agda), we call such objects *indexed sets*.

Predicates are normally used for writing "proofs" (whose computational meaning is not of interest), whereas indexed sets are used for writing "programs" (whose computational meaning is their main point of interest), and of these two actually only *predicates* form complete lattices. Therefore, in this paper, whose main focus is on coinductive *proofs*, we shall largely ignore indexed sets, and only briefly discuss them in Section 8.

There are two ways in which a formalization can be done, namely what we call the *external* and the *internal* approach. They differ in the way in which the parameterized greatest fixed point is constructed.

- The *external approach* develops a library of complete lattices that uses Tarski's construction for greatest fixed points, and then uses that library to define G. Defining such a library, however, requires impredicative quantification, and so this approach only works for Isabelle and Coq, but not Agda.
- The *internal approach* defines G directly using the proof assistant's primitive mechanism for defining coinductive types (*e.g.,* Coq's **CoInductive** or Agda's ∞ constructor). In Isabelle/HOL and related systems, where coinductive types/relations are not primitive, one must instead follow the external approach.

In Coq, where both approaches are applicable, the internal approach, besides being more direct, is also easier to use because Coq's automation works much better there. The main problem is that in the external approach, Coq's automation tactics do not know how to unfold the lattice-theoretic constructs, and so the user has to instruct Coq manually to do so.

We now discuss the two approaches in more detail.

The External Approach. In this approach, one defines a generic library of complete lattices and greatest fixed points of arbitrary monotone endofunctions, and uses that to construct G and prove its properties. The library can then be instantiated to the application domain at hand.

This approach is arguably as general and modular as it gets, and works quite well for both Coq and Isabelle/HOL. The Isabelle implementation is actually simpler: Isabelle already has a complete lattice theory, which only needs to be extended with our parameterized coinduction.

To apply the library definitions and lemmas to arbitrary predicates, one has to prove that `Prop` (in Coq, or `bool` in Isabelle) forms a complete lattice and that the space of (dependent) functions to a complete lattice forms again a complete lattice (the pointwise lifting). Using *type classes* (in Isabelle or Coq) or *canonical structures* (only in Coq), one can easily arrange that the appropriate lattice structure for a given predicate is automatically inferred.

The Internal Approach. As mentioned above, the internal approach depends on having primitive support for coinductive types, but if it is available, it can be more a convenient option. However, the applicability of the internal approach is somewhat limited for two further reasons:

1. The "only" objects that one can define with the primitive coinductive definition mechanism are predicates (and indexed sets). Hence, this approach does not work for arbitrary complete lattices. However, it is still very useful in practice, because (*i*) predicates are already quite expressive, and (*ii*) we found a clever trick that enables us to extend this approach to *refined* predicates, explained in detail in Section 6.2.
2. Moreover, the predicates themselves must have a certain syntactic form: all recursive uses of a (co-)inductively defined object in its definition must be *strictly positive*, *i.e.,* roughly speaking, not occur on the left of an arrow. While this syntactic condition is overly restrictive for predicates, it is important for ensuring consistency in the case of indexed sets, and is thus imposed for uniformity on all coinductive definitions.

What does this entail for the formalization? Consider the powerset lattice from Section 3. It satisfies condition (1) above (*i.e.,* it is a predicate type), but condition (2) prevents one from parameterizing the definition of G_f over f. Recall its definition:

$$\begin{aligned}&\mathrm{G} \in (\mathcal{P}(\mathsf{exp}^2) \xrightarrow{\text{mon}} \mathcal{P}(\mathsf{exp}^2)) \xrightarrow{\text{mon}} (\mathcal{P}(\mathsf{exp}^2) \xrightarrow{\text{mon}} \mathcal{P}(\mathsf{exp}^2))\\ &\mathrm{G}_f(x) \overset{\text{def}}{=} \nu y.\ f(x \sqcup y)\end{aligned}$$

This can be translated into Coq as follows:

```
Definition erel :=  exp → exp → Prop.
CoInductive G (f: erel → erel) (fM: monotonic f)
           (x: erel) (e1 e2: exp) : Prop :=
  | G_fold (IN: f (x ⊔ G f fM x) e1 e2). (* REJECTED *)
```

This definition, however, is rejected because it violates strict positivity: in the type of the constructor argument `IN`, `G` occurs in the argument of a function application, where the function, `f`, is a variable. This is forbidden, intuitively because `f` could be instantiated to a function that uses its argument on the left side of an arrow.

Fortunately, there is a simple—but clever—trick that lets us work around the strict positivity requirement as long as the function in question (here: $\lambda y. f(x \sqcup y)$) is monotone. This trick is based on *Mendler-style recursion* and explained in Section 6.1. Consequently, in the internal approach, we can also define a library for parameterized greatest fixed points of arbitrary monotone predicates up to some fixed arity—see Section 7.4.

6. Mendler-Style Recursion to the Rescue!

Mendler [12] proposed a strongly normalizing calculus featuring an unusual treatment of inductive and coinductive types. In this section, we show how recursion in the Mendler style can be used to address the two issues mentioned in Section 5 that come up when mechanizing parameterized greatest fixed points following the internal approach.

We first review the idea of Mendler-style recursion in the setting of complete lattices and show how it can be employed to overcome Coq's strict positivity restriction in the case of predicates. This observation is not novel, and has been made before by Matthes [11].

Second, we generalize this theory in a way that yields a method for defining fixed points in complete *sub*lattices. In the internal approach, this enables us to define *refined* predicates coinductively, while, in the external approach, it aids in avoiding dealing with dependent types. As far as we can tell, both the generalization of the theory and its application to mechanization are new.

6.1 Strict Positivization

A convenient way to view Mendler's fixed point constructions is as ordinary fixed points of *monotonized* functions. Given a function $g \in C \to C$ (where C is a complete lattice), there are two canonical ways of adapting this function slightly to make it monotone:

Definition 4 (Monotonizations)**.** We define $\lfloor g \rfloor, \lceil g \rceil \in C \xrightarrow{\text{mon}} C$:

$$\begin{aligned}\lfloor g \rfloor &\overset{\text{def}}{=} \lambda x.\ \textstyle\bigsqcap\{g(y) \mid y \sqsupseteq x\}\\ \lceil g \rceil &\overset{\text{def}}{=} \lambda x.\ \textstyle\bigsqcup\{g(y) \mid y \sqsubseteq x\}\end{aligned}$$

Their monotonicity is easy to see, given that for $x \sqsubseteq x'$ we have:

$$\begin{aligned}&\{g(y) \mid y \sqsupseteq x\} \supseteq \{g(y) \mid y \sqsupseteq x'\}\\ \text{and}\quad &\{g(y) \mid y \sqsubseteq x\} \subseteq \{g(y) \mid y \sqsubseteq x'\}.\end{aligned}$$

It is also easy to verify that both $\lfloor - \rfloor$ and $\lceil - \rceil$ form a Galois connection with the canonical embedding of $C \xrightarrow{\text{mon}} C$ in $C \to C$, in the way depicted in Figure 5. This means that

$$\begin{tikzcd}[column sep=huge]
C \xrightarrow{\text{mon}} C \arrow[r] \arrow[r, phantom, shift left=3ex, "\perp"] \arrow[r, phantom, shift right=3ex, "\perp"] & C \to C \arrow[l, bend right=50, "\lceil - \rceil"'] \arrow[l, bend left=50, "\lfloor - \rfloor"]
\end{tikzcd}$$

Figure 5. Monotonization operators and their Galois connections.

- $\forall f \in C \xrightarrow{\text{mon}} C.\ f \sqsubseteq \lfloor g \rfloor \iff f \sqsubseteq g$
  i.e., $\lfloor g \rfloor$ is the greatest monotone function below g, and
- $\forall f \in C \xrightarrow{\text{mon}} C.\ \lceil g \rceil \sqsubseteq f \iff g \sqsubseteq f$
  i.e., $\lceil g \rceil$ is the least monotone function above g,

and thus $\lfloor g \rfloor \sqsubseteq g \sqsubseteq \lceil g \rceil$ (which explains our choice of notation). Now, if g is already monotone, then all three are equivalent, and consequently so are their least and greatest fixed points:

Proposition 16. If $g \in C \xrightarrow{\text{mon}} C$, then $\lfloor g \rfloor \equiv g \equiv \lceil g \rceil$, and hence $\mu\lfloor g \rfloor \equiv \mu g \equiv \mu\lceil g \rceil$ and $\nu\lfloor g \rfloor \equiv \nu g \equiv \nu\lceil g \rceil$.

So why is this interesting at all? The point is that monotonizing an already monotone function using $\lceil - \rceil$ yields the same function in a form that translates to a strictly positive one in Coq. To see this, let us apply it to the definition of G that we attempted at the end of the previous section.

Given a monotone function f, there are actually two ways to apply Proposition 16 to $\mathrm{G}_f(x)$. In the first, we "monotonize" f:

$$\mathrm{G}_f(x) \equiv \mathrm{G}_{\lceil f \rceil}(x) = \nu z.\ \bigsqcup\{f(y) \mid y \sqsubseteq x \sqcup z\}$$

In the second, we "monotonize" $\lambda z.\ f(x \sqcup z)$:

$$\mathrm{G}_f(x) = \nu z.\ f(x \sqcup z) \equiv \nu z.\ \bigsqcup\{f(x \sqcup y) \mid y \sqsubseteq z\}$$

Hence we have two possible definitions of G in Coq:

```
CoInductive G (f: erel → erel)
          (x: erel) (e1 e2: exp) : Prop :=
  | G_fold y (LE: y ⊆ x ∪ G f x) (IN: f y e1 e2).

CoInductive G (f: erel → erel)
          (x: erel) (e1 e2: exp) : Prop :=
  | G_fold y (LE: y ⊆ G f x) (IN: f (x ∪ y) e1 e2).
```

Note the similarity to the rejected definition in the previous section and the absence of the `fM` assumption. These new definitions are well-formed for arbitrary functions f (not necessarily monotone), and dropping `fM` makes them more convenient to work with. Of course, we need to assume monotonicity in statements about `G` then instead. We prefer the first definition, because then monotonicity of f is required only for unfolding G_f (the $\sqsubseteq$ direction of Lemma 3) and is not needed for the accumulation theorem (Theorem 4).

We remark that this trick of monotonizing functions that are already monotone in order to obtain a strictly positive form applies to inductive predicates (defined using Coq's Inductive command) as well, but we do not exploit that observation in this development.

6.2 Fixed Points in Sublattices

The Problem. Recall that, in the internal approach, (co-)inductive definitions are limited to predicates and indexed sets. Here we show how to broaden this to *refined* predicates.

By refined predicates, we mean objects whose type has the form

$$\{x : \Pi a_1{:}A_1.\ \ldots \Pi a_n{:}A_n(a_1, \ldots, a_{n-1}).\ \texttt{Prop} \mid P(x)\},$$

i.e., a regular predicate type refined by some property P. This is best illustrated with an example. Imagine we want to define the greatest fixed point of a function

$$f \in (\mathcal{P}(\mathsf{exp}^2) \to \mathcal{P}(\mathsf{exp}^2)) \xrightarrow{\text{mon}} (\mathcal{P}(\mathsf{exp}^2) \to \mathcal{P}(\mathsf{exp}^2)).$$

We can easily do this using Coq's **CoInductive** mechanism since $\mathcal{P}(\mathsf{exp}^2) \to \mathcal{P}(\mathsf{exp}^2)$ is naturally expressed as a predicate type (we may have to bring f into strictly positive form, of course). Now imagine we want to define the greatest fixed point of a different function

$$g \in (\mathcal{P}(\mathsf{exp}^2) \xrightarrow{\text{mon}} \mathcal{P}(\mathsf{exp}^2)) \xrightarrow{\text{mon}} (\mathcal{P}(\mathsf{exp}^2) \xrightarrow{\text{mon}} \mathcal{P}(\mathsf{exp}^2)).$$

Note that the complete lattice $\mathcal{P}(\mathsf{exp}^2) \xrightarrow{\text{mon}} \mathcal{P}(\mathsf{exp}^2)$ cannot be expressed as a predicate type due to the restriction of the function space. Instead, it can be seen as a refined predicate type (where P is monotonicity).

In the external approach, there is no problem: we can define the complete lattice structure for this type, then use that to write down the faithful definition of g, prove that it is monotone, and finally just apply the greatest fixed point operator to it. But in the internal approach, this is not possible. The best we can do there, so it seems, is take the greatest fixed point in the unrestricted function space. However, (i) this assumes that g is well-defined and monotone in the larger space, and (ii) even if that fixed point exists, it will not necessarily be the desired one.

The Solution. Our solution is as follows: we do indeed take the greatest fixed point in the unrestricted space, but only after modifying the function in a way that ensures (i) that it is well-defined and monotone, and (ii) that the result actually coincides with the desired greatest fixed point in the original restricted space. The math behind this is a generalization of what we saw in Section 6.1, and is basically stated in terms of an arbitrary complete lattice and an arbitrary complete sublattice thereof that preserves meets and/or joins.

Because the theory is so general, it could actually also be used in the external approach, where the (non-negligible) benefit would be avoiding to work with (dependent) subset types.

The Theory. Consider two complete lattices B and C. We are interested in the scenario where there exists an embedding of B in C, in the following sense:

Definition 5. A function $i \in B \to C$ is an *embedding* of B in C, written $i : B \hookrightarrow C$, iff $\forall b, b' \in B.\ b \sqsubseteq_B b' \iff i(b) \sqsubseteq_C i(b')$.

(Note that this implies injectivity.) In the case where B is a complete sublattice of C, the canonical injection from B to C constitutes such an embedding.

With the help of an embedding, we can now define generalized versions of the monotonization operators from Section 6.1:

Definition 6 (Generalized Monotonizations)**.** For $i : B \hookrightarrow C$ and a function $g \in B \to B$ we define $\lfloor g \rfloor_i, \lceil g \rceil^i \in C \xrightarrow{\text{mon}} C$:

$$\begin{aligned} \lfloor g \rfloor_i &\overset{\text{def}}{=} \lambda x.\ \textstyle\bigsqcap\{i(g(y)) \mid y \in B \wedge i(y) \sqsupseteq x\} \\ \lceil g \rceil^i &\overset{\text{def}}{=} \lambda x.\ \textstyle\bigsqcup\{i(g(y)) \mid y \in B \wedge i(y) \sqsubseteq x\} \end{aligned}$$

Now, if i preserves meets and joins (*e.g.,* because C is a function space with a pointwise ordering and B its restriction to monotone functions), then the Galois connections from earlier generalize as well. And, if moreover g is monotone, then the least and greatest fixed points of g, $\lfloor g \rfloor_i$ and $\lceil g \rceil^i$ coincide modulo the embedding.

Proposition 17. If $\forall X.\ i(\bigsqcap X) \equiv \bigsqcap(i(X))$, the following hold for any $f \in C \xrightarrow{\text{mon}} C$ and $g \in B \to B$:

1. $f \sqsubseteq \lfloor g \rfloor_i \iff \lfloor f \rfloor_i \sqsubseteq g$, for $\lfloor f \rfloor_i \overset{\text{def}}{=} \bigsqcap\{y \mid i(y) \sqsupseteq f(i(x))\}$.
2. If g is monotone, then $\mu\lfloor g \rfloor_i \equiv i(\mu g)$ and $\nu\lfloor g \rfloor_i \equiv i(\nu g)$.

Proposition 18. If $\forall X.\ i(\bigsqcup X) \equiv \bigsqcup(i(X))$, the following hold for any $f \in C \xrightarrow{\text{mon}} C$ and $g \in B \to B$:

1. $\lceil g \rceil^i \sqsubseteq f \iff g \sqsubseteq |f|^i$, for $|f|^i \overset{\text{def}}{=} \bigsqcup\{y \mid i(y) \sqsubseteq f(i(x))\}$.
2. If g is monotone, then $\mu\lceil g \rceil^i \equiv i(\mu g)$ and $\nu\lceil g \rceil^i \equiv i(\nu g)$.

Finally, observe that the results in Section 6.1 are merely a special case of the results presented here, namely where $B = C$ and $i = \text{id}$, in which case $\lfloor - \rfloor_i = \lfloor - \rfloor$, $\lceil - \rceil^i = \lceil - \rceil$, and $|-|^i = |-|_i = \text{id}$.

The Example. To see what this looks like in practice, let us return to the example g from the beginning. Since $\mathcal{P}(\mathsf{exp}^2) \xrightarrow{\text{mon}} \mathcal{P}(\mathsf{exp}^2)$ is a complete sublattice of $\mathcal{P}(\mathsf{exp}^2) \to \mathcal{P}(\mathsf{exp}^2)$, we have by Propositions 17 and 18 (for the canonical injection i):

$$\nu g \equiv \nu \lfloor g \rfloor_i \equiv \nu \lceil g \rceil^i$$

We can now mechanize either $\lfloor g \rfloor_i$ or $\lceil g \rceil^i$. Because $\lceil - \rceil^i$ has the added benefit of yielding a strictly positive form (even if g does not have one), we pick the latter. To see how this translates into Coq, note that

$$\lceil g \rceil^i \in (\mathcal{P}(\mathsf{exp}^2) \to \mathcal{P}(\mathsf{exp}^2)) \xrightarrow{\text{mon}} \mathcal{P}(\mathsf{exp}^2) \to \mathcal{P}(\mathsf{exp}^2)$$
$$\lceil g \rceil^i = \lambda x.\ \bigsqcup\{i(g(y)) \mid y \in \mathcal{P}(\mathsf{exp}^2) \xrightarrow{\text{mon}} \mathcal{P}(\mathsf{exp}^2) \wedge i(y) \sqsubseteq x\}$$

and that $\nu\lceil g \rceil^i$, by unfolding, is equal to:

$$\bigsqcup\{i(g(y)) \mid y \in \mathcal{P}(\mathsf{exp}^2) \xrightarrow{\text{mon}} \mathcal{P}(\mathsf{exp}^2) \wedge i(y) \sqsubseteq \nu\lceil g \rceil^i\}$$

Accordingly, we write:

```
CoInductive nu_g (r: erel) (e1 e2: exp): Prop :=
  nu_g_fold (y: erel → erel) (yM: monotonic y)
         (LE: ∀r', y r' ⊆ nu_g r') (IN: g y yM e1 e2).
```

Here, `g y yM e1 e2` may look a bit different, depending on how `g` is defined (if it is not defined explicitly, one can just inline it here).

Similar to the first part, and as is evident from the theory, all this applies to induction (*i.e.,* least fixed points) as well.

Remark. The type that we used for illustration,

$$(\mathcal{P}(\mathsf{exp}^2) \xrightarrow{\text{mon}} \mathcal{P}(\mathsf{exp}^2)) \xrightarrow{\text{mon}} (\mathcal{P}(\mathsf{exp}^2) \xrightarrow{\text{mon}} \mathcal{P}(\mathsf{exp}^2)),$$

is highly reminiscent of one that occurs in the meta-theory of *Relation Transition Systems (RTS)* [7], a new kind of semantic model that we recently introduced for compositional reasoning about program equivalences in higher-order stateful languages. Reasoning using this method feels very much like a regular bisimulation argument: to show the equivalence of two functions, one has to construct a "local knowledge" relating them and then prove its "consistency". In our mechanized RTS proofs, we found that explicitly defining this local knowledge up front was quite a painful experience, for essentially the same reasons as defining the Tarski-style simulation relation was painful in the example in Section 3.

It turns out that being a consistent local knowledge can be expressed as being a postfixed point of a certain monotone function of basically the above type:

$$(\mathcal{P}(\mathsf{exp}^2) \xrightarrow{\text{mon}} \mathcal{P}(\mathsf{exp}^2)) \xrightarrow{\text{mon}} (\mathcal{P}(\mathsf{exp}^2) \xrightarrow{\text{mon}} \mathcal{P}(\mathsf{exp}^2))$$

By following the internal approach and using the trick presented here, we were able to bring the benefits of parameterized coinduction to our RTS framework, while keeping changes to existing definitions to a minimum.

7. Coq Implementation and Evaluation

In this section, we discuss our implementation of parameterized coinduction in Coq, and compare it to other approaches, most notably Coq's builtin `cofix` tactic.

Technique	Lines	Proof (s)	Qed (s)	Composes?
Tarski	74	8.5	3.5	no
Coq's `cofix` (§7.3)	8	7.9	14.0	no
Internal (§7.1)	8	8.6	5.1	yes
External (§7.2)	8	11.8	4.3	yes

Table 1. Comparison of the mechanization approaches

For a fair comparison, we have carried out the simulation proof of the example from Section 3 using a number of different approaches, and show the results in Table 1. In each case, we report: (a) the number of lines of proof and auxiliary lemmas/definitions, not counting the lines of generic library lemmas and tactics as these can be defined once and for all; (b) the time taken to execute the corresponding proof script; and (c) the time taken to check (`Qed`) that the constructed proof object (from running the script) is valid.

Tarski: Explicitly define a simulation relation, prove that it is valid (*i.e.,* a postfixed point of sim), and then apply (TARSKI). This requires by far the most human effort, which is partially reflected in lines of code needed to define the simulation relation; the proof itself is quite short, and hence `Qed`-checking is fast.

Cofix: Use Coq's builtin `cofix` tactic to simulate an incremental proof. We shall discuss this approach in detail in Section 7.3, but for the moment we remark that proof checking (the Qed column) for `cofix` is significantly slower than for all the other approaches, and can become a serious usability bottleneck in larger examples.

Internal: Use a direct encoding of G_{sim} following the internal approach described in Section 5. (See Section 7.1 for the details.)

External: Use a library-based definition of G_{sim} following the external approach. (See Section 7.2 for the details.)

Among these approaches, it is quite clear that the internal and external approaches are roughly equivalent, with the internal approach being somewhat more efficient by virtue of using primitive definitions, and avoiding the use of clever inference during type checking (namely, canonical structures or type classes) that the external approach requires. The two are much faster than the `cofix` approach. Moreover, they enable compositional proof developments, unlike the other two approaches.

7.1 Internal Implementation of Parameterized Coinduction

We now show what the internal approach to mechanizing parameterized coinduction (discussed in Section 5) looks like in practice, by applying it to the example from Section 3.

For the sake of a sharper contrast with the external approach, we do not parameterize G (which we could, with the help of the trick from Section 6.1), but directly define G_{sim}:

```
CoInductive psimil r (e1 e2 : exp) : Prop :=  psimil_fold
  (VAL: ∀v : val, e1 = v → e2 ⇝τ v)
  (EXP: ∀q e1', e1 →q e1' →
       ∃ e2', e2 ⇝q e2' ∧ (psimil r e1' e2' ∨ r e1' e2')).
```

Observe how `psimil r` corresponds to $\mathrm{G}_{sim}(r)$. Next, we prove the following accumulation property, which we could in principle instrument Coq to generate and prove automatically.

```
Theorem psimil_acc : ∀l r,
  (∀ r', r ⊆ r' → l ⊆ r' → l ⊆ psimil r') →
  l ⊆ psimil r.
```

This is a variant of the ($\Longleftarrow$) direction of Theorem 4 in "Mendler form", which avoids the explicit join operation and is thus a bit more convenient to use interactively than its simpler counterpart. Also, note that the unfold property (Lemma 3) is implicit in the definition of psimil, and that we do not need to state Lemma 2 either, because we simply work with $\mathrm{G}_{sim}(\bot)$ directly.

We can now already state and prove the example from Section 3:

```
Theorem fg_simulated :
  ∀ n m (EQ: n = 2 * m), psimil bot2 (f @ n) (g @ m).
Proof.
  pcofix CIH using psimil_acc.
  intros; subst; do 6 psimil_step 1; do 2 psimil_step 0.
  destruct m0; psimil_step 2; [|eauto].
  left; fold restart f g; generalize (m+(m+0)).
  pcofix CIH' using psimil_acc.
  intros; do 3 psimil_step 1.
  destruct n; psimil_step 1; [eauto|].
  do 3 psimil_step 1; eauto.
Qed.
```

In this proof script, all tactics except psimil_step and pcofix are standard (*i.e.,* part of Coq). The former matches one step of e_1 with n steps of e_2. The latter assists in applying parameterized coinduction: pcofix rewrites the goal to a form suitable for applying the accumulation theorem, applies the theorem, and then simplifies the result. For instance, at the beginning of the proof (when the goal is simply the theorem statement), applying psimil_acc directly is not possible, because Coq cannot figure out how to instantiate its l parameter. Hence we invoke pcofix, which first transforms the goal into the following equivalent statement:

```
(fun x y ⇒ ∃n m, f @ n = x ∧ g @ m = y ∧ n = 2 * m)
⊑ psimil bot2
```

After that, it applies psimil_acc (its argument), which now matches the goal, and finally pcofix simplifies the resulting proof state so that the user never sees this explicit function expression:

```
r : exp → exp → Prop
CIH : ∀n m, n = 2 * m → r (f @ n) (g @ m)
────────────────────────────────────────────────
∀ n m, n = 2 * m → psimil r (f @ n) (g @ m)
```

Note that the new goal essentially also appears as the coinduction hypothesis CIH, except that psimil has been removed, and so the hypothesis is "semantically guarded."

The second use of pcofix corresponds to the "Accumulation Step" of the proof sketch, and results in the following proof state:

```
r : exp → exp → Prop
CIH : ∀n m, n = 2 * m → r (f @ n) (g @ m)
CIH' : ∀n, r ((restart @ f) @ n) ((restart @ g) @ n)
────────────────────────────────────────────────
∀ n, psimil r ((restart @ f) @ n) ((restart @ g) @ n)
```

It essentially added the additional assumption CIH', but to achieve this, it had quite some cleaning up to do internally: after applying the accumulation theorem, a new r' (greater than r) had been introduced, so pcofix used transitivity of ⊑ to convert all existing hypotheses involving r into statements about r', which it then finally renamed to r again.

Although we have demonstrated the use of our pcofix tactic on one example, it is not tailored to this particular example—rather, it is a general tactic that applies to arbitrary coinductive predicates. It is implemented in Ltac with the help of the **hpattern** [6] library for handling dependent types.

7.2 External Implementation of Parameterized Coinduction

We implement G from Section 2 using a complete lattice library that provides a type of complete lattices (cola) and operations such as gfp, ⊑, ⊔ (all with the obvious meaning):

```
Definition G {C : cola} (f : C → C) (x : C) :=
  gfp (fun y ⇒ f (x ⊔ y)).
```

Next, we prove Lemma 3 (here as two separate lemmas) and (the interesting direction of) the accumulation property (Theorem 4, again expressed in Mendler-style to make it easier to use). Like before, there is no point in defining $\nu\, sim$ explicitly and proving Lemma 2, because we can just work with $\mathrm{G}_{sim}(\bot)$ directly.

```
Lemma G_fold: ∀(C:cola) (f:C → C), monotone f →
  ∀ r: C, f (G f r ⊔ r) ⊑ G f r.
Lemma G_unfold: ∀(C:cola) (f:C → C), monotone f →
  ∀ r: C, G f r ⊑ f (G f r ⊔ r).
Theorem G_acc: ∀(C:cola) (f:C → C), monotone f →
  ∀ l r, (∀ r', r ⊑ r' → l ⊑ r' → l ⊑ G f r') →
  l ⊑ G f r.
```

To use this library, we simply define the generating function sim.

$$\begin{array}{l}\textbf{Inductive}\ \texttt{sim r (e1 e2: exp) : Prop :=\ \ sim\_fold}\\ \quad\texttt{(VAL: }\forall\texttt{v : val, e1 = v}\rightarrow\texttt{e2}\overset{\tau}{\rightsquigarrow}\texttt{v)}\\ \quad\texttt{(EXP: }\forall\texttt{q e1', e1}\xrightarrow{\texttt{q}}\texttt{e1'}\rightarrow\exists\texttt{e2', e2}\overset{\texttt{q}}{\rightsquigarrow}\texttt{e2'}\wedge\texttt{r e1' e2').}\end{array}$$

Then G sim corresponds to G_{sim}, where the implicit argument C:cola is automatically inferred from the type of sim with the help of Coq's canonical structure mechanism.

The statement of the example from Section 3 is the same as in the internal implementation (fg_simulated above), except that psimil is replaced by (G sim). In the corresponding proof script, the argument to the pcofix tactic changes from psimil_acc to (G_acc sim). Similarly, the psimil_step tactic is replaced by one that applies (G_fold sim) instead of psimil_fold.

7.3 Coq's cofix

Coq's standard coinduction principle is rather syntactic and thus quite different from the Tarski principle. It basically works as follows: to show $x \sqsubseteq \nu f$, one invokes Coq's builtin cofix tactic, which adds the very same proposition as an assumption to the local context. Being an ordinary assumption, it can be used at any point in the proof script. However, once the proof is finished, Coq runs a syntactic check on the proof term and accepts it only if the use of the coinductive assumption is guarded [3].

It is possible, although not well known, that one can nest uses of cofix and thereby achieve a form of incremental (albeit non-compositional) coinduction. For example, if we take the proof script from Section 7.1 and simply replace the two occurrences of "pcofix CIH['] using psimil_acc" with "cofix CIH[']", we obtain a valid proof!

Despite its surprising usefulness in allowing a form of accumulation, Coq's cofix approach to coinductive proofs has several important drawbacks due to its syntactic nature:[5]

- It is non-compositional. Inside a proof via cofix, the use of normal lemmas involving the coinduction hypothesis is not permitted by guardedness checking, because they are treated opaquely. One can of course mark these lemmas as transparent (thereby further slowing down proof checking), but this does not yield proper compositionality: from the *statements* of the transparent lemmas alone, one cannot know whether their proofs can be composed together to yield a valid proof.[6]
- It is inefficient. Guardedness checking can be very slow, mainly because it has to reduce proof terms to normal forms, which may be huge. This problem is already apparent from Table 1 and

[5] Another critique of Coq's syntactic guardedness checking can be found in Barthe et al. [2].

[6] One way to think of this is by the following analogy. Our (COMPOSE) rule corresponds to the rely-guarantee parallel composition rule [8], whereas transparent lemma application within a cofix-proof corresponds to the earlier non-compositional Owicki-Gries rule [15] with the syntactic non-interference side condition.

becomes aggravated in larger developments. In our accompanying Coq code, we provide an example from an earlier project, where proof checking takes 192 seconds due to `cofix`. Replacing `cofix` with `pcofix` reduces this to 40 seconds.

- It is not at all user-friendly. The user interface does not indicate when exactly in a proof it is safe to use the coinductive assumption. Coq provides a designated command for explicitly checking guardness, but, due to the previous issue, its repeated use during the proof is often impractical.
- It interacts poorly with builtin automation tactics. They do not know about guardedness and hence very frequently produce incorrect "proofs". Usually this happens because automation applies constructors and hypotheses in the wrong order, solving the goal but causing the proof to be rejected later by the guardedness checker. To make matters worse, as a consequence of the two previous issues, it is very painful to debug such situations. Consequently, one has to be extremely careful when using automation.

As a demonstration of automation leading to a dead end, consider the following code:

```
Definition monotone0 (f: Prop → Prop) :=
  ∀ (p q: Prop), f p → (p → q) → f q.
CoInductive A f: Prop :=
  foldA (p: Prop) (LE: p → A f) (IN: f (f p)).

Goal ∀(f: Prop → Prop) (MON: monotone0 f),
  ∀ p: Prop, (p → f (f p)) → p → A f.
Proof.
  cofix CIH; intros; eapply foldA; eauto.
Qed. (* REJECTED *)
```

Here, although we explicitly apply the constructor (`foldA`) first, `eauto` somehow manages to construct an invalid proof term. We can obtain a proper proof by manually applying the coinduction hypothesis before letting `eauto` take over (this is the result of trial and error):

```
cofix CIH; intros; eapply foldA; [apply CIH|]; eauto.
```

Alternatively, we can just do the proof using our `pcofix` tactic instead of `cofix` (where `A_acc` is the corresponding accumulation lemma):

```
pcofix CIH using A_acc; eauto using foldA.
```

7.4 Paco: A Coq Library for Parameterized Coinduction

In order to make parameterized coinduction more easily applicable, we have built the Coq library *Paco* (standing for *pa*rameterized *co*induction). Paco contains internal implementations of parameterized coinduction for predicates of arity up to 15, with paco$\{n\}$ f standing for G_f for any monotone function f from predicates of arity n to predicates of arity n.

Besides the `pcofix` tactic that we have already seen in Section 7.1, the library provides tactics for folding (`pfold`) and unfolding (`punfold`) the definition of G_f, for proving monotonicity of predicates (`pmonauto`) and for simplifying hypotheses by reducing occurrences of $r \sqcup \bot$ to r (`pclearbot`). It also provides multiplication lemmas (paco$\{n\}$_mult) of the form $G_f(G_f(r)) \sqsubseteq G_f(r)$. These follow easily from Theorem 4 and monotonicity of G_f, and show that "doubly guarded" assumptions r are also simply guarded. We have found these multiplication lemmas useful for composing parameterized coinduction proofs.

To illustrate the use of our library, we build on the example of the introduction. We represent a graph by a type of nodes and a relation, R, characterizing the edges between nodes:

```
Variables (Node: Type) (R: Node → Node → Prop).
```

We now define infinite paths using parameterized coinduction. (We write `paco1` and `bot1` as the predicate is unary.)

```
Inductive step (X: Node → Prop) (x: Node) : Prop :=
 | step_intro : ∀y, R x y → X y → step X x.
Hint Constructors step.
Definition infpath :=  paco1 step bot1.
```

We also prove monotonicity of `step`, and register the corresponding lemma in the `paco` hint database that is used by `punfold`.

```
Lemma step_mon : monotone1 step. Proof. pmonauto. Qed.
Hint Resolve step_mon : paco.
```

Further, we define the predicate `path n x` to say that there exists an outgoing path of length `n` from node `x`.

```
Fixpoint path n x := match n with
   | O ⇒ True
   | S n ⇒ ∃y, R x y ∧ path n y
  end.
```

Then we can establish that if an infinite path emanates from x, then so do paths of length n, for any n. This is proved by induction on n and unfolding and inverting the definition of infinite paths.

```
Goal ∀n x, infpath x → path n x.
Proof.
  induction n; intros; simpl; auto.
  punfold H; inversion H; pclearbot; eauto.
Qed.
```

This shows that Paco-style coinductive definitions can, after unfolding, be inverted just as native Coq coinductive definitions can.

We move on to a more interesting property involving transitive closure that was suggested to us by an anonymous reviewer. The goal is to prove that if there is a predicate P holding of a node x in a graph and that whenever P holds of a node there is a non-empty path at the end of which P holds again, then there is an infinite path starting from x. The infinite path can be constructed by concatenating these non-empty paths, which is formally done by an inner induction inside a coinductive proof.

```
Goal ∀(P: Node → Prop),
     (∀ x, P x → ∃y, clos_trans_1n _ R x y ∧ P y) →
     ∀ x, P x → infpath x.
Proof.
  pcofix CIH; intros P M x Px.
  destruct (M _ Px) as (y & C & Py); clear Px.
  induction C; pfold; eauto.
Qed.
```

Using Paco, the proof is straightforward: `destruct` instantiates the second assumption and destructs the existential quantifier to expose the transitive closure, upon which an induction is later performed, whereas `clear Px` simply forgets `Px` to avoid confusing the later automation. Moreover, `pcofix` ensures that the coinductive hypothesis `CIH` is used in a semantically guarded way by construction, thereby placing no further restrictions on the proof. In contrast, carrying out this proof using Coq's builtin `cofix` tactic is surprisingly difficult because the inner inductive proof turns out to violate Coq's conservative syntactic notion of guardedness, thus necessitating ugly workarounds.

8. Discussion and Related Work

In this section, we compare to some related forms of incremental coinduction that have appeared in the literature, including the earlier versions of our construction due to Winskel [23] and Moss [14]. We conclude with some thoughts about Coq.

Local Model Checking and the "Reduction Lemma". To our knowledge, the earliest account of the parameterized greatest fixed point is in a 1989 paper by Winskel [23]. That paper, building on prior work of Larsen [10] and Stirling and Walker [21], is focused on the specific problem of "local model checking" in the modal μ-calculus—*i.e.,* deciding whether a particular state or process in a labelled transition system satisfies some recursively-defined assertion. However, in the course of attacking this specific problem, Winskel presents a generally useful construction (on power sets, but easily generalizable to lattices) that, with hindsight, we can analyze and appreciate in a more abstract way.

Winskel's key innovation is a parameterized recursive assertion, which he writes as $\nu X\{\vec{r}\}A$. One can understand this assertion as representing the greatest fixed point $\nu X.A$ under the "accumulated knowledge" $\vec{r}$, but where—and this is the key difference from our parameterized greatest fixed point—the use of the accumulated knowledge is *not guarded*. Winskel correspondingly provides the following rules for checking recursive assertions incrementally (where $p \vdash A$ denotes that process p satisfies assertion A):

$$\frac{p \in \{\vec{r}\}}{p \vdash \nu X\{\vec{r}\}A} \qquad \frac{p \notin \{\vec{r}\} \quad p \vdash A[\nu X\{p, \vec{r}\}A/X]}{p \vdash \nu X\{\vec{r}\}A}$$

Due to the *un*-guardedness embodied by the first of these rules, Winskel's parameterized ν-assertion does not support compositional reasoning along the lines of our COMPOSE rule. In particular, from $p \vdash \nu X\{q, \vec{r}\}A$ and $q \vdash \nu X\{p, \vec{r}\}A$, one cannot conclude that $p \vdash \nu X\{\vec{r}\}A$ or $q \vdash \nu X\{\vec{r}\}A$, since the premises hold trivially if $p = q$. Interestingly, a subsequent paper by Andersen, Stirling and Winskel [1], building on Winskel's parameterized ν-assertions, presents a "compositional proof system" for the modal μ-calculus, but they mean "compositional"—an admittedly overloaded term—in a very different sense from us (their "compositional" concerns the structure of *processes*).

Our parameterized greatest fixed point may be understood as a guarded version of Winskel's. Formally, Winskel's model of $\nu X\{\vec{r}\}A$ is synonymous with $\nu X.(\{\vec{r}\} \cup A)$. In the notation of our paper, this suggests the following alternative to our $\mathrm{G}_f(x)$:

$$W_f(x) \overset{\text{def}}{=} \nu z.\ x \sqcup f(z)$$

The connection to $\mathrm{G}_f(x)$, provable using a few straightforward applications of Tarski's principle, is then very simple:

$$\begin{aligned} W_f(x) &\equiv x \sqcup \mathrm{G}_f(x) \\ \mathrm{G}_f(x) &\equiv f(W_f(x)) \end{aligned}$$

For the purpose of proving soundness of local model checking, the lack of guardedness in $W_f(x)$ was not an issue, but for compositional proof development it is.

That said, the key technical result that Winskel employs in order to prove soundness of his local model checking algorithm, namely a "reduction lemma" due to Kozen [9], is in fact interderivable with our Theorem 4. The reduction lemma states, for monotone f:

$$y \sqsubseteq \nu f \iff y \sqsubseteq f(W_f(y))$$

To see the connection with Theorem 4, first observe that since $f(W_f(y)) \equiv G_f(y)$, the reduction lemma can be restated as:

$$y \sqsubseteq \nu f \iff y \sqsubseteq G_f(y)$$

And since $\nu f \equiv \mathrm{G}_f(\bot)$, the lemma can thus be seen as an instantiation of Theorem 4 where $x := \bot$. At the same time, Theorem 4 can also be seen as an instantiation of the reduction lemma! Specifically, if (given x) we instantiate the reduction lemma's f with $f(x \sqcup -)$, it yields

$$y \sqsubseteq \nu z.\ f(x \sqcup z) \iff y \sqsubseteq \nu z.\ f(x \sqcup y \sqcup z)$$

This is precisely Theorem 4.

Incremental Coinduction. In 2010, Popescu and Gunter [16] proposed a proof system for incremental coinduction, tailored towards bisimilarity in a process calculus, and they established the soundness of their system by a global, monolithic argument. Their judgment $\theta \vdash \theta'$ corresponds precisely to $\theta' \sqsubseteq \theta \sqcup \mathrm{G}_f(\theta)$ (where f is the generating function of their process bisimilarity). This suggests that our/Winskel's lattice-theoretic account of parameterized coinduction might (a) offer a simpler alternative proof of their logic's soundness, and (b) support a shallow embedding of their logic in Isabelle/HOL, which is much more efficient than a deep embedding since it reuses the proof assistant's underlying infrastructure. Finally, we note that because their proof system (like Winskel's) does not offer a way of expressing guarded entailments, it does not admit a circular compositional rule, such as (COMPOSE).

Circular Coinduction. Rather than developing a custom proof system, another approach to incremental coinduction that has been tried is to engineer a tactic that builds the full simulation relation on the fly as the interactive proof progresses [5]. The idea is to use a unification metavariable during the proof and try to show that $R_0 \subseteq ?r$ and $?r \subseteq f(?r)$. To solve the first goal, we set $?r := R_0 \cup ?r'$, where $?r'$ is a fresh metavariable. Each time we later encounter a goal $R'_0 \subseteq R_0 \cup ?r'$, where $R'_0 \not\subseteq R_0$, we instantiate $?r' := R'_0 \cup ?r''$, where $?r''$ is another fresh metavariable, and proceed. If this exploration phase ever terminates, then at the end there should be one uninstantiated $?r^{(n)}$ metavariable left, which we can instantiate to the empty relation, $\emptyset$. In this way, we will have constructed a valid postfixed point of f incrementally, and Coq can thus conclude that $R_0 \subseteq \nu f$.

Under this approach, even if the construction of the simulation relation can be done incrementally using clever tactics, the whole simulation has to eventually be constructed. As a result, this approach does not support compositional reasoning.

Coen's Principle. Isabelle's standard library (theory `Inductive`) contains an interesting coinduction principle for sets attributed to Martin Coen. Generalized to complete lattices, it reads:

$$x \sqsubseteq f(\mu y.\ f(y) \sqcup x \sqcup \nu f) \implies x \sqsubseteq \nu f \qquad \text{(COEN)}$$

This principle is strictly less useful than our parameterized coinduction. While it lets us remember the initial x for as many unfoldings of νf as desired, it does not let us accumulate further knowledge during the proof, nor does it support compositional reasoning. Interestingly, if we change the least fixed point into a greatest one, then the result, $f(\nu y.\ f(y) \sqcup x \sqcup \nu f)$, is equivalent to our G_f.

Parametric Corecursion. Moss [14] first introduced the construction that we use in this paper in a categorical setting, calling it parametric corecursion and proving an analogue of Theorem 4. He was chiefly concerned with defining elements of coinductive sets, rather than constructing proofs of coinductive predicates, and thus did not observe the compositionality of this simple construction nor its practical utility in mechanized theorem proving. More specifically, in his work, given sets X and Y of variables, a function of type $X \to \mathrm{G}_f(Y)$ models a set of equations of the form $\{\, x = \phi_x(Y) \,\}_{x \in X}$, where $\mathrm{G}_f(Y)$ is the coinductive set defined by the functor $f(Y + (-))$ (which corresponds to our G_f constructor) and ϕ_x's are formulae of some particular form related to f. Then, given a set of recursive equations of type $X \to \mathrm{G}_f(X + Y)$, the variant of Theorem 4 gives its solution for X, which is a set of non-recursive equations of type $X \to \mathrm{G}_f(Y)$. In particular, when Y is the empty set, $X \to \mathrm{G}_f(\emptyset)$ determines elements of $\mathrm{G}_f(\emptyset)$, the final coalgebra of f (*i.e.,* the coinductive set defined by f).

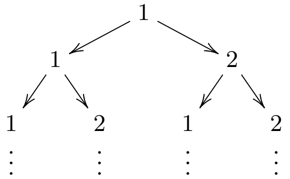
Below, we present an example showing that this construction can be useful for programming (guarded) corecursive functions. Take `pcotree` to be the set of parameterized infinite binary trees with nodes containing natural numbers, defined as follows:

```
CoInductive pcotree (R: Type) : Type :=
| tcons (a: nat) (tl: pcotree R + R) (tr: pcotree R + R).
```

We can then define the coiterator combinator `coitr` of the following type using the construction above (see the website for details):

```
coitr: ∀L R, (L → cotree (L + R)) → L → cotree R
```

Now we define the infinite tree `fliptree` containing 1 on every left child and 2 on every right child, depicted below:



Using `coitr`, we can write the corecursive definition of `fliptree` without using **CoFixpoint**, thus avoiding guardedness checking:

```
Definition fliptree : cotree False :=
  coitr (fun onetree ⇒ tcons 1
    (inr (inl onetree))
    (inl (coitr (fun twotree ⇒ tcons 2
            (inr (inr (inl onetree)))
            (inr (inl twotree))) I))) I.
```

Nice as these examples are, we believe that the more important use of Moss's categorical construction is in building *proofs* incrementally and compositionally using tactics, as we have sought to demonstrate in this paper.

Application to Agda. We remark that the direct internal approach also works for Agda and Coq's indexed sets, because Lemmas 2–3 and Theorem 4 can be generalized to a lattice where the greatest fixed point $\mathrm{G}_f(x)$ exists for a particular f (*i.e.,* it does not have to exist for all f as in a complete lattice). We can easily show that indexed sets form (non-complete) lattices, as they have a natural order, finite products and coproducts. Also, for a strictly positive function f, we can define $\mathrm{G}_f(x)$ using Agda's ∞ constructor (or Coq's **CoInductive**), which gives a greatest fixed point of f on this lattice, and we can thus reason about statements of the form $y \sqsubseteq \mathrm{G}_f(x)$ using our principle. In this setting, however, we cannot use Mendler-style recursion because it requires impredicative quantification.

Coinduction vs Induction in Coq. It is instructive to contrast Coq's support for induction and coinduction. In both cases, Coq provides a built-in (co-)recursion combinator that comes with a syntactic guardedness condition. As we have seen in Section 7.3, it is possible (but quite inconvenient) to use corecursion directly inside a proof via the `cofix` tactic.

For each inductively defined type, Coq additionally generates induction lemmas (proved using the aforementioned recursion combinator), which semantically enforce guardedness. These lemmas can conveniently be applied using the `induction` tactic, rendering the use of the low-level recursion combinator inside proofs unnecessary for most users.

For coinductively defined types, however, Coq does not generate any lemmas, nor does it provide a "`coinduction`" tactic analogous to its `induction` tactic. Our work can be seen as filling this gap by providing lemmas such as `G_unfold` and `G_acc` and the tactic `pcofix`. It is also worth pointing out that, unlike the Coq-generated principles for induction, ours are *complete*: whatever can be proved using `cofix` can also be proved using `G_acc` & co.

Acknowledgements

We would like to give special thanks to Deepak Garg for alerting us to the connection with Winskel's work. We would also like to thank Andreas Abel, Fritz Henglein, Neel Krishnaswami, Andy Pitts, and Glynn Winskel for helpful discussions, as well as the anonymous reviewers for their constructive feedback.

References

[1] H. R. Andersen, C. Stirling, and G. Winskel. A compositional proof system for the modal μ-calculus. In *LICS*, pages 144–153. IEEE Computer Society, 1994.

[2] G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Comp. Sci.*, 14(1):97–141, Feb. 2004.

[3] E. Giménez. Codifying guarded definitions with recursive schemes. In *Types for Proofs and Programs*, volume 996 of *LNCS*, pages 39–59. Springer, 1995.

[4] A. D. Gordon. Bisimilarity as a theory of functional programming. *Theoretical Computer Science*, 228(1-2):5–47, 1999.

[5] D. Hausmann, T. Mossakowski, and L. Schroeder. Iterative circular coinduction for CoCasl in Isabelle/HOL. In *FASE*, volume 3442 of *LNCS*, pages 341–356. Springer, 2005.

[6] C.-K. Hur. Heq: a Coq library for heterogeneous equality, 2010. Presented at Coq-2 workshop.

[7] C.-K. Hur, D. Dreyer, G. Neis, and V. Vafeiadis. The marriage of bisimulations and Kripke logical relations. In *POPL*, 2012.

[8] C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.

[9] D. Kozen. Results on the propositional μ-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.

[10] K. G. Larsen. Proof systems for Hennessy-Milner logic with recursion. In *CAAP*, volume 299 of *LNCS*, pages 215–230. Springer, 1988.

[11] R. Matthes. Recursion on nested datatypes in dependent type theory. In *Computability in Europe (CiE)*, volume 5028 of *LNCS*, pages 431–446. Springer, 2008.

[12] N. P. Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(1-2):159 – 172, 1991.

[13] R. Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, 1999.

[14] L. S. Moss. Parametric corecursion. *Theor. Comput. Sci.*, 260(1-2):139–163, June 2001.

[15] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[16] A. Popescu and E. L. Gunter. Incremental pattern-based coinduction for process algebra and its Isabelle formalization. In *FOSSACS*, pages 109–127, 2010.

[17] D. Sangiorgi. On the bisimulation proof method. *Mathematical Structures in Comp. Sci.*, 8(5):447–479, Oct. 1998.

[18] D. Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2011.

[19] D. Sangiorgi and J. Rutten. *Advanced Topics in Bisimulation and Coinduction*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2011.

[20] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. Relaxed-memory concurrency and verified compilation. In *POPL*, 2011.

[21] C. Stirling and D. Walker. Local model checking in the modal mu-calculus. In *TAPSOFT, Vol.1 (CAAP)*, volume 351 of *LNCS*, pages 369–383. Springer, 1989.

[22] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5(2):285–309, 1955.

[23] G. Winskel. A note on model checking the modal ν-calculus. In *ICALP*, volume 372 of *LNCS*, pages 761–772. Springer, 1989.