

Choice Trees: Representing and Reasoning About Nondeterministic, Recursive, and Impure Programs in Coq (draft)

NICOLAS CHAPPE

Univ Lyon, ENS Lyon, UCBL, CNRS, Inria, LIP, F-69342, LYON Cedex 07, France
e-mail: nicolas.chappe@ens-lyon.fr

PAUL HE

University of Pennsylvania
e-mail: paulhe@cis.upenn.edu

LUDOVIC HENRIO

Univ Lyon, ENS Lyon, UCBL, CNRS, Inria, LIP, F-69342, LYON Cedex 07, France
e-mail: ludovic.henrio@cnrs.fr

ELEFThERIOS IOANNIDIS

University of Pennsylvania
e-mail: elefthei@seas.upenn.edu

YANNICK ZAKOWSKI

Univ Lyon, ENS Lyon, UCBL, CNRS, Inria, LIP, F-69342, LYON Cedex 07, France
e-mail: yannick.zakowski@inria.fr

STEVE ZDANCEWIC

University of Pennsylvania
e-mail: stevez@cis.upenn.edu

Abstract

This paper introduces Choice Trees (CTrees), a monad for modeling nondeterministic, recursive, and impure programs in Coq. Inspired by Xia et al.'s ITrees, this novel data structure embeds computations into coinductive trees with three kinds of nodes: external events, internal steps, and nondeterministic branching. This structure allows us to provide shallow embedding of denotational models with internal choice in the style of ccs, while recovering an inductive LTS view of the computation. CTrees inherit a vast collection of bisimulation and refinement tools, with respect to which we establish a rich equational theory.

We connect CTrees to the ITrees infrastructure by showing how a monad morphism embedding the former into the latter permits using CTrees to implement nondeterministic effects. We demonstrate the utility of CTrees by using them to model concurrency semantics in two case studies: ccs and cooperative multithreading.

Key Words: Nondeterminism, Formal Semantics, Interaction Trees, Concurrency

1 Introduction

Reasoning about and modeling nondeterministic computations is important for many purposes. Formal specifications use nondeterminism to abstract away from the details of implementation choices. Accounting for nondeterminism is crucial when reasoning about the semantics of concurrent and distributed systems, which are, by nature, nondeterministic due to races between threads, locks, or message deliveries. Consequently, precisely defining nondeterministic behaviors and developing the mathematical tools to work with those definitions has been an important research endeavor, and has led to the development of formalisms like nondeterministic automata, labeled transition systems and relational operational semantics (Bergstra et al., 2001), powerdomains (Smyth, 1976), or game semantics (Abramsky and Melliès, 1999; Rideau and Winskel, 2011), among others, all of which have been used to give semantics to nondeterministic programming language features such as concurrency (Sangiorgi and Walker, 2001; Milner, 1989; Harper, 2016).

In this paper, we are interested in developing tools for modeling nondeterministic computations in a dependent type theory such as Coq’s CIC (Team, 2022). Although any of the formalisms mentioned above could be used for such purposes (and many have been (Sevcík et al., 2013; Kang et al., 2017; Lee et al., 2020; Koenig and Shao, 2020; Oliveira Vale et al., 2022)), those techniques offer various tradeoffs when it comes to the needs of formalization: automata, and labeled transitions systems, while offering powerful bisimulation proof principles, are not easily made modular (except, perhaps, with complex extensions to the framework (Henrio et al., 2016)). Relationally-defined operational semantics are flexible and expressive, but again suffer from issues of compositionality, which makes it challenging to build general-purpose libraries that support constructing complex models. Conversely, powerdomains and game semantics are more denotational approaches, aiming to ensure compositionality by construction; however, the mathematical structures involved are themselves very complex, typically involving many relations and constraints (Abramsky and Melliès, 1999; Melliès and Mimram, 2007; Rideau and Winskel, 2011) that are not easy to implement in constructive logic (though there are some notable exceptions (Koenig and Shao, 2020; Oliveira Vale et al., 2022)). Moreover, in all of the above-mentioned approaches, there are other tensions at play. For instance, how “deep” the embedding is affects the amount of effort needed to implement a formal semantics—“shallower” embeddings typically allow more re-use of metalanguage features, e.g., meta-level function application can obviate the need to define and prove properties about a substitution operation; “deeper” embeddings can side-step meta-level limitations (such as Coq’s insistence on pure, total functions) at the cost of additional work to define the semantics. Moreover, these tradeoffs can have significant impact on how difficult it is to use other tools and methodologies: for instance, to use QuickChick (Lampropoulos and Pierce, 2018), one must be able to extract an executable interpreter from the semantics, something that isn’t always easy or possible.

This paper introduces a new formalism designed specifically to facilitate the definition of and reasoning about nondeterministic computations in Coq’s dependent type theory. The key idea is to update Xia, et al.’s *interaction trees* (ITrees) framework (Xia et al., 2020) with native support for nondeterministic “choice nodes” that represent internal choices made

93 during computation. The main technical contributions of this paper are to introduce the
94 definition of these CTrees (“choice trees”) and to develop the suitable metatheory and
95 equational reasoning principles to accommodate that change.

96 We believe that CTrees offer an appealing, and novel, point in the design space of
97 formalisms for working with nondeterministic specifications within type theory. Unlike
98 purely relational specifications, CTrees build nondeterminism explicitly into a datatype,
99 as nodes in a tree, and the nondeterminism is realized propositionally at the level of the
100 equational theory, which determines when two CTree computations are in bisimulation.
101 This means that the user of CTrees has more control over how to represent nondeterminism
102 and when to apply the incumbent propositional reasoning. By reifying the choice construct
103 into a data structure, one can write meta-level functions that manipulate CTrees, rather
104 than working entirely within a relation on syntax. This design allows us to bring to bear
105 the machinery of monadic interpreters to refine the nondeterminism into an (executable)
106 implementation. While ITrees can represent such choice nodes, in our experience, using
107 that feature to model “internal” nondeterminism is awkward: the natural equational theory
108 for ITrees is too fine, and other techniques, such as interpretation into *Prop*, don’t work out
109 neatly.


110 At the same time, the notion of bisimulation for CTrees is still connected to familiar
111 definitions like those from labeled transition systems (LTS), meaning that much of the
112 well-developed theory from prior work can be imported whole-sale. Indeed, we define
113 bisimilarity for CTrees by viewing them as LTSs and applying standard definitions. The
114 fact that the definition of bisimulation ends up being subtle and nontrivial is a sign that
115 we gain something by working with the CTrees: like their ITree predecessors, CTrees have
116 compositional reasoning principles, the type is a monad, and the useful combinators for
117 working with ITrees, namely sequential composition, iteration and recursion, interpretation,
118 etc., all carry over directly. CTrees, though, further allow us to conveniently, and flexibly,
119 define nondeterministic semantics, ranging from simple choice operators to various flavors
120 of parallel composition. The benefit is that, rather than just working with a “raw” LTS
121 directly, we can construct one using the CTrees combinators—this is a big benefit because,
122 in practice, the LTS defining the intended semantics of a nondeterministic programming
123 language cannot easily be built in a compositional way without using some kind of inter-
124 mediate representation, which is exactly what CTrees provides (see the discussion about
125 Figure 4). A key technical novelty of our CTrees definition is that it makes a distinction
126 between *stepping* choices (which correspond to τ transitions and introduce new LTS states)
127 and *delayed* choices (which don’t correspond to a transition and don’t create a state in the
128 LTS). This design allows for compositional construction of the LTS and generic reasoning
129 rules that are usable in any context.

130 The net result of our contributions is a library, entirely formalized in Coq, that offers
131 flexible building blocks for constructing nondeterministic, and hence concurrent, models
132 of computation. To demonstrate the applicability of this library, we use it to implement
133 the semantics from two different formalisms: ccs (Milner, 1989) and a language with
134 cooperative threads inspired from the literature (Abadi and Plotkin, 2010). Crucially, in both
135 of these scenarios, we are able to define the appropriate parallel composition combinators
136 such that the semantics of the programming language can be defined fully compositionally
137 (i.e., by straightforward induction on the syntax). Moreover, we recover the classic definition
138

of program equivalence for ccs directly from the equational theory induced by the encoding of the semantics using CTrees; for the language with cooperative threading, we prove some standard program equivalences.

To summarize, this paper makes the following contributions:

- We introduce CTrees, a novel data structure for defining nondeterministic computations in type theory, along with a set of combinators for building semantic objects using CTrees.
- We develop the appropriate metatheory needed to reason about strong and weak bisimilarity of CTrees, connecting their semantics to concepts familiar from labeled transition systems.
- We show that CTrees admit appropriate notions of refinement and that we can use them to construct monadic interpreters; we show that ITrees can be faithfully embedded into CTrees.
- We demonstrate how to use CTrees in two case studies: (1) to define a semantics for Milner’s classic ccs and prove that the resulting derived equational theory coincides with the one given by the standard operational semantics, and (2) to model in stages cooperative multithreading with support for fork and yield operations and prove nontrivial program equivalences.

All of our results have been implemented in Coq, and all claims in this paper are fully mechanically verified. For expository purposes, we stray away from Coq’s syntax in the body of this paper, but systematically link our claims to their formal counterpart via hyperlinks represented as .

The remainder of the paper is organized as follows. The next section gives some background about interaction trees and monadic interpreters, along with a discussion of the challenges of modeling nondeterminism, laying the foundation for our results. We introduce the CTrees data structure and its main combinators in Section 3. Section 4 introduces several notions of program equivalence and comparison over CTrees—(coinductive) equality, strong bisimilarity, strong similarity, complete similarity, weak bisimilarity, and trace equivalence—and describes its core equational theory. Section 5 gives another view on strong bisimilarity and strong similarity of CTrees, enabling new proof techniques. Section 6 describes how to interpret uninterpreted events in an ITree into “choice” branches in a CTree, as well as how to define the monadic interpretation of events from CTrees. Section 7 describes our first case study, a model for ccs. Section 8 describes our second case study, a model for the imp language extended with cooperative multithreading. Finally, Section 9 discusses related work and concludes.

This journal paper is a follow-up to the one published in POPL’23 (Chappe et al., 2023), in which we had introduced CTrees and their usage. The present version is extensively updated and enriched to describe the current reimplementations of our library. In particular, we present updated core definitions of CTrees to improve their usability and applicability; we develop additional notions of program equivalence and refinements over CTrees in Section 4; we define a novel alternate characterization of our strong (bi)simulation and illustrate how it eases some reasoning in Section 5; we leverage this updated theory to extend our results related to interpretation and refinement, as well as improve our model of

```

185 CoInductive itree (E: Type → Type) (R: Type) : Type :=
186   (* computation terminating with value r *)
187   | Ret (r: R)
188   (* event e yielding an answer in A *)
189   | Vis {A: Type} (e : E A) (k : A → itree E R)
190   (* "silent tau" transition with child t *)
191   | Step (t: itree E R).

```

Fig. 1: Interaction trees: definition

cooperative scheduling. We further discuss some trade-offs in the design of such shallow structures for divergence and non-determinism in Section 9.

2 Background

2.1 Interaction trees and monadic interpreters

Monadic interpreters have grown to be an attractive way to mechanize the semantics of a wide class of computational systems in dependent typed theory, such as the one found in many proof assistants, for which the host language is purely functional and total. In the Coq ecosystem, interaction trees (Xia et al., 2020) provide a rich library for building and reasoning about such monadic interpreters. By building upon the free(r) monad (Kiselyov and Ishii, 2015; Letan et al., 2018), one can both design highly reusable components, as well as define modular models of programming languages more amenable to evolution. By modeling recursion coinductively, in the style of Capretta’s delay monad (Capretta, 2005; Altenkirch et al., 2017), such interpreters can model non-total object languages while retaining the ability to *extract* correct-by-construction, executable, reference interpreters. By generically lifting monadic implementations of effects into a monad homomorphism, complex interpreters can be built by stages, starting from an initial structure where all effects are free and incrementally introducing their implementation. Working in a proof assistant, these structures are well suited for reasoning about program equivalence and program refinement: each monadic structure comes with its own notion of refinement, and the layered infrastructure gives rise to increasingly richer equivalences (Yoon et al., 2022), starting from the free monad, which comes with no associated algebra.

Interaction trees are coinductive data structures for representing (potentially divergent) computations that interact with an external environment through *visible events*. A definition of the `ITree` datatype is shown in Figure 1.¹ The datatype takes as its first parameter a signature—described as a family of types $E : \mathbf{Type} \rightarrow \mathbf{Type}$ —that specifies the set of interactions the computation may have with the environment. The `Vis` constructor builds a node in the tree representing such an interaction, followed by a continuation indexed by the return type of the event. The second parameter, R , is the *result type*, the type of values that the entire computation may return, if it halts. The constructor `Ret` builds such a pure computation, represented as a leaf. Finally, the `Step` constructor models

¹ The definition is presented with a positive coinductive datatype for expository purposes. The actual implementation is defined in the negative style.

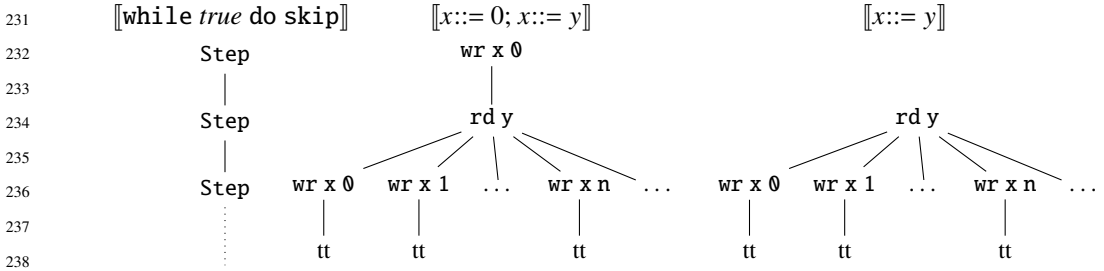


Fig. 2: Example ITrees denoting the `imp` programs p_1 , p_2 , and p_3 .

an internal, non-observable step of computation, allowing the representation of silently diverging computations; it is also used for guarding corecursive definitions.²

To illustrate the approach supported by ITrees, and motivate the contributions of this paper, we consider how to define the semantics for a simple imperative programming language, `imp`:

$$\text{comm} \triangleq \text{skip} \mid x ::= e \mid c1; c2 \mid \text{while } b \text{ do } c$$

The language contains a `skip` construct, assignments, sequential composition, and loops—we assume a simple language of expressions, e , that we omit here. Consider the following `imp` programs:

$$p_1 \triangleq \text{while } \text{true} \text{ do skip} \quad p_2 \triangleq x ::= 0; x ::= y \quad p_3 \triangleq x ::= y$$

Following a semantic model for `imp` built on ITrees in the style of Xia et al. (2020), one builds a semantics in two stages. First, commands are represented as monadic computations of type `itree MemE unit`: commands do not return values, so the return type of the computation is the trivial `unit` type; interactions with the memory are (at first) left uninterpreted, as indicated by the event signature `MemE`. This signature encodes two operations: `rd` yields a value, while `wr` yields only the acknowledgment that the operation took place, which we encode again using `unit`.

```
Variant MemE : Type → Type :=
  | rd (x : var)           : MemE value
  | wr (x : var) (v : value) : MemE unit
```

Indexing by the `value` type in the continuation of `rd` events gives rise to non-unary branches in the tree representing these programs. For instance, the programs p_1 , p_2 , p_3 are, respectively, modeled at this stage by the trees shown in Figure 2. These diagrams omit the `Vis` and `Ret` constructors, because their presence is clear from the picture. For example, the second tree p_2 would be written as

$$\text{Vis } (\text{wr } x \ 0) \ (\lambda _ \Rightarrow \text{Vis } (\text{rd } y) \ (\lambda \text{ ans} \Rightarrow (\text{Vis } (\text{wr } x \ \text{ans}) \ (\lambda _ \Rightarrow \text{Ret } \text{tt}))))).$$

The `Step` nodes in the first tree are the guards from Capretta’s monad: because the computation diverges silently, it is modeled as an infinite sequence of such guards. The

² The ITree library uses `Tau` to represent `Step` nodes. `Tau` and τ are overloaded in our context, so we rename it to `Step` here to avoid ambiguity and unify the notations with the `Ctree` ones.

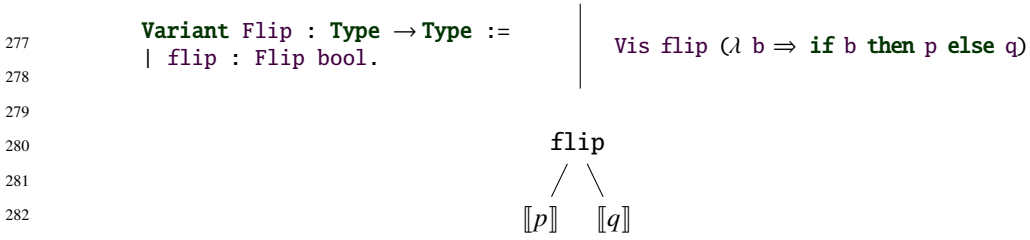


Fig. 3: A boolean event, an example of its use, and the corresponding CTree.

equivalence used for computations in the `ITree` monad is a *weak bisimulation*, dubbed *equivalence up-to taus* (`eutt`), which allows one to ignore finite sequences of `Step` nodes when comparing two trees. It remains termination-sensitive: the silently diverging computation is not equivalent to any other `ITree`.

With `ITrees`, no assumption about the semantics of the uninterpreted memory events is made. Although one would expect p_2 and p_3 to be equivalent as `imp` programs, their trees are not `eutt` since the former starts with a different event than the latter. This missing algebraic equivalence is concretely recovered at the second stage of modeling: `imp` programs are given a semantics by interpreting the trees into the state monad, by *handling* the `MemE` events. This yields computations in `stateT mem (itree voidE) unit`, or, unfolding the definition of `stateT, mem → itree voidE (mem * unit)`. Here, `voidE` is the “empty” event signature, such that an `ITree` at that type either silently diverges or deterministically returns an answer. For p_2 and p_3 , assuming an initial state m , the computations become (writing the trees horizontally to save space):

$$\begin{aligned} \text{interp } h_{\text{mem}} \llbracket p_2 \rrbracket m &= \text{Step} - \text{Step} - \text{Step} - (m\{x \leftarrow 0\}\{x \leftarrow m(y)\}, tt) \\ \text{interp } h_{\text{mem}} \llbracket p_3 \rrbracket m &= \text{Step} - \text{Step} - (m\{x \leftarrow m(y)\}, tt) \end{aligned}$$

The `Step` nodes are introduced by the interpretation of the memory events. More precisely, an `interp` combinator applies the handler h_{mem} to the `rd` and `wr` nodes of the trees, implementing their semantics in terms of the state monad. Assuming an appropriate implementation of the memory, one can show that $m\{x \leftarrow 0\}\{x \leftarrow m(y)\}$ and $m\{x \leftarrow m(y)\}$ are extensionally equal, and hence p_2 and p_3 are `eutt` after interpretation.

2.2 Nondeterminism

While the story above is clean and satisfying for stateful effects, nondeterminism is much more challenging. Suppose we extend `imp` with a branching operator `br p or q` whose semantics is to nondeterministically pick a branch to execute. This new feature is modeled very naturally using a boolean-indexed `flip` event, creating a binary branch in the tree. The new event signature, a sample use, and the corresponding tree are shown on Figure 3

Naturally, as with memory events, `flip` does not come with its expected algebra: associativity, commutativity and idempotence. To recover these necessary equations to establish program equivalences such as $p_3 \equiv \text{br } p_2 \text{ or } p_3$, we need to find a suitable monad to interpret `flip` into.

323 [Zakowski et al.](#) used this approach in the Vellvm project ([Zakowski et al., 2021](#)) for
 324 formalizing the nondeterministic features of the LLVM IR. Their model consists of a
 325 propositionally-specified set of computations: ignoring other effects, the monad they use is
 326 $\text{itree } E _ \rightarrow \text{Prop}$. The equivalence they build on top of it essentially amounts to a form of
 327 bijection up-to equivalence of the contained monadic computations. However, this approach
 328 suffers from several drawbacks. First, one of the monadic laws is broken: the `bind` operation
 329 does not associate to the left. Although stressed in the context of Vellvm ([Zakowski et al.,](#)
 330 [2021](#)) and Yoon et al.’s work on layered monadic interpreters (Yoon et al., [2022](#)), this
 331 issue is not specific to ITrees but rather to a hypothetical “Prop Monad Transformer”,
 332 i.e. to $\lambda M X \Rightarrow M X \rightarrow \text{Prop}$, as pointed out previously in (Maillard et al., [2020](#)). The
 333 definition is furthermore particularly difficult to work with. Indeed, the corresponding
 334 monadic equivalence is a form of bijection up-to setoid: for any trace in the source, we must
 335 existentially exhibit a suitable trace in the target. The inductive nature of this existential is
 336 problematic: one usually cannot exhibit upfront a coinductive object as witness, they should
 337 be produced coinductively. This challenge is particularly apparent in (Beck et al., [2024](#))
 338 when proving that the change in perspective from an infinite domain of memory addresses
 339 to a finite one is sound.

340 Second, the approach is very much akin to identifying a communicating system with
 341 its set of traces, except using a richer structure, namely monadic computations, instead of
 342 traces: it forgets all information about *when* nondeterministic choices are made. As has been
 343 well identified by the process calculi tradition, and while trace equivalence is sometimes
 344 the desired relation, such a model leads to equivalences of programs that are too coarse to
 345 be compositional in general. In a general purpose semantics library, we believe we should
 346 strive to provide as much compositional reasoning as possible and thus our tools support
 347 both trace equivalence, and bisimulations (Bloom et al., [1988](#)).

348 Third, because the set of computations is captured propositionally, this interpretation
 349 is incompatible with the generation of an *executable* interpreter by extraction, losing one
 350 of the major strengths of the ITree framework. [Zakowski et al.](#) work around this difficulty
 351 by providing two interpretations of their nondeterministic events, and formally relating
 352 them. But this comes at a cost — the promise of a sound interpreter for free is broken —
 353 and with constraints — non-determinism must come last in the stack of interpretations,
 354 and combinators whose equational theory is sensible to nondeterminism are essentially
 355 impossible to define.

356 This difficulty with properly tackling nondeterminism extends also to concurrency. [Lesani](#)
 357 [et al.](#) used ITrees to prove the linearizability of concurrent objects (Lesani et al., [2022](#)).
 358 Here too, they rely on sets of linearized traces and consider their interleavings. While a
 359 reasonable solution in their context, that approach strays from the monadic interpreter style
 360 and fails to capture bisimilarity.

361 This paper introduces CTrees, a suitable monad for modeling nondeterministic effects.
 362 As with ITrees, the structure is compatible with divergence, external interaction through
 363 uninterpreted events, extraction to executable reference interpreters, and monadic interpre-
 364 tation. The core intuition is based on the observation that the tree-like structure from ITrees
 365 is indeed the right one for modeling nondeterminism. The problem arises from how the
 366 ITrees definition of `eut t` observes which branch is taken, requiring that *all* branches be
 367 externally visible: while appropriate to model nondeterminism that results from a lack of
 368

information, it does not correspond to true *internal* choice. Put another way, thinking of the trees as labeled transition systems, ITrees are *deterministic*. With CTrees, we therefore additionally consider truly branching nodes, explicitly build the associated nondeterministic LTS, and define proper bisimulations on the structure.

The resulting definitions are very expressive. As foreseen, they form a proper monad, validating all monadic laws up-to coinductive structure equality, they allow us to establish desired `imp` equations such as $p_3 \equiv \text{br } p_2 \text{ or } p_3$, but they also scale to model ccs and cooperative multithreading.

Before getting to that, and to better motivate our definitions, let us further extend our toy language with a `block` construction that cannot reduce, and a `print` instruction that simply prints a dot. We will refer to this language as `ImpBr`.

`comm` \triangleq `skip` | `x ::= e` | `c1; c2` | `while b do c` | `br c1 or c2` | `block` | `print`

Consider the program $p \triangleq \text{br } (\text{while } \text{true} \text{ do } \text{print}) \text{ or } \text{block}$. Depending on the intended operational semantics associated with `br`, this program can have one of two behaviors: (1) either to always reactively print an infinite chain of dots, or (2) to become nondeterministically either similarly reactive, or completely unresponsive.

When working with (small-step) operational semantics, the distinction between these behaviors is immediately apparent in the reduction rule for `br` (we only show rules for the left branch here).

$$\frac{}{\text{br } c_1 \text{ or } c_2 \rightarrow c_1} \text{BRINTERNAL} \qquad \frac{c_1 \rightarrow c'_1}{\text{br } c_1 \text{ or } c_2 \rightarrow c'_1} \text{BRDELAYED}$$

`BRINTERNAL` specifies that `br` may simply reduce to the left branch, while `BRDELAYED` specifies that `br` can reduce to any state reachable from the left branch. From an observational perspective, the former situation describes a system where, although we do not observe which branch has been taken, we do observe that a branch has been taken. On the contrary, the latter only progresses if one of the branches can progress, we thus directly observe the subsequent evolution of the chosen branch, but not the branching itself.

In order to design the right monadic structure allowing for enough flexibility to model either behavior, it is useful to look ahead and anticipate how we will reason about program equivalence, as described in detail in Section 4. The intuition we follow is to interpret our computations as labeled transition systems and define bisimulations over those, as is generally done in the process algebra literature. From this perspective, the `imp` program p may correspond to three distinct LTSs depending on the intended semantics, as shown in Figure 4.

Figure 4a describes the case where picking a branch is an unknown external event, hence where taking a specific branch is an *observable* action with a dedicated label: this situation is naturally modeled by a `Vis` node in the style of ITrees, that is $\llbracket \text{br } p \text{ or } q \rrbracket \triangleq \text{Vis flip } \lambda b \cdot \text{if } b \text{ then } \llbracket p \rrbracket \text{ else } \llbracket q \rrbracket$.

Figure 4b corresponds to `BRINTERNAL`: both the stuck and the reactive states are reachable, but we do not observe the label of the transition. This transition exactly corresponds to the internal τ step of process algebra. This situation could³ be captured by introducing a

³ We will see shortly that these nodes are actually an encoding in the implementation.

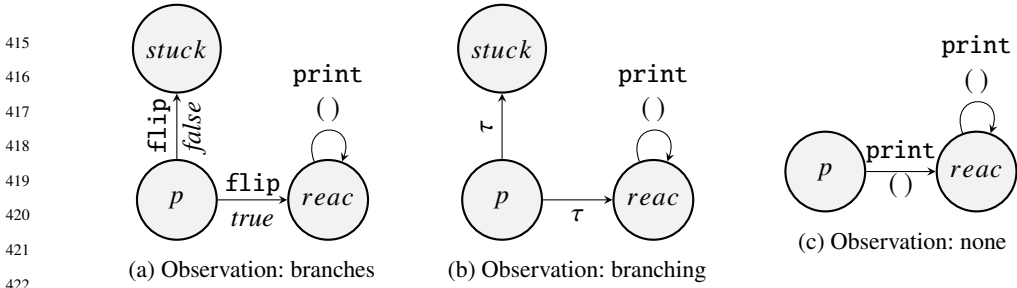


Fig. 4: Three possible semantics for the program p , from an LTS perspective

new kind of node in our data structure, a Br_S branch, that maps in our bisimulations defined in Section 4 to a nondeterministic *internal step*. For this semantics, we thus have $\llbracket \text{br } p \text{ or } q \rrbracket \triangleq Br_S^2 \lambda b \cdot \text{if } b \text{ then } \llbracket p \rrbracket \text{ else } \llbracket q \rrbracket$.⁴

Figure 4c corresponds to Br_{DELAYED} but raises the question: how do we build such a behavior? A natural answer can be to assume that it is the responsibility of the model, i.e., the function mapping imp 's syntax to the semantic domain, $CTrees$, to explicitly build this LTS. Here, $\llbracket p \rrbracket$ would be an infinite sequence of Vis print nodes, containing no other node. While that would be convenient for developing the meta-theory of $CTrees$, this design choice would render them far less compositional (and hence less useful) than we want them to be. Indeed, we want our models to be defined as computable functions by recursion on the syntax, whenever possible. But to build this LTS directly, the model for $\text{br } p \text{ or } q$ needs to introspect the models for $\llbracket p \rrbracket$ and $\llbracket q \rrbracket$ to decide whether they can take a step, and hence whether it should introduce a branching node. But, in general, statically determining whether the next reachable instruction is semantically equivalent to block is intractable, so that introspection will be hard (or impossible) to implement. We thus extend $CTrees$ with a third category of nodes, a Br node, which does not directly correspond to states of an LTS. Instead, Br nodes aggregate sub-trees such that the (inductively reachable) Br children of a Br node are “merged” in the LTS view of the $CTree$. This design choice means that, for the Br_{DELAYED} semantics, the model is again trivial to define: $\llbracket \text{br } p \text{ or } q \rrbracket \triangleq Br^2 \lambda b \cdot \text{if } b \text{ then } \llbracket p \rrbracket \text{ else } \llbracket q \rrbracket$, but the definition of bisimilarity for $CTrees$ ensures that the behavior of $\llbracket p \rrbracket$ is precisely the LTS in Figure 4c.

Although Br_{DELAYED} and its corresponding LTS in Figure 4c is *one* example in which Br nodes are needed, we will see another concrete use in modeling ccs in Section 7. Similar situations arise frequently, for modeling mutexes, locks, and other synchronization mechanisms (e.g., the “await” construct in Encore (Brandauer et al., 2015)), dealing with crash failures in distributed systems, encoding relaxed-consistency shared memory models (e.g., the THREADPROMISE rule of promising semantics (Kang et al., 2017)). The Br construct is needed whenever the operational semantics includes a rule whose possible transitions depend on the existence of other transitions, i.e., for any rule of the following shape, typically when it may be the case that $P \rightarrow$:

$$\frac{P \rightarrow P'}{C[P] \rightarrow C[P']} \text{CONTINGENTSTEP}$$

⁴ The 2 indicates the arity of the branching.

```

461 (* Core datatype *)
462 CoInductive ctree (E B : Type → Type) (R : Type) :=
463   | Ret (r : R) (* pure computation *)
464   | Stuck (* stuck process *)
465   | Step (t : ctree) (* internal guard *)
466   | Guard (t : ctree) (* invisible guard *)
467   | Vis {X : Type} (e : E X) (k : X → ctree) (* external event *)
468   | Br {X : Type} (c : B X) (k : X → ctree) (* delayed branching *)
469
470 (* Bind, sequencing computations *)
471 CoFixpoint bind {E T U} (t : ctree E B T) (k : T → ctree E B U)
472   : ctree E B U :=
473   match u with
474   | Ret r ⇒ k r
475   | Stuck ⇒ Stuck
476   | Step t ⇒ Step (bind t k)
477   | Guard t ⇒ Guard (bind t k)
478   | Vis e h ⇒ Vis e (λ x ⇒ bind (h x) k)
479   | Br b h ⇒ Br b (λ x ⇒ bind (h x) k)
480   end
481
482 (* Stepping branching *)
483 Definition BrS {X : Type} (c : B X) (k : X → ctree) :=
484   Br c (λ x ⇒ Step (k x))
485
486 (* Main fixpoint combinator *)
487 CoFixpoint iter {I : Type} (body : I → ctree E B (I + R))
488   : I → ctree E B R :=
489   bind (body i) (λ lr ⇒ match lr with
490     | inr r ⇒ Ret r
491     | inl i ⇒ Guard (iter body i)
492     end)
493
494 Notation "E ~> F" := (∀ X, E X → F X)
495 (* Atomic ctree triggering a single event *)
496 Definition trigger : E ~> ctree E B := λR (e : E R) ⇒ Vis e (λ x ⇒ Ret x)
497 (* Atomic branching ctrees *)
498 Definition branchS : ctree E B R := λc ⇒ BrS c (λ x ⇒ Ret x)
499 Definition branch : ctree E B R := λc ⇒ Br c (λ x ⇒ Ret x)

```

Fig. 5: CTrees: definition and core combinators (👁️)

The point is that to implement the LTS corresponding to `CONTINGENTSTEP` without using a `Br` node would require introspection of P to determine whether it may step, which is potentially non-computable. The `Br` node bypasses the need for that introspection at *representation time*, instead delaying it to the characterization of the CTree as an LTS, which is used only for *reasoning* about the semantics. The presence of the `Br` nodes retains the constructive aspects of the model, in particular the ability to *interpret* these nodes at later stages, for instance to obtain an executable version of the semantics.

3 CTrees: Definition and Combinators

3.1 Core definitions

We are now ready to define our core datatype, displayed in the upper part of Figure 5. The definition remains close to a coinductive implementation of the free monad, but hard-codes support for an additional effect: unobservable, nondeterministic branching. The CTree datatype, much like an ITree, is parameterized by a signature of (external) events E encoded as a family of types, and a return type R . Contrary to ITree, it is parameterized by a second family of types B characterizing the allowed arities of branching. CTrees are coinductive tree⁵ with four main kinds of nodes: pure computations (**Ret**), external events (**Vis**), unary node with an implicitly associated τ step (**Step**), and delayed internal branching (**Br**). Nullary (**Stuck**) and unary (**Guard**) delayed internal branching could be expressed as special cases of *Br* over an appropriate interface B , but we provide specific constructors for them for convenience given their central role. The continuation following external events is indexed by the return type specified by the emitted event. Similarly, the continuation following internal branching is indexed by the return type specified by the emitted branch. When using finite branching in examples, we assume that we are implicitly parameterized by a suitable interface B supporting such finite indexed types, and we abuse notations and write, for instance, $Br^2 t u$ for a computation branching on a finite type with two inhabitants, rather than explicitly spelling out an event with a boolean signature and the continuation that branches on the boolean index: $\lambda i \Rightarrow \mathbf{match\ } i \mathbf{ with\ } 0 \Rightarrow t \mid 1 \Rightarrow u \mathbf{ end}$. We write **Stuck** nodes as “ \emptyset ” in equations.

The remainder of Figure 5 displays (superficially simplified) definitions of the core combinators. In particular, guarded branching, Br_S , as informally introduced in Section 2.2, is defined as the composition of *Br* and *Step*. It materializes an external nondeterministic choice, one that may be observed. The minimal computations respectively triggering an event e , delaying a branch, or generating observable branching, are defined as **trigger**, **branch** and **branchS**.

As expected, $\mathbf{ctree\ } E\ B$ forms a monad for any interfaces E and B : the **bind** combinator simply lazily crawls the potentially infinite first tree, and passes the value stored in any reachable leaf to the continuation. The **iter** combinator is central to encoding looping and recursive features: it takes as argument a body, **body**, intended to be iterated, and is defined such that the computation returns either a new index over which to continue iterating, or a final value; **iter** ties the recursive knot. Its definition is analogous to the one for ITrees, except that we need to ask ourselves how to guard the **cofix**: if **body** is a constant, pure, computation, unguarded corecursion would be ill-defined. ITrees use a **Step** node for this purpose, treated weakly by the equivalence built on the structure. Here, we instead use the **Guard** constructor, encoding a unary non-observable branch.

Convenience in building models comes at a cost: many CTrees represent the same LTS. Figure 6 illustrates this phenomenon by defining several CTrees implementing the stuck LTS and the silently spinning one. Indeed, the **Stuck** constructor is intended to canonically represent the stuck process. However, it can be mimicked via nullary branching: **stuckE** asks the environment a question without answer, while **stuckB** internally chose among none. Worse, the **spin*** trees are infinitely deep, but never find in their structure a transition to take. We define formally the necessary equivalences on computations to prove this informal statement in Section 4. Similarly, all the **spinS*** process are infinite traces of τ s.

⁵ The actual implementation uses a negative style with primitive projections. We omit this technical detail in the presentation.

```

553 (* Stuck processes *)
554 Definition stuck : ctree E B void := Stuck
555 Definition stuckE (e : E void) : ctree E B void :=
556   trigger e
557 Definition stuckB (b : B void): ctree E B void :=
558   branch b
559 CoFixpoint spin : ctree E B R := Guard spin
560 CoFixpoint spin_nary n (bn : B (fin n)) : ctree E B R :=
561   branch bn ;; spin_nary n bn
562
563 (* Spinning processes *)
564 CoFixpoint spinS : ctree E B R := Step spinS
565 CoFixpoint spinS_nary n (bn : B (fin n)) : ctree E B R :=
566   branchS bn ;; spinS_nary n

```






Fig. 6: Concrete representations of stuck and spinning LTSs, where `fin n` is a finite type with `n` elements.

```

567
568 Variant action E B R :=
569   | ARet (r : R)
570   | AStep (t : ctree E B R)
571   | AVis {X} (e : E X) (k : X → ctree E B R).
572
573 CoFixpoint head {E B R} (t : ctree E B R) : ctree E B (action E B R) :=
574   match t with
575   | Ret r ⇒ Ret (ARet r)
576   | Stuck ⇒ Stuck
577   | Step t ⇒ Ret (AStep t)
578   | Guard t ⇒ Guard (head t)
579   | Vis e k ⇒ Ret (AVis e k)
580   | Br b k ⇒ Br b (λ x ⇒ head (k x))
581   end.

```

Fig. 7: Lazily computing the set of reachable observable nodes (👁️)

3.2 A hint of introspection: heads of computations

Br nodes prevent the need for introspection over trees when modelling something as generic as a delayed branching construct such as the one specified by `BRDELAYED`. However, introspection becomes necessary to build a tree that depends on the reachable external actions of the sub-trees. This is the case for example for `CCS`'s parallel operator that we model in Section 7. The set of reachable external actions is not computable in general, as we may have to first know if the computation to the left of a sequence terminates before knowing if the events contained in the continuation are reachable. We are, however, in luck, as we have at hand a semantic domain able to represent potentially divergent computations: `CTrees` themselves!

The `head` combinator, described on Figure 7, builds a pure, potentially diverging computation *only made of delayed choices*, and whose leaves contain all reachable subtrees starting with an observable node. These “immediately” observable trees are captured in an `action` datatype, which is used as the return type of the built computation. The `head` combinator simply crawls the tree by reconstructing all delayed branches, until it reaches

a subtree with any other node at its root; it then returns that subtree as the corresponding **action**. The resulting **head** tree could be more precisely typed at the empty event interface if need be.

4 Equivalences and Equational Theory for CTrees

Section 3 introduced CTrees, the domain of computations we consider, as well as a selection of combinators upon it. We now turn to the question of comparing computations represented as CTrees for notions of equivalence and refinement. In particular, we formalize the LTS representation of a CTree that we have followed to justify our definitions. Through this section, we introduce a (coinductive) syntactic equality of CTrees, and lift a range of traditional process relations defined on LTSs to ctree: strong and weak bisimilarity, trace equivalence, (complete) similarities. We equip each of these notions with primitive up-to principals and equational theory for CTrees. Section 4.7 shows how our notions of program comparison generalizes to allow relating programs with different return types or signatures. Finally, Section 4.8 describes observations and preliminary results on weak bisimilarity. The theories discussed throughout this section provide the building blocks necessary for deriving domain-specific equational theories, such as the ones established in Section 7 for ccs, and in Section 8 for cooperative scheduling.

4.1 Coinductive proofs and up-to principles in Coq

Working with CTrees requires the use of a swarm of coinductive predicates and relations to describe equivalences, refinements and invariants. Doing so at scale in Coq would be highly impractical by relying only on its native support for coinductive proofs. Indeed, the language provides no abstract reasoning principle, native proofs by coinduction boil down to writing corecursive terms. But these terms must be provably productive to maintain the language’s soundness as a logic, which Coq enforces through a syntactic guard checker. However, in the case of corecursion, guard checking is incompatible with any automation, and not compositional. Thanks fully, coinduction is nowadays possible in Coq thanks to library support (Hur et al., 2013; Zakowski et al., 2020; Pous, 2016). In essence, they all rely on bypassing Coq’s native coinductive type in the definition of the relation of interest, and instead rely on an internalization of the theory of coinduction inside a library.

In particular, our development relies on Pous’s coinduction library (Pous, 2024) to define and reason about the various coinductive predicates and relations we manipulate. We briefly recall the essential facilities, based on the *companion* (Pous, 2016), and *tower induction* (Schäfer and Smolka, 2017), that the library provides. We finish this subsection by defining a subset of candidate “up-to” functions—we indicate in the subsequent subsections which ones constitute valid up-to principles for the various relations we introduce. Note: this section is aimed at the reader interested in understanding the internals of our library: it can be safely skipped at first read.

Knaster-Tarski. The core construction provided by Pous’s library is a greatest fixpoint operator ($\text{gfp } b : X$), for any complete lattice X , and monotone endofunction $b : X \rightarrow X$. In

particular, the sort of Coq propositions `Prop` forms a complete lattice, as do any function from an arbitrary type into a complete lattice—coinductive relations, of arbitrary arity, over arbitrary types, can therefore be built using this combinator. In the context of this paper, we mostly instantiate `X` with the complete lattice of binary relations over `CTrees`, written `C`: `C := ctree E B A → ctree E B A → Prop` for fixed parameters `E`, `B` and `A`. We write such binary relations as `rel(A,B)` for `A → B → Prop`, and `rel(A)` for `rel(A,A)`; for instance: `C := rel(ctree E B A)`.

At the most elementary level, the library provides tactic support for coinductive proofs based on Knaster-Tarski’s theorem: any post fixpoint is below the greatest fixpoint. Spelled out formally in the general case over a complete lattice (X, \sqsubseteq) (left), and specialized to C (right):

$$\frac{x \sqsubseteq y \sqsubseteq by}{x \sqsubseteq \mathbf{gfp} b} \qquad \frac{R \subseteq S \quad \forall t u, S t u \rightarrow b S t u}{\forall t u, R t u \rightarrow \mathbf{gfp} b t u}$$

Over C , the proof method therefore consists in exhibiting a relation R , that can be thought of as a set of pairs of trees, providing a “coinduction candidate”, and proving that it is stable by a *play of the bisimulation game*, i.e., stable under the endofunction b .

Enhanced coinduction. The larger the coinduction candidate R , the more work needed: one must play the bisimulation game over any pair of trees in R . As often, this inherent difficulty gets even more salient in a proof assistant. Enhanced coinduction principles, or equivalently *up-to principles*, seek to provide more general reasoning principles. They intuitively consist in allowing one to fall slightly out of the coinduction principle after playing the game, and still conclude: rather than looking for a post fixpoint of b , we look for one of $b\mathbf{f}$, where \mathbf{f} should be thought of as enlarging the candidate. Concretely, we say that a function $\mathbf{f} : X \rightarrow X$ defines a valid enhanced coinduction principle if the following reasoning principle is valid:

$$\frac{x \sqsubseteq y \sqsubseteq b\mathbf{f}y}{x \sqsubseteq \mathbf{gfp} b} \qquad \frac{R \subseteq S \quad \forall t u, S t u \rightarrow b (\mathbf{f} S) t u}{\forall t u, R t u \rightarrow \mathbf{gfp} b t u}$$

Let us illustrate the concept on a few concrete examples. Suppose you start from a non-reflexive candidate R , and observe during your proof that pairs of processes in R either progress to pairs in R , or to definitionally equal pairs of processes. Unable to conclude in the latter case, one would typically backtrack and expand R by taking its reflexive closure, before going through the proof again, adding in the process proofs for the new pairs. Instead, one may prove that the function $f_{refl} R \triangleq R \cup \{(t, t)\}$ is a valid principle, and close the original proof with R as a candidate.

As a second standard example, consider a proof of bisimilarity over your trees, as introduced formally in Section 4.4. Your pairs may progress only so slightly out of the candidate R : on each side, the resulting trees are themselves bisimilar to elements in R . Saturating your candidate to close it under strong bisimilarity might complicate greatly your proof, while establishing the validity of bisimulation up-to bisimilarity, that is the principle associated to $f_{bisim} R \triangleq \{(t, u) \mid \exists t' u', t \sim t' \wedge u' \sim u \wedge R t' u'\}$, is sufficient to conclude.

691 Lastly, given a source language, up-to congruence are a commonly crucial reasoning
 692 principle. For instance, considering in ImpBr the $\text{br} \cdot \text{or} \cdot$ construct, one may wonder
 693 whether $f_{br} R \triangleq \{(\text{br } p \text{ or } q, \text{br } p' \text{ or } q') \mid R p p' \wedge R q q'\}$ is valid.

694 But what if we need to use both up-to reflexivity and up-to $\text{br} \cdot \text{or} \cdot$ context in
 695 the same proof? Is the combined principle still valid? To answer such questions, and
 696 more generally ease the construction of valid up-to principles, significant effort has been
 697 invested in identifying classes of sound up-to principles, and developing ways to combine
 698 them (Sangiorgi, 1998; Pous, 2007; Sangiorgi and Rutten, 2012).

699 In particular, Pous’s library is built upon the so-called *companion*, a construction which
 700 relies on one particular class of functions, the so-called *compatible* functions. Their charac-
 701 terization is fairly elementary: f is compatible with b if $fb \sqsubseteq bf$. The class is of particular
 702 interest for two main properties. First, all compatible functions are sound up-to functions.
 703 Furthermore, the set of compatible functions for a given b forms a complete lattice. One
 704 may therefore consider the greatest compatible function, dubbed the *companion* and written
 705 t_b , and observe it is itself compatible. We refer the interested reader to Pous (2016) for
 706 more details, and to Pous (2024) for pedagogical example of the library in action.

707 From a user perspective, this is priceless: rather than craft and pick the right up-to
 708 principle for each proof, they can systematically work up to the companion, progressively
 709 enhance their database of proven compatible principles,⁶ and access them on the fly during
 710 the proofs. In the three examples above, access to these up-to principles would therefore be
 711 granted by proving respectively $f_{refl} \sqsubseteq t_b$, $f_{bisim} \sqsubseteq t_b$, and $f_{br} \sqsubseteq t_b$, which in turn can be
 712 in particular proved by showing that f_{refl} , f_{bisim} , and f_{br} are compatible.

713 **Tower induction.** The final concept we greatly benefit from through our development,
 714 via Pous’s library, is Schäfer and Smolka’s characterization of the companion via tower
 715 induction (Schäfer and Smolka, 2017). They associate to the endofunction b the so-called
 716 b -tower, or C_b the *Chain* of b , i.e., the inductive type closed under b and infimum.
 717 The greatest fixpoint of b is recovered as the infimum of its chain, $\text{gfp } b = \inf C_b$, and
 718 more generally the companion as $t_b(x) = \inf\{y \in C_b \mid x \sqsubseteq y\}$. Once again, we refer the
 719 interest reader to (Schäfer and Smolka, 2017) for technical details, and only highlight here
 720 the consequences for us, users of the library: since version 1.7, Pous’s library has been
 721 reimplemented following Schäfer and Smolka’s construction.

722 The benefit shows when proving the validity of additional proof principles: in particular,
 723 exploiting previously established sound principles in the proof of a new one required a
 724 clever but non-trivial notion of second-order companion. In contrast, when working with
 725 the tower, the statement of validity a function is much more natural: it essentially amounts
 726 to proving that f respects membership to the chain.

727 Let us make things concrete over our the three illustrative examples. Validity of f_{refl} is
 728 now expressed by stating that all elements of the chain are reflexive, i.e., $\forall(c : C_b) t, c t t$.
 729 Validity of f_{bisim} captures that elements of the chain are stable by bisimilarity on both
 730 side, i.e., $\forall(c : C_b) t t' u u', t \sim t' \rightarrow u' \sim u \rightarrow c t' u' \rightarrow c t u$, or more concisely expressed
 731 in Coq as `Proper (sbisim => sbisim => iff) c`. Finally, validity of f_{br} is written as $\forall(c :$

732
733
734
735 ⁶ Or slightly more generally, of any function proven below the companion.

$$\text{REFL}^{up} R \triangleq \{(x, x)\} \qquad \text{SYM}^{up} R \triangleq \{(y, x) \mid R x y\}$$

$$\text{TRANS}^{up} R \triangleq \{(x, z) \mid \exists y, R x y \wedge R y z\}$$

$$\text{BIND}^{up}(\text{equiv}) R \triangleq \{(x \gg k, y \gg l) \mid \text{equiv } x y \wedge \forall v, R (k v) (l v)\}$$

$$\text{UPTO}^{up}(\text{equiv}) R \triangleq \{(x, y) \mid \exists x' y', \text{equiv } x x' \wedge R x' y' \wedge \text{equiv } y' y\}$$

Fig. 8: Main generic up-to principles used for relations of CTrees where $R : \text{rel } (\text{ctree } E B X)$

C_b) $p p' q q', c p p' \rightarrow c q q' \rightarrow c (\text{br } p \text{ or } q) (\text{br } p' \text{ or } q')$, or again more concisely expressed in Coq as `Proper (c => c => c) br`.

Furthermore, the construction comes with a powerful proof principle for proving these properties: *tower induction*, i.e., it suffices to prove that these universal properties over elements of the chain are stable under b and infimum to establish them.

During proofs by coinduction, the difference is mostly cosmetic, although it also lightens the proofs: since every principle is directly expressed in terms of the chain, there is no need to awkwardly pull out valid principles from the chain, we can directly apply them.

Conventions and preliminary definitions. In the remainder of this section, we describe the family of relations over CTrees our library support. All these coinductive definitions are defined via the library as greatest fixed point, i.e., behind the scene, as the infimum of the corresponding chain. We hope the context provided through this subsection will help the interested reader investigating our development, but we will naturally mostly stick to a standard presentation.

In particular, we present proof systems over the relations, i.e., the greatest fixpoints, but also up-to principles that their corresponding endofunction satisfies. Once again, behind the scene, they are expressed in terms of properties of the elements of the corresponding chain. Through this presentation however we elect to keep it under a simpler format. Consider the example of strong bisimilarity, written \sim . Up-to principles (see Fig. 13) are presented as proof rules whose conclusion is expressed in terms of $\sim_{\mathcal{R}}$, i.e., the goal of an ongoing proof by bisimulation with candidate \mathcal{R} . The coinduction hypothesis might not be yet accessible in this case, we may have to play the bisimulation game beforehand. The premise may involve three kinds of relations: other relations, such as bisimilarity as is the case of f_{bisim} , the candidate \mathcal{R} , illustrating that applying the rule *unlocks* the coinduction hypothesis, allowing to soundly conclude based on it, or $\sim_{\mathcal{R}}$ itself, where it is sound to apply the rule, but does not give access to the coinduction hypothesis.

Figure 8 describes some main generic up-to principles we use for our relations on CTrees.⁷ Recall that over C , these potential up-to functions are endofunctions of relations: for instance, REFL^{up} is the constant diagonal relation, SYM^{up} builds the symmetric relation, TRANS^{up} is the composition of relations. The validity of REFL^{up} , SYM^{up} and TRANS^{up} for a given endofunction b entails respectively the reflexivity, symmetry and transitivity of the relations $(b\tau_b R)$ and $(\tau_b R)$. These two relations are precisely the ones involved

⁷ These are the core examples of library level up-to principles we provide. For our CCS case study, we also prove the traditional language level ones.

during a proof by coinduction up-to companion: the former as our goal, the latter as our coinduction hypothesis. The $\text{BIND}^{\text{up}}(_)$ up-to function helps when reasoning structurally, allowing to cross through `bind` constructs during proofs by coinduction. Finally, validity of the $\text{UPRO}^{\text{up}}(\text{equiv})$ principle allows for rewriting via the *equiv* relation during coinductive proofs for *b*.

Remark that REFL^{up} is slightly simpler than f_{refl} defined above: rather than extend *R*, it is the constant equality relation. This stems from working with the companion/tower induction: we can pull any valid principle from the companion at any time, here when we are ready to conclude by reflexivity, it is not forced on us ahead of time.

4.2 Coinductive equality for CTree

Coq's equality, `eq`, is not a good fit to express the structural equality of coinductive structures—even the eta-law for a coinductive data structure does not hold up-to `eq`. We therefore define, as is standard, a structural equality⁸ by coinduction `equ`: `rel(CTree E B A)` (written \cong in infix). The endofunction simply matches head constructors and behaves extensionally on continuations.

Definition 1. *Structural equality* (🔴)

$$\begin{aligned} \text{equ} \triangleq & \text{gfp } \lambda R . \{(\text{Ret } v, \text{Ret } v)\} \cup \\ & \{(Vis\ e\ k, Vis\ e\ k') \mid \forall v, R\ (k\ v)\ (k'\ v)\} \cup \\ & \{(Br^c\ k, Br^c\ k') \mid \forall v, R\ (k\ v)\ (k'\ v)\} \cup \\ & \{(\emptyset, \emptyset)\} \cup \\ & \{(\text{Guard } t, \text{Guard } u) \mid R\ t\ u\} \cup \\ & \{(\text{Step } t, \text{Step } u) \mid R\ t\ u\} \end{aligned}$$

The `equ` relation raises no surprises: it is an equivalence relation, and is adequate to prove all eta-laws—for the CTree structure itself and for the cofixes we manipulate. Similarly, the usual monadic laws are established with respect to `equ`.

Lemma 1. *Monadic laws* (🔴)

$$\text{Ret } v \ggg k \cong k\ v \quad x \leftarrow t ;; \text{Ret } x \cong t \quad (t \ggg k) \ggg l \cong t \ggg (\lambda x \Rightarrow k\ x \ggg l)$$

Of course, formal equational reasoning with respect to an equivalence relation other than `eq` comes at the usual cost: all constructions introduced over CTrees must be proved to respect `equ` (in Coq parlance, they must be `Proper`), allowing us to work painlessly with setoid-based rewriting.⁹

Finally, we establish some enhanced coinduction principles for `equ`.

⁸ Note that for ITrees, this relation corresponds to what Xia et al. dub as *strong bisimulation*, and name `eq_itree`. We carefully avoid this nomenclature here to reserve this term for the relation we define in Section 4.4

⁹ As is the case for the `eq_itree` over ITrees, postulating as an axiom that `equ` coincide with definitional equality would be sound in Coq. We are however doomed to embrace setoids anyway for all other relations, we therefore avoid doing so.

label \triangleq tau | obs e v | val v

$$\frac{k \ v \xrightarrow{l} t}{Br^b \ k \xrightarrow{l} t} \quad \frac{t \xrightarrow{l} u}{Guard \ t \xrightarrow{l} u} \quad \frac{}{Step \ t \xrightarrow{\text{tau}} t} \quad \frac{}{Vis \ e \ k \xrightarrow{\text{obs } e \ v} k \ v} \quad \frac{}{Ret \ v \xrightarrow{\text{val } v} \emptyset}$$

Fig. 9: Inductive characterization of the LTS induced by a CTree (👁)

Lemma 2. *Enhanced coinduction for equ* (👁)

REFL^{UP}, SYM^{UP}, TRANS^{UP}, BIND^{UP}(\cong) (👁) and UPTO^{UP}(\cong) (👁) provide valid up-to principles for equ.

Meaning, to spell it out explicitly a last time, that one has the ability, *in the middle of a proof by coinduction aiming to establish that two trees are equ*, to invoke reflexivity, symmetry, transitivity, congruence for bind, and to rewrite previously established equations.

While equ, as a structural equivalence, is very comfortable to work with, it, naturally, is much too stringent. To reason semantically about CTrees, we need a relation that remains termination sensitive, but allows for mismatch in the amount of internal steps, that still imposes a tight correspondence over external events, but relaxes its requirement for non-deterministically branching nodes. We achieve this by drawing from standard approaches developed for process calculi.

4.3 Looking at CTrees under the lens of labeled transition systems

To build a notion of bisimilarity between CTree computations, we associate a labeled transition system to a CTree, as defined in Figure 9. This LTS exhibits three kinds of labels: a tau¹⁰ witnesses a stepping branch, an obs e x observes the encountered event together with the answer from the environment considered, and a val v is emitted when returning a value. Interestingly, there is a significant mismatch between the structure of the tree and the induced LTS: the states of the LTS correspond to the nodes of the CTree *that are not immediately preceded by a delayed choice*. Accordingly, the definition of the transition relation between states inductively iterates over delayed branches. Stepping branches and visible nodes map immediately to a set of transitions, one for each outgoing edge; finally a return node generates a single val transition, moving onto a stuck state, encoded as the \emptyset constructor. These rules formalize the intuition we gave in Section 2.2 that allowed us to derive the LTSs of Figure 4 from the corresponding ImpBr terms.

Defining the property of a tree to be *stuck*, that is: $t \rightarrow \triangleq \forall l u, \neg(t \xrightarrow{l} u)$, we can make the depictions from Figure 6 precise: \emptyset itself and nullary nodes are stuck by construction, since stepping would require a branch, while spin_nary is proven to be stuck by induction, since it cannot reach any step.

$$\emptyset \rightarrow \quad Br^0 \rightarrow \quad \text{spin_nary } n \ b_n \rightarrow$$

¹⁰ We warn again the reader accustomed to ITrees to think of tau under the lens of the process algebra literature, and not as a representation of ITree's Tau constructor.

$$\begin{array}{c}
875 \quad \frac{t \xrightarrow{l} u \quad l \neq \text{val } v}{t \gg k \xrightarrow{l} u \gg k} \qquad \frac{t \xrightarrow{\text{val } v} \emptyset \quad k \xrightarrow{l} u}{t \gg k \xrightarrow{l} u} \\
876 \\
877 \\
878 \\
879 \quad \frac{t \gg k \xrightarrow{l} u}{(l \neq \text{val } v \wedge \exists t', t \xrightarrow{l} t' \wedge u \cong t' \gg k) \vee (\exists v, t \xrightarrow{\text{val } v} \emptyset \wedge k \xrightarrow{l} u)} \\
880 \\
881
\end{array}$$

Fig. 10: Transitions under bind (🔥)

882 The stepping relation interacts slightly awkwardly with `bind`: indeed, although a unit for
883 `bind`, the `Ret` construct is not inert from the perspective of the LTS. Non `val` transitions can
884 therefore be propagated below the left-hand-side of a `bind`, while a `val` transition in the
885 prefix does not entail the existence of a transition in the `bind`. If one prefer to imagine LTSs,
886 the LTSs we consider have a set of "final" states, the `Ret` ones, that take a last observable
887 transition to a well; the `bind` operator fuses a LTSs to a family of LTSs in sequence by
888 rewiring the arrows to the well to the entry point of the corresponding LTS in the family.
889 Figure 10 describes the lemmas capturing this intuition, by distinguishing the special case
890 of a `val v` transition.

891 We additionally define the traditional *weak transition* $s \xRightarrow{l} t$ on the LTS that can perform
892 tau transitions before or after the l transition (and possibly be none if $l = \tau$). This part of
893 the theory is so standard that we can directly reuse parts of the development for CCS that
894 Pous developed to illustrate the companion (Pous, 2024), with the exception that we need
895 to work in a Kleene Algebra with a model closed under equ rather than eq.

900 4.4 Bisimilarity

901 Having settled on the data structure and its induced LTS, we are back on a well-traveled
902 road: strong bisimilarity (referred simply as bisimilarity in the following) is defined in a
903 completely standard way over the LTS view of CTrees.

904 **Definition 2** (Bisimulation for CTrees (🔥)). *The progress function sb for bisimilarity maps
905 a relation \mathcal{R} over CTrees to the relation such that $sb \mathcal{R} s t$, also noted $t \sim_{\mathcal{R}} u$, holds if and
906 only if:*

$$907 \quad \forall l t', t \xrightarrow{l} t' \implies \exists u'. t' \mathcal{R} u' \wedge u \xrightarrow{l} u'$$

908 and conversely

$$909 \quad \forall l u', u \xrightarrow{l} u' \implies \exists t'. t' \mathcal{R} u' \wedge t \xrightarrow{l} t'$$

910 The bisimulation game is represented visually in Figure 11. Bisimilarity, written $t \sim u$, is
911 defined as the greatest fixpoint of sb : $\text{sbisim} \triangleq \text{gfp } sb$.

912 All the traditional tools surrounding bisimilarity can be transferred to our setup. We omit
913 the details, but additionally provide a characterization of the traces (represented as colists)
914 of a CTree, used to define trace-equivalence (written \equiv_{tr}) (🔥). We provide elementary



Fig. 11: The bisimulation game $\sim_{\mathcal{R}}$

$$\begin{array}{c}
 \frac{x = y}{\text{Ret } x \sim \text{Ret } y} \quad \frac{\forall x, h x \sim k x}{\text{Vis e } h \sim \text{Vis e } k} \quad \frac{(\forall x, \exists y, h x \sim k y) \wedge (\forall y, \exists x, h x \sim k y)}{\text{Br}^c h \sim \text{Br}^d k} \\
 \\
 \frac{t \sim u}{\text{Step } t \sim \text{Step } u} \quad \frac{(\forall x, \exists y, h x \sim k y) \wedge (\forall y, \exists x, h x \sim k y)}{\text{Br}_S^c h \sim \text{Br}_S^d k} \\
 \\
 \frac{t \sim u \quad (\forall x, g x \sim k x)}{t \gg g \sim u \gg k} \\
 \\
 \text{Guard } t \sim t \quad \frac{u \rightarrow}{\text{Br}^2 t u \sim t} \quad \text{Br}^2 t (\text{Br}^2 u v) \sim \text{Br}^2 (\text{Br}^2 t u) v \quad \text{Br}^2 t u \sim \text{Br}^2 u t \\
 \\
 \text{Br}^2 t t \sim t \quad \text{Br}^2 (\text{Br}^2 t u) v \sim \text{Br}^3 t u v \quad \text{Br}_S^2 t u \sim \text{Br}_S^2 u t \quad \text{Br}_S^2 t t \sim \text{Step } t \\
 \\
 \text{Step } t \approx t \quad \text{spin_nary } n \sim \text{spin_nary } m \quad \frac{(n > 0 \wedge m > 0) \vee (n = m = 0)}{\text{spin}_S \text{_nary } n \sim \text{spin}_S \text{_nary } m}
 \end{array}$$

Fig. 12: Elementary equational theory for CTrees (A)

results for it, notably that sbisim entails trace-equivalence, but that the converse does not hold.

4.4.1 Core equational theory

Bisimilarity forms an equivalence relation satisfying a collection of primitive laws for CTrees summed up in Figure 12. We use simple inference rules to represent an implication from the premises to the conclusion, and double-lined rules to represent equivalences. Each rule is proved as a lemma with respect to the definitions above.

The first four rules recover some structural reasoning on the syntax of the trees from its semantic interpretation. These rules are much closer to what eut t provides by construction for ITrees: leaves are bisimilar if they are equal, computations performing the same external interaction must remain point-wise bisimilar, and unary steps can be matched against one another. Delayed branches, potentially of distinct arity, can be matched one against another if both domains of indexes can be injected into the other to reestablish bisimilarity. This is only an implication and not an equivalence since the points of the continuation structurally immediately accessible do not correspond to accessible states in the LTS. By contrast, this

same condition is necessary and sufficient for *stepping* branches, as their *Step* nodes induce additional τ transitions, thus new states in the LTS. Finally, bisimilarity is a congruence for *bind*.

Another illustration of the absence of equivalence for delayed branching nodes as head constructor is that such a computation may be strongly bisimilar to a computation with a different head constructor. The simplest example is that *sbisim* can ignore (finite numbers of) *Guard* nodes. Another example is that stuck processes behave as a unit for delayed branching nodes. We furthermore obtain the equational theory that we expect for non-deterministic effects. Delayed branching is associative, commutative, idempotent, and can be merged into delayed branching nodes of larger arity w.r.t. *sbisim*.¹¹ By contrast, stepping branches are only commutative, and almost idempotent, provided we introduce an additional *Step*. This *Step* can in turn be ignored by moving to weak bisimilarity, making stepping branches commutative and properly idempotent; however, it crucially remains *not* associative, a standard fact in process algebra.¹²

Finally, two delayed spins are always bisimilar (neither process can step) while two stepping spins are bisimilar if and only if they are both nullary (neither one can step), or both non-nullary.

We omit the formal equations for sake of space here, but we additionally prove that the *iter* combinator deserves its name: the Kleisli category of the *ctree E* monad is iterative w.r.t. strong bisimulation (♣). Concretely, we prove that the four equations described by Xia et al. (2020), Section 4, hold true. The fact that they hold w.r.t. strong bisimulation is a direct consequence of the design choice taken in our definition of *iter*: recursion is guarded by a *Guard*. We conjecture that one could provide an alternate iterator guarding recursion by a *Step*, and recover the iterative laws w.r.t. weak bisimulation, but have not proved it and leave it as future work. We expand further on a handful of delicate points related to weak bisimilarity in Section 4.8.

Naturally, this equational theory gets trivially lifted at the language level for *ImpBr* (♣). The acute reader may notice that in exchange for being able to work with strong bisimulation, we have mapped the silently looping program to *spinD*, hence identifying it with stuck processes. For an alternate model observing recursion, one would need to investigate the use of the alternate iterator mentioned above and work with weak bisimilarity.

4.4.2 Proof system for bisimulation proofs

As is usual, the laws in Figure 12, enriched with domain-specific equations, allow for deriving further equations purely equationally. But to ease the proof of these primitive laws, as well as new nontrivial equations requiring explicit bisimulation proofs, we provide proof rules that are valid during bisimulation proofs, derived from valid up-to principles.

We recall that given a bisimulation candidate \mathcal{R} , we write $t \sim_{\mathcal{R}} u$ for *sb R*, i.e. the proof goal in which the coinduction hypothesis is not yet accessible.¹³ We depict the main rules we use in Figure 13. In particular, from the standpoint of the proof scientist, these rules

¹¹ Stating these facts generically in the arity of branching is quite awkward, we hence state them here for binary branching, but adapting them at other arities is completely straightforward.

¹² This choice would be referred as external in this community

¹³ As explained in Section 4.1, in the implementation, R would hence be specifically an element of the chain of *sb*, and the up-to principles stated in terms of closure properties valid on all elements of this chain.

Ret $v \sim_{\mathcal{R}} \text{Ret } v$

$$\frac{\forall v, (k \ v) \ \mathcal{R} \ (k' \ v)}{\text{Vis } e \ k \sim_{\mathcal{R}} \text{Vis } e \ k'}$$

$$\frac{(\forall x, \exists y, (k \ x) \sim_{\mathcal{R}} (k' \ y)) \wedge (\forall y, \exists x, (k \ x) \sim_{\mathcal{R}} (k' \ y))}{\text{Br}^c \ k \sim_{\mathcal{R}} \text{Br}^d \ k'}$$

$$\frac{t \sim_{\mathcal{R}} u}{\text{Guard } t \sim_{\mathcal{R}} \text{Guard } u}$$

$$\frac{t \ \mathcal{R} \ u}{\text{Step } t \sim_{\mathcal{R}} \text{Step } u}$$

$$\frac{(\forall x, \exists y, (k \ x) \ \mathcal{R} \ (k' \ y)) \wedge (\forall y, \exists x, (k \ x) \ \mathcal{R} \ (k' \ y))}{\text{Br}_S^c \ k \sim_{\mathcal{R}} \text{Br}_S^d \ k'}$$

$$\frac{\forall v, (k \ v) \ \mathcal{R} \ (k' \ v)}{\text{Br}_S^c \ k \sim_{\mathcal{R}} \text{Br}_S^c \ k'}$$

Fig. 13: Proof rules for coinductive proofs of sbisim (🔥)

notably avoid the exponential explosion in the number of subgoals in our proofs that would arise by simply playing the bisimulation game iteratively: it entails a binary split at each level of play. These rules essentially match up counterpart CTree constructors at the level of bisimilarity, but additionally make a distinction whether applying the rule soundly acts as playing the game—i.e., the premises refer to \mathcal{R} , allowing to conclude using the coinduction hypothesis—or whether they do not—i.e., the premises still refer to $\sim_{\mathcal{R}}$. The latter situation arises when using the proof rules that strip off delayed branches from the structure of our trees: on either side, they do not entail any step in the corresponding LTSs, but rather correspond to recursive calls to its inductive constructor.

Furthermore, we provide a rich set of valid up-to principles:

Lemma 3. *Enhanced coinduction for sbisim (🔥) The functions REFL^{up} , SYM^{up} , TRANS^{up} , $\text{BIND}^{up}(\sim)$ (🔥), $\text{UPTO}^{up}(\cong)$ (🔥) and $\text{UPTO}^{up}(\sim)$ (🔥) provide valid up-to principles for sbisim .*

In particular, the equation $\text{Guard } t \sim t$ can be used for rewriting during bisimulation proofs, allowing for asymmetric stripping of guards.

4.5 Similarity

Similarity is a well-studied notion of refinement that is typically coinductively defined over an LTS. We provide two of the many existing variants of similarity: *strong similarity* and *complete strong similarity*. The latter is slightly more complex but behaves better in presence of stuck LTSs.

4.5.1 Strong similarity

Strong similarity is defined as the greatest fixpoint of the bisimulation left half-game previously depicted in Figure 11.

$$\begin{array}{c}
1059 \quad \frac{t \rightarrow}{t \lesssim_{\mathcal{R}} u} (ss_stuck) \quad \text{Ret } v \lesssim_{\mathcal{R}} \text{Ret } v (ss_ret) \quad \frac{\forall v, (k \ v) \ \mathcal{R} \ (k' \ v)}{\text{Vis } e \ k \lesssim_{\mathcal{R}} \text{Vis } e \ k'} (ss_vis_id) \\
1060 \\
1061 \\
1062 \quad \frac{\forall x, (k \ x) \lesssim_{\mathcal{R}} u}{Br^c \ k \lesssim_{\mathcal{R}} u} (ss_br_l) \quad \frac{\exists y, t \lesssim_{\mathcal{R}} (k' \ y)}{t \lesssim_{\mathcal{R}} Br^d \ k'} (ss_br_r) \\
1063 \\
1064 \\
1065 \quad \frac{\forall x, \exists y, (k \ x) \lesssim_{\mathcal{R}} (k' \ y)}{Br^c \ k \lesssim_{\mathcal{R}} Br^d \ k'} (ss_br) \quad \frac{\forall v, (k \ v) \lesssim_{\mathcal{R}} (k' \ v)}{Br^c \ k \lesssim_{\mathcal{R}} Br^c \ k'} (ss_br_id) \\
1066 \\
1067 \\
1068 \\
1069 \quad \frac{t \lesssim_{\mathcal{R}} u}{\text{Guard } t \lesssim_{\mathcal{R}} u} (ss_guard_l) \quad \frac{t \lesssim_{\mathcal{R}} u}{t \lesssim_{\mathcal{R}} \text{Guard } u} (ss_guard_r) \\
1070 \\
1071 \\
1072 \quad \frac{t \lesssim_{\mathcal{R}} u}{\text{Guard } t \lesssim_{\mathcal{R}} \text{Guard } u} (ss_guard) \quad \frac{t \ \mathcal{R} \ u}{\text{Step } t \lesssim_{\mathcal{R}} \text{Step } u} (ss_step) \\
1073 \\
1074 \\
1075 \quad \frac{\forall x, \exists y, (k \ x) \ \mathcal{R} \ (k' \ y)}{Br_S^c \ k \lesssim_{\mathcal{R}} Br_S^d \ k'} (ss_brS) \quad \frac{\forall v, (k \ v) \ \mathcal{R} \ (k' \ v)}{Br_S^c \ k \lesssim_{\mathcal{R}} Br_S^c \ k'} (ss_brS_id) \\
1076 \\
1077 \\
1078 \\
1079 \\
1080 \\
1081 \\
1082 \\
1083 \\
1084 \\
1085 \\
1086 \\
1087 \\
1088 \\
1089 \\
1090 \\
1091 \\
1092 \\
1093 \\
1094 \\
1095 \\
1096 \\
1097 \\
1098 \\
1099 \\
1100 \\
1101 \\
1102 \\
1103 \\
1104
\end{array}$$

Fig. 14: Rules for coinductive proofs of `ssim`, with their names in our Coq library (🔥)

Definition 3 (Simulation for CTrees (🔥)). *The progress function `ss` for similarity maps a relation \mathcal{R} over CTrees to the relation such that $ss \ R \ t \ u$ (also noted $t \lesssim_{\mathcal{R}} u$) holds if and only if:*

$$\forall t', t \xrightarrow{l} t' \implies \exists u'. t' \ \mathcal{R} \ u' \wedge u \xrightarrow{l} u'$$

Similarity, written $t \lesssim u$, is defined as the greatest fixpoint of `ss`: `ssim` \triangleq `gfp ss`.

The coinductive proof rules for simulation are depicted on Figure 14. The rules are similar to those for bisimulation (Figure 13), but more permissive. In particular, a stuck process is simulated by any CTree (`ss_stuck`). Furthermore, additional asymmetric reasoning principles are available over `Br` nodes. `ss_br_l` states that a `Br` node is simulated by a CTree u if and only if all its branches are simulated by u . Conversely, `ss_br_r` states that a CTree starting with a `Br` node simulates a CTree t if one of its branches simulates t . These powerful rules illustrate the semantic particularity of `Br` nodes: they are collapsed and thus completely invisible in the LTS. The composition of `ss_br_l` and `ss_br_r` gives `ss_br`, which is similar to the `sbisim` proof rule for `Br`, with the symmetric condition dropped.

For convenience, the proof rules are lifted to `ssim`-level, raising equations similar to the ones in Figure 12. We omit the details as the resulting rules can be trivially deduced from the $\lesssim_{\mathcal{R}}$ ones by replacing occurrences of both \mathcal{R} and $\lesssim_{\mathcal{R}}$ by \lesssim in Figure 14.

All the up-to principles valid for `sbisim` are also valid for `ssim` except of course for the symmetry principle.



Fig. 15: The complete simulation game $\lesssim_{\mathcal{R}}^C$

Lemma 4. *Enhanced coinduction for ssim (🔴)* The functions REFL^{uP} , TRANS^{uP} , $\text{BIND}^{uP}(\lesssim)$ (🔴), $\text{UPTO}^{uP}(\cong)$ (🔴) and $\text{UPTO}^{uP}(\sim)$ (🔴) provide valid up-to principles for ssim .

4.5.2 Complete similarity

A well-known limitation of similarity is its deadlock-insensitivity: it directly follows from the definition of the simulation game that a stuck LTS is simulated by any LTS, and we have indeed a corresponding proof rule in Figure 14.

In many applications, when stating that a program or process p refines a non-stuck process q , we mean to prove that p exhibit a *non-empty* subset of the behaviors of q , which strong similarity therefore fails to capture. Several notions of similarity tackle this limitation (see for instance Chapter 6 of (Sangiorgi, 2012)). This section is dedicated to complete similarity, an intuitive answer to this problem. Concretely, a complete simulation game is defined like a simulation game, with the additional constraint that if either LTS is stuck, the other one should be too. Figure 15 depicts graphically the following complete simulation game, where In these rules, we write $t \rightarrow$ for $\exists l t', t \xrightarrow{l} t'$, i.e. t is not stuck.

Definition 4 (Complete simulation for CTrees (🔴)). *The progress function css for complete similarity maps a relation \mathcal{R} over CTrees to the relation such that $\text{css } \mathcal{R} t u$ (also noted $t \lesssim_{\mathcal{R}}^C u$) holds if and only if*

$$(\text{ss } \mathcal{R} t u) \wedge (\text{if } u \rightarrow \text{ then } t \rightarrow)$$

Complete similarity, written $t \lesssim^C u$, is defined as the greatest fixpoint of css : $\text{cssim} \triangleq \text{gfp } \text{css}$.

The coinductive proof rules for complete simulation are depicted on Figure 16. The proof rules are basically the same as with ssim , with conditions to ensure that a CTree does not become stuck after taking a step. Note however that in cases where a node is matched against the exact same node, no additional condition is enforced. These cases are greyed out in the figure, as they are identical to the rules for ssim .

The up-to principles that are valid for ssim are also valid for cssim , except the up-to bind principle. We define a more restrictive up-to principle for complete similarity that requires the continuation not to be stuck.

$$\text{BIND}_c^{uP}(\text{equiv}) R \triangleq \{(x \gg k, y \gg l) \mid \text{equiv } x y \wedge \forall v, R(k v)(l v) \wedge k v \rightarrow\}$$

$$\begin{array}{c}
1151 \quad \text{Ret } v \lesssim_{\mathcal{R}}^C \text{Ret } v \\
1152 \\
1153 \\
1154 \quad \frac{\forall v, (k v) \mathcal{R} (k' v)}{\text{Vis } e k \lesssim_{\mathcal{R}}^C \text{Vis } e k'} \quad \frac{\forall x \in X, (k x) \lesssim_{\mathcal{R}}^C u \text{ inhabited } X}{\text{Br}^c k \lesssim_{\mathcal{R}}^C u} \quad \frac{\exists y, t \lesssim_{\mathcal{R}}^C (k' y) \quad t \rightarrow}{t \lesssim_{\mathcal{R}}^C \text{Br}^d k'} \\
1155 \\
1156 \\
1157 \quad \frac{\forall x, \exists y, (k x) \lesssim_{\mathcal{R}}^C (k' y) \quad \exists x, k x \rightarrow}{\text{Br}^c k \lesssim_{\mathcal{R}}^C \text{Br}^d k'} \quad \frac{\forall v, (k v) \lesssim_{\mathcal{R}}^C (k' v)}{\text{Br}^c k \lesssim_{\mathcal{R}}^C \text{Br}^c k'} \quad \frac{t \lesssim_{\mathcal{R}}^C u}{\text{Guard } t \lesssim_{\mathcal{R}}^C \text{Guard } u} \\
1158 \\
1159 \\
1160 \quad \frac{t \mathcal{R} u}{\text{Step } t \lesssim_{\mathcal{R}}^C \text{Step } u} \quad \frac{\forall x \in X, \exists y, (k x) \mathcal{R} (k' y) \text{ inhabited } X}{\text{Br}_S^c k \lesssim_{\mathcal{R}}^C \text{Br}_S^d k'} \quad \frac{\forall v, (k v) \mathcal{R} (k' v)}{\text{Br}_S^c k \lesssim_{\mathcal{R}}^C \text{Br}_S^c k'} \\
1161 \\
1162
\end{array}$$

Fig. 16: Proof rules for coinductive proofs of cssim (🔥)

Lemma 5. *Enhanced coinduction for cssim (🔥) The functions REFL^{up} , TRANS^{up} , $\text{BIND}_c^{up} (\lesssim^C)$ (🔥), $\text{UPTO}^{up} (\cong)$ (🔥) and $\text{UPTO}^{up} (\sim)$ (🔥) provide valid up-to principles for cssim .*

4.6 Proof example

Let us pause to illustrate concretely over a minimalist toy example how one can conduct in our library a proof of similarity by enhanced coinduction. We consider here CTrees with a *print* event for printing a boolean value, and binary internal branches.

```
1175 Variant PrintE : Type → Type := print : bool → PrintE unit.
```

```
1176 CoFixpoint t : ctree PrintE B2 void :=
1177   Vis (print true) (λ _ ⇒
1178     Vis (print false) (λ _ ⇒ t)).
```

```
1179
1180 CoFixpoint u : ctree PrintE B2 void :=
1181   br2
1182     (Vis (print true) (λ _ ⇒ u))
1183     (Vis (print false) (λ _ ⇒ u)).
```

```
1184 CoFixpoint u' : ctree PrintE B2 void :=
1185   br2
1186     (trigger (print true))
1187     (trigger (print false))
1188   ;;
1189   Guard u'.
```

Example CTrees t and u represent programs that repeatedly print booleans, with slightly different behaviors. t alternates printing `true` and `false`, while each iteration of u non-deterministically chooses a boolean and prints it. u' has the same semantics as u , but it is written in a different style: u exclusively uses the CTree constructors while u' uses higher-level `trigger` and `bind` (through the `;;` syntax) operators. In the second case, there is an additional `Guard` node as Coq needs a syntactic guard before co-recursive calls, but this

Fig. 17: The LTS for t (left) and u / u' (right)

has no consequence on the underlying LTS.¹⁴ The LTS underlying t and u are represented on Figure 17.

It is clear that program t is a refinement of program u . We establish this fact, $t \lesssim u$, through the following detailed proof steps, which correspond exactly to the Coq implementation (🔗). We write the current state of the proof goal to the right of the sequent, the coinduction hypothesis to its left, and the proof rule leading us there at the beginning of the line.

$$\begin{array}{l}
 1212 \quad \vdash t \lesssim u \\
 1213 \quad \xleftarrow{\text{coinduction}} t \mathcal{R} u \vdash t \lesssim_{\mathcal{R}} u \\
 1214 \quad \xleftarrow{\text{unfold}} t \mathcal{R} u \vdash \text{Vis} (\text{print true}) (\lambda _ \Rightarrow \text{Vis} (\text{print false}) (\lambda _ \Rightarrow t)) \lesssim_{\mathcal{R}} \\
 1215 \quad \text{br branch2} (\lambda b \Rightarrow \text{Vis} (\text{print b}) (\lambda _ \Rightarrow u)) \\
 1216 \quad \xleftarrow{\text{ss_br_r true}} t \mathcal{R} u \vdash \text{Vis} (\text{print true}) (\lambda _ \Rightarrow \text{Vis} (\text{print false}) (\lambda _ \Rightarrow t)) \lesssim_{\mathcal{R}} \\
 1217 \quad \text{Vis} (\text{print true}) (\lambda _ \Rightarrow u) \\
 1218 \quad \xleftarrow{\text{ss_vis_id}} t \mathcal{R} u \vdash \text{Vis} (\text{print false}) (\lambda _ \Rightarrow t) \mathcal{R} u \\
 1219 \quad \xleftarrow{\text{step}} t \mathcal{R} u \vdash \text{Vis} (\text{print false}) (\lambda _ \Rightarrow t) \lesssim_{\mathcal{R}} u \\
 1220 \quad \xleftarrow{\text{unfold}} t \mathcal{R} u \vdash \text{Vis} (\text{print false}) (\lambda _ \Rightarrow t) \lesssim_{\mathcal{R}} \\
 1221 \quad \text{br branch2} (\lambda b \Rightarrow \text{Vis} (\text{print b}) (\lambda _ \Rightarrow u)) \\
 1222 \quad \xleftarrow{\text{ss_br_r false}} t \mathcal{R} u \vdash \text{Vis} (\text{print false}) (\lambda _ \Rightarrow t) \lesssim_{\mathcal{R}} \\
 1223 \quad \text{Vis} (\text{print false}) (\lambda _ \Rightarrow u) \\
 1224 \quad \xleftarrow{\text{ss_vis_id}} t \mathcal{R} u \vdash t \mathcal{R} u
 \end{array}$$

The proof proceeds by coinduction: taking the singleton pair (t, u) as the simulation candidate \mathcal{R} ,¹⁵ we prove that $t \lesssim_{\mathcal{R}} u$, i.e., the pair (t', u') obtained after a simulation step is still in \mathcal{R} . After initializing the coinduction, we unfold one iteration of t and one iteration of u .¹⁶ At this point, CTree constructors appear in the goal. In particular the $\text{Vis} (\text{print true})$ at the head of the left-hand side should be matched against another $\text{Vis} (\text{print true})$ in

¹⁴ More formally, u and u' are in bisimulation.

¹⁵ Note that this candidate would not be a valid simulation without any enhancement.

¹⁶ Unfolding the body of a loop is a non-trivial operation that involves an unfolding lemma and the up-to-eq principle.

order to progress. This can be achieved by choosing the `true` outcome of the `br` on the right-hand side using `ss_br_r`. Then, the matching `Vis` (`print true`) can be stripped using the `ss_vis_id` theorem. Notice that the relation in the goal is no longer $\lesssim_{\mathcal{R}}$ but \mathcal{R} , as consuming `Vis` nodes performs a simulation step. But we want to perform one more simulation step to consume the second `Vis` of the left CTree, so we strengthen the proof goal from \mathcal{R} to $\lesssim_{\mathcal{R}}$ again. Then, the remaining `Vis` on the left can be matched using the same method as the first one, finally reducing to the goal $\tau \mathcal{R} u$, which is precisely our coinduction hypothesis, concluding the proof.

Through this example, we emphasize not the complexity of the result, but the fact that our infrastructure is robust enough to formally conduct, in Coq, a proof that is as simple as the detailed one we would write on paper.

For a similar pedagogical proof of bisimilarity of u and u' , illustrating the use of bisimulation up-to bisimilarity, we refer the interested reader to our development (👉).

4.7 Heterogeneous (bi)similarity

Throughout this section, we have defined various relations for comparing programs which essentially match transitions with identical labels—with the exception of weak bisimilarity that we have briefly mentioned, and come back to in Section 4.8. From the perspective of program verification, this is insufficient: `val v` labels may carry in `v` the memory configuration of our language, and we may need to express non-trivial relational invariants between such configuration, rather than enforcing equality.

The ITree library has had this notion since its inception, parameterizing their equivalence, `eutt`, by an arbitrary relation on leaves. When dealing with some more advanced reasoning, one may even wish to relate distinct external events. This led Silver et al. Silver et al. (2023) to introduce over ITrees the `rut` relation in order to express security invariants—Michelland et al. Michelland et al. (2024) have also made crucial use of this facility to relate concrete and abstract events to prove the soundness of abstract interpreters.

Although we have omitted for conciseness these details through our presentation, our library supports such a similar generalization for all the relations we have introduced. In our setting, we recover immediately the full generality of `rut` by parameterizing the relations by an arbitrary relation \mathcal{L} on labels. As an example, we reproduce below our definition of strong bisimilarity, and the other ones are generalized the same way.

Definition 5 (Bisimulation for CTrees, with an arbitrary relation on labels (👉)). *The progress function $sb \mathcal{L}$ for bisimilarity maps a relation \mathcal{R} over CTrees to the relation such that $sb \mathcal{L} \mathcal{R} s t$ holds if and only if:*

$$\forall l t', t \xrightarrow{l} t' \implies \exists l' u'. t' \mathcal{R} u' \wedge l \mathcal{L} l' \wedge u \xrightarrow{l'} u'$$

and conversely

$$\forall l u', u \xrightarrow{l} u' \implies \exists l' t'. t' \mathcal{R} u' \wedge l' \mathcal{L} l \wedge t \xrightarrow{l'} t'$$

Bisimilarity w.r.t. \mathcal{L} , written $t \sim_{\mathcal{L}} u$, is defined as the greatest fixpoint of $sb \mathcal{L}$: $sbisim \mathcal{L} \triangleq \text{gfp}(sb \mathcal{L})$.

Many of the proof rules and up-to principles introduced in the previous sections are generalized to this setting, and can be used seamlessly in presence of such a heterogeneous relation on labels. The most interesting rule to extend is the `bind` rule. In terms of usefulness first, it becomes a very general cut rule allowing for the introduction of an intermediate relational invariant on the values returned by the first parts of the computations. In terms of statement second, where we must be careful to introduce the necessary machinery to allow for this update to the `val` of the relation, while maintaining a consistent view on the other labels.

4.8 Weak bisimilarity

Weak bisimilarity, written $s \approx t$, is derived from the definition of the weak transition (👉). We define it as the standard asymmetric game (see for instance Chapter 4 of (Sangiorgi, 2012)), and omit details.

Our library currently offers restricted support for weak bisimilarity. This situation stems in part by needs: existing applications of CTree, including the examples in Sections 7 and 8, but also the artifact of Chappe et al. (2024) have so far only leveraged strong bisimilarity. While in the case of `ccs` is naturally only a matter of having not pushed the development of the case study further, the two other examples are more interesting: by using *Guard* in corecursive definition and careful definitions of the models, strong bisimilarity appears to be sufficient to recover a satisfying equivalence.

But the situation also comes from interesting technical challenges. As is well-known in the field of process algebra, weak bisimilarity can be unwieldy. Specifically, it is not a congruence for the `+` operator, of which the CTree *Br* nodes can be seen as a direct generalization. This has several consequences in the CTree setting. First, unlike strong bisimilarity, weak bisimilarity is not a congruence for *Br*: we cannot define a general proof rule on the same model. Furthermore, the up-to bind principle is not verified in the general case either, as it would, with well-chosen CTree, imply the former result.

As an example, consider the CTree t and the continuations k and k' below. We have $t \approx t$, $k \text{ true} \approx k' \text{ true}$, and $k \text{ false} \approx k' \text{ false}$; yet, we do not have $t \ggg k \approx t \ggg k'$.

$$\begin{aligned} t &:= Br^2 (\text{Ret } true) (\text{Ret } false) \\ k \text{ true} &:= \text{Ret } 2 \\ k \text{ false} &:= Br^2 (\text{Step } (\text{Ret } 0)) (\text{Step } (\text{Ret } 1)) \\ k' \text{ true} &:= \text{Ret } 2 \\ k' \text{ false} &:= \text{Step } (Br^2 (\text{Step } (\text{Ret } 0)) (\text{Step } (\text{Ret } 1))) \end{aligned}$$

This limitation of weak bisimilarity is commonly circumvented in process algebra by requiring processes to be *guarded*: the operands of the `+` operator should begin with a τ . With CTree terminology, it means that each branch of a *Br* should begin with a `Step`, which is precisely the definition of Br_S (👉). Similarly, we proved that an up-to bind principle that requires continuations to begin with `Step` is valid (👉).

$$\text{BIND}_W^{up}(\approx) R \triangleq \{(x \gg (\lambda x \Rightarrow \text{Step } (k \ x)), y \gg (\lambda x \Rightarrow \text{Step } (l \ x))) \mid \\ x \approx y \wedge \forall v, R(k \ v) (l \ v)\}$$

This principle is however not completely satisfying: we can note that it removes the guards in the co-recursive call. Transferred to an enhanced principle, this would translate to a bind-rule for `Step`-guarded continuations that *do not* unlock the coinduction hypothesis. We conjecture that the following more comfortable principle could be established, but we have not proved it at this point. It is particularly desirable, as it would be the key to adapting directly the `ITree` proofs of the iterative laws to the iterative laws for a `Step`-guarded `iter` against weak bisimilarity.

Conjecture:

$$\text{BIND}_W^{up}(\approx) R \triangleq \{(x \gg (\lambda x \Rightarrow \text{Step } (k \ x)), y \gg (\lambda x \Rightarrow \text{Step } (l \ x))) \mid \\ x \approx y \wedge \forall v, R(\text{Step}(k \ v)) (\text{Step}(l \ v))\}$$

5 Reasoning on a *Br*-aware LTS

In some cases, (bi)simulation proofs that involve *Br* nodes can be unwieldy. Consider the following property: $\forall t, \text{spin} \lesssim t$. We could of course simply apply the rule `ss_stuck`, but let us try to proceed by coinduction, using the rules from Figure 14.

$$\begin{array}{c} \vdash \forall t, \text{spin} \lesssim t \\ \xleftarrow{\text{coinduction}} \forall t, \text{spin } \mathcal{R} \ t \vdash \forall t, \text{spin} \lesssim_{\mathcal{R}} t \\ \xleftarrow{\text{unfold}} \forall t, \text{spin } \mathcal{R} \ t \vdash \forall t, \text{Guard } \text{spin} \lesssim_{\mathcal{R}} t \\ \xleftarrow{\text{step_ss_guard_1}} \forall t, \text{spin } \mathcal{R} \ t \vdash \forall t, \text{spin} \lesssim_{\mathcal{R}} t \end{array}$$

After initializing the coinduction, we can unfold one iteration of `spin` and use the relevant proof rule to remove the `Guard`, but this gives back the original proof state. Rather, in the middle of our game, we need to proceed by induction over the transition considered—in this case to realize there are none such transition, but more generally to capture enough information on the state this transition can reach. While in this trivial case, it is manageable, it becomes extremely painful in more complex theorems that require proper nesting of coinductive and inductive reasoning, including many results established on `iter` and `interp` from Section 6.

This need for inductive reasoning is a common problem, encountered with many flavor of weak (bi)similarity built on weak transition systems (van Glabbeek, 1993; Sangiorgi, 2012). For instance for weak bisimilarity, a natural definition would be to define it as the strong bisimulation game over the weak LTS, quantifying over weak challenges forces undesired inductive reasoning upon us. The standard solution is to observe that the same

label $\triangleq \epsilon \mid \text{tau} \mid \text{obs } e \ v \mid \text{val } v$

$$\frac{}{Br^b \ k \xrightarrow{\epsilon} t} \quad \frac{}{\text{Guard } t \xrightarrow{\epsilon} u} \quad \frac{}{\text{Step } t \xrightarrow{\text{tau}} t} \quad \frac{}{Vis \ e \ k \xrightarrow{\text{obs } e \ v} k \ v} \quad \frac{}{\text{Ret } v \xrightarrow{\text{val } v} \emptyset}$$

Fig. 18: Inductive characterization of the alternative ϵ -LTS induced by a CTree. The *Br* and *Guard* cases differ from the original LTS.

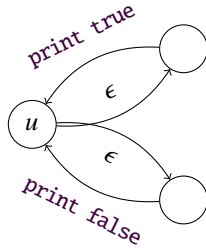


Fig. 19: The LTS built from the CTree *u* of Section 4.6, with explicit ϵ transitions

relation can be generated by an asymmetric game, where challenges are strong (Sangiorgi, 2012)—which we indeed follow in our definition of the weak bisimulation.

Similarly, section 5.1 develops an alternative characterization of CTree simulation based on this principle, yielding a standard definition of similarity. Section 5.2 does the same for CTree bisimulation, which gives an unusual definition of bisimilarity based on mutual coinduction.

5.1 Alternative characterization of CTree similarity

We have defined the strong simulation game in a standard and intuitive way (see Definition 3), and it admits convenient proof rules and up-to principles. However, it is defined on a slightly unusual LTS where all the *Br* nodes are skipped/collapsed. An unfortunate consequence is that we *cannot* reason at the level of the simulation on these *Br* nodes, as they do not even appear in the LTS. Alternatively, in this section, we consider LTSs in which *Br* nodes generate a special ϵ transition, and define a fitting simulation game that “ignores” these ϵ transitions, enabling finer-grained reasoning. Figure 18 shows the definition of this alternative ϵ -LTS. Considering the example CTree *u* from Section 4.6, the resulting finer-grained LTS is shown on Figure 19.¹⁷

With this definition, the previously introduced strong transition (strong w.r.t. τ transitions) becomes weak with respect to ϵ transitions. Figure 20 depicts the simulation game *ss* introduced in Section 4.5.1, but from the perspective of the ϵ -LTS. We note $t \xrightarrow{\epsilon} u$ for ϵ transitions and $t \xrightarrow{l} u$ for other transitions. This simply explicit that the simulation challenge

¹⁷ For historical reasons, this LTS is not explicitly defined in our Coq development, but rather baked in the definition of the alternative simulation game. Making it explicit would make many Coq definitions described in this section cleaner, and closer to the definitions in this paper, but as this has no consequence on the theory, we leave this refactoring for future work.

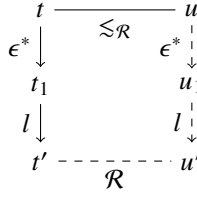


Fig. 20: The simulation game and bisimulation half-game $t \lesssim_{\mathcal{R}} u$, in the ϵ -LTS



Fig. 21: The two cases of the simulation game $t \lesssim'_{\mathcal{R}} u$

we have been considering may involve an arbitrary number of ϵ transitions before reaching an observable label, which is indicative of a flavor of a weak simulation game, *defined as a strong simulation game over a flavor of a weak transition system*. And as illustrated in the toy example proof above, the awkwardness in the game translates in the proof system: consider Figure 14, the proof rule for Br does not perform a coinductive step.

By switching our perspective over the ϵ -LTS, it is possible to provide an definition of CTree simulation equivalent to $ssim$, while avoid its drawbacks: Figure 21 shows its game (🔴). The key to the definition of this new game ss' (written $\lesssim'_{\mathcal{R}}$) is to distinguish apart ϵ -transition as a special kind of challenge. While other transitions are handled as in the original definitions, ϵ transition can be answered by any number of ϵ transitions, possibly 0. This distinction is standard in the context of weak simulations.

We note \lesssim' for the induced notion of similarity $\mathbf{gfp} \ ss'$. Of course, a critical result is the equivalence of this notion of similarity with the original notion of strong similarity.

Lemma 6. *Equivalence of the two notions of similarity (🔴).*

$$\forall t u, t \lesssim u \iff t \lesssim' u$$

All the proof rules for ss are also proved valid on ss' , and more powerful rules are valid for Br and Guard nodes. Among the newly valid proof rules is one that performs a coinductive step when encountering a Guard node, while an induction nested in the coinductive proof would have been needed if working with ss .

$$\frac{t \lesssim_{\mathcal{R}} u}{\text{Guard } t \lesssim_{\mathcal{R}} u} \text{ (ss_guard_1)} \qquad
 \frac{t \mathcal{R} u}{\text{Guard } t \lesssim'_{\mathcal{R}} u} \text{ (ss'_guard_1)}$$

The up-to-bind and up-to-equ principles are valid for ss' . However, the up-to-sbisim principle that allows rewriting top-level strongly bisimilar terms is no longer valid. Indeed, rewriting using the theorem $\mathbf{sb_guard} : \text{Guard } t \sim t$ would allow introducing a guard node and coinductively removing it using $\mathbf{step_ss'_guard_1}$. This echoes the classical result

that weak (bi)simulation is not valid up-to weak (bi)simulation when presented as generated by the asymmetric game. Fortunately, it is still possible to strip Guard nodes using a dedicated up-to principle:

$$\epsilon_r^{up} R \triangleq \{(x, y) \mid y' \xrightarrow{\epsilon^*} y \wedge \forall v, x \mathcal{R} y'\}$$

This up-to principle is used to recover asymmetric Guard- and Br-stripping proof rules for the right-hand side of the simulation.

Lemma 7. *Enhanced coinduction for \lesssim' (♣) The functions REFL^{up} (♣), $\text{BIND}^{up}(\lesssim)$ (♣), ϵ_r^{up} (♣), ss (♣) and $\text{UPTO}^{up}(\cong)$ (♣) provide valid up-to principles for \lesssim' .*

Another interesting new up-to principle is ss , the original strong simulation game. In fact, it is more than an up-to principle. At any point during the proof of a $ssim'$ simulation, we can perform a regular ss step *instead of* an ss' step. This is because an ss step always corresponds to one or more ss' steps. To state this fact formally, we have to leak the implementation details of our relations in terms of tower induction:

Lemma 8. *ss is a sub-chain of ss' (♣) Given $\mathcal{R} : C_{ss'}$, i.e., a chain for ss' , the following implication holds.*

$$\forall t u, ss \mathcal{R} t u \implies t \lesssim'_{\mathcal{R}} u$$

We can now revisit our motivating example from Section 5, and conclude via a purely coinductive proof (♣).

$$\begin{aligned} & \vdash \forall t, \text{spin} \lesssim t \\ & \Leftarrow \vdash \forall t, \text{spin} \lesssim' t \\ & \xleftarrow{\text{coinduction}} \forall t, \text{spin} \mathcal{R} t \vdash \forall t, \text{spin} \lesssim'_{\mathcal{R}} t \\ & \xleftarrow{\text{unfold}} \forall t, \text{spin} \mathcal{R} t \vdash \forall t, \text{Guard spin} \lesssim'_{\mathcal{R}} t \\ & \xleftarrow{\text{step_ss_guard_1}} \forall t, \text{spin} \mathcal{R} t \vdash \forall t, \text{spin} \mathcal{R} t \end{aligned}$$

5.2 Alternative characterization of CTree bisimilarity

Naturally, we want a similar improvement for strong bisimilarity. However, while the solution in the case of similarity turned out to be a fairly standard recipe, bisimilarity calls for more inventivity. Indeed, bisimulation games are usually defined as the intersection of the simulation game and its symmetrized version—that was the case for the bisimulation game in Section 4.4. However, this would not work for the alternative game. Consider the CTrees in Figure 22. They are bisimilar since they have the same available transitions, $(\text{val } i)$ for $i \in \{0, 1, 2, 3\}$, to the empty state. But a naïve symmetrization of \lesssim' would require an ϵ challenge to be matched by *exactly* one ϵ transition, which is not possible in

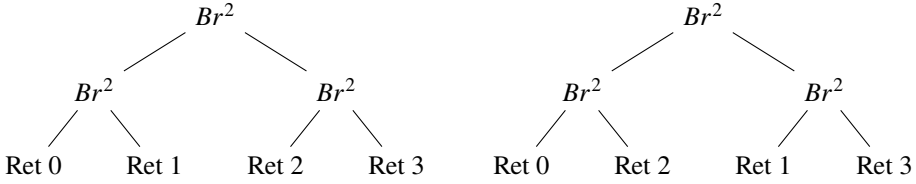


Fig. 22: Two bisimilar ctree

the example. No matter which branch of the left CTree is taken, there is no branch in the right CTree that is bisimilar with the intermediate node. Hence, this definition would be too strong.

Rather, we define an alternative bisimulation over two *mutually coinductive* relations: intuitively, one for the left half-game, and another one for the right half-game. The intersection of the greatest fixpoints of these relations gives the bisimulation relation. This construction is reminiscent of *coupled simulations* (Parrow and Sjödin, 1994; Sangiorgi, 2012), though the principles are not comparable. A notion of bisimilarity based on mutual coinduction has been proposed before in the context of applicative bisimilarity (Levy, 2006).

To define formally our alternate bisimilarity, we define the pair of games we consider by indexing them by a boolean, encoding which half we are currently participating in. While the games encountered so far were monotone function over binary CTree relations, \sim'_\square is a monotone function over ternary `bool * ctree * ctree` relations \mathcal{R} . In the following we note \mathcal{R}_l for \mathcal{R} true, \mathcal{R}_r for \mathcal{R} false, and \mathcal{R}_{lr} for the intersection of \mathcal{R}_l and \mathcal{R}_r . Similarly, we write $\sim'_{\mathcal{R}_l}$ for $sb' \mathcal{R}$ true, $\sim'_{\mathcal{R}_r}$ for $sb' \mathcal{R}$ false, and $\sim'_{\mathcal{R}_{lr}}$ for their intersection.

The bisimulation relation $t \sim' u$ is defined as $\forall \text{side}:\text{bool}, (\text{gfp } \sim'_\square) \text{ side } t u$. The “ $\forall \text{side}$ ” is critical as it requires a CTree pair to be both in \mathcal{R}_l and \mathcal{R}_r to be considered bisimilar. Finally, we write $t \sim'_l u$ for $\text{gfp } \sim'_\square \text{ true } t u$ and $t \sim'_r u$ for $\text{gfp } \sim'_\square \text{ false } t u$.

The definition of $\sim'_{\mathcal{R}_l}$ and $\sim'_{\mathcal{R}_r}$ (depicted in Figure 23) is similar to the one of $\lesssim'_{\mathcal{R}}$, but the ϵ case in the bisimulation left (resp. right) half-game has a weaker conclusion: it only leads to the left (resp. right) half-relation. When the head of a CTree is an ϵ node, both the left and right half games need to be played (possibly several times) to get back to $\mathcal{R}_l \cap \mathcal{R}_r$.

For readers familiar with ITrees (Xia et al., 2020), our boolean parameter may evoke a proof device that was used in the definition of `eqit`, a coinductive relation parameterized by two booleans. However, the point of the booleans in `eqit` was to factor out four similar definitions of simulation and bisimulation (`eutt`, etc.). In this context, the booleans were *outside* the greatest fixpoint, while our boolean is not fixed and does evolve during a coinductive proof.

The main proof rules are shown on Figure 24.¹⁸ The ones related to Guard and *Br* are more powerful than the *sb* rules, and the other ones (greyed out) are equivalent.

As usual, various up-to principles are valid, but this time \mathcal{R} is not a binary but a ternary relation (because of the additional boolean). For each of the up-to principles proved for \lesssim' in Lemma 7, we proved the validity of a ternary counterpart for \sim' .

¹⁸ Note: for technical reasons, the rules involving Br_S nodes are only valid when the relation \mathcal{R} is an element of the chain $C_{sb'}$. This is completely transparent, as we never consider any other relation.

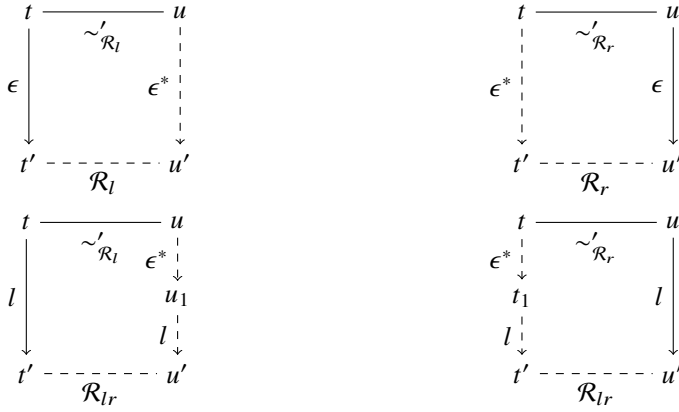


Fig. 23: The four cases of the bisimulation game $t \sim'_{\mathcal{R}_{lr}} u$. Note that the left and right half-games are symmetric, with \mathcal{R}_l and \mathcal{R}_r swapped.

$$\begin{array}{c}
 \text{Ret } v \sim'_{\mathcal{R}_{lr}} \text{Ret } v \quad \frac{\forall v, (k v) \mathcal{R}_{lr} (k' v)}{\text{Vis } e k \sim'_{\mathcal{R}_{lr}} \text{Vis } e k'} \quad \frac{\forall x, (k x) \mathcal{R}_l u}{\text{Br}^c k \sim'_{\mathcal{R}_l} u} \quad \frac{\forall x, t \mathcal{R}_r (k x)}{t \sim'_{\mathcal{R}_r} \text{Br}^c k} \\
 \\
 \frac{(\forall x, \exists y, (k x) \sim'_{\mathcal{R}_{lr}} (k' y)) \wedge (\forall y, \exists x, (k x) \sim'_{\mathcal{R}_{lr}} (k' y))}{\text{Br}^c k \sim'_{\mathcal{R}_{lr}} \text{Br}^d k'} \quad \frac{\forall v, (k v) \mathcal{R}_{lr} (k' v)}{\text{Br}^c k \sim'_{\mathcal{R}_{lr}} \text{Br}^c k'} \\
 \\
 \frac{t \mathcal{R}_{lr} u}{\text{Guard } t \sim'_{\mathcal{R}_{lr}} \text{Guard } u} \quad \frac{t \mathcal{R}_{lr} u}{\text{Step } t \sim'_{\mathcal{R}_{lr}} \text{Step } u} \\
 \\
 \frac{(\forall x, \exists y, (k x) \mathcal{R} (k' y)) \wedge (\forall y, \exists x, (k x) \mathcal{R} (k' y))}{\text{Br}_S^c k \sim'_{\mathcal{R}_{lr}} \text{Br}_S^d k'} \quad \frac{\forall v, (k v) \mathcal{R}_{lr} (k' v)}{\text{Br}_S^c k \sim'_{\mathcal{R}_{lr}} \text{Br}_S^c k'}
 \end{array}$$

Fig. 24: Proof rules for coinductive proofs of \sim' (♣)

Lemma 9. *Enhanced coinduction for \sim' (♣). The up-to reflexivity (♣), up-to equ (♣), up-to bind (♣/♣), up-to ss (♣) and up-to epsilon (♣) principles are valid for \sim' . A new up-to negated symmetry $\text{NSYM}_3^{\text{up}}$ principle is also valid (♣).*

$$\text{NSYM}_3^{\text{up}} R \triangleq \{(b, x, y) \mid R \neg b y x\}$$

The up-to negated symmetry principle is a ternary variant of the standard up-to symmetry principle. It swaps two CTrees operands of a relation and negates its boolean, because after swapping the operands, \mathcal{R}_l and \mathcal{R}_r become inverted.

Again, a major result on this alternative characterization of bisimilarity is its equivalence with the bisimilarity previously presented in Section 4.4. Interestingly, the theorem statement can be split into a result on \sim'_l and one on \sim'_r .



Fig. 25: The two cases of the homogeneous alternative bisimulation game $t \sim'_{\mathcal{R}} u$. \mathcal{R}° represents the converse of \mathcal{R} .

Theorem 1. *Equivalence of the two notions of bisimilarity* (🔴).

$$\begin{aligned} \forall t u, t \sim u &\iff t \sim' u \\ \forall t u, ss \text{ (sbisim } t u) &\iff t \sim'_l u \\ \forall t u, ss \text{ (sbisim } u t) &\iff t \sim'_r u \end{aligned}$$

We have presented alternative characterizations of similarity and bisimilarity in the homogeneous case, but as with most definitions of Section 4, we implemented them in Coq with support for heterogeneous relations. In fact, in the homogeneous case, the bisimulation game of Figure 23 degenerates to a simpler game (Figure 25) that does not rely on mutual coinduction, nor a ternary relation with a boolean. This simpler game stems from the observation that an homogeneous \mathcal{R} verifies $\mathcal{R} b t u \iff \mathcal{R} \neg b u t$ (🔴).

6 Interpretation from and to CTrees

The ITree ecosystem fundamentally relies on the incremental interpretation of effects, represented as external events, into their monadic implementations. Through this section, we show how CTrees fit into this narrative both by supporting the interpretation of their own external events, and by being a suitable target monad for ITrees, for the implementation of nondeterministic branching.

6.1 Interpretation

ITrees support interpretation: provided a *handler* $h:E \rightsquigarrow \mathbf{M}$ implementing its signature of events E into a suitable monad \mathbf{M} , the $(\text{interp } h):\text{itree } E \rightsquigarrow \mathbf{M}$ combinator provides an implementation of any computation into \mathbf{M} . The only restriction imposed on the target monad \mathbf{M} is that it must support its own *iter* combinator, i.e., be iterative, so that divergence, modelled coinductively in the tree, can also be internalized in \mathbf{M} . For this implementation to be sensible and amenable to verification in practice, one must, however, check an additional property: $\text{interp } h$ should form a monad morphism—in particular, it should map eutt ITrees to equivalent monadic computations in \mathbf{M} .

Unsurprisingly, given their structure, CTrees enjoy their own *interp* combinator. Its definition, provided in Figure 26, is very close to its ITree counterpart. The interpreter relies on the target monad's own *iter* to chain the implementations of the external events in the process. But additionally to being iterative, i.e., being able to internalize divergence,

```

1657 Definition interp (h : E ~ M) : ctree E B ~ M := λR ⇒
1658   miter (λ t ⇒ match t with
1659     | Ret r ⇒ ret (inr r)
1660     | Stuck ⇒ mStuck
1661     | Guard t ⇒ ret (inl t)
1662     | Step t ⇒ bind mstep (λ _ ⇒ inl t)
1663     | Br n k ⇒ bind (mBr n) (λ x ⇒ ret (inl (k x)))
1664     | Vis e k ⇒ bind (h e) (λ x ⇒ ret (inl (k x)))
1665   end).

```

Fig. 26: Interpreter for CTrees (class constraints omitted) (🔒)

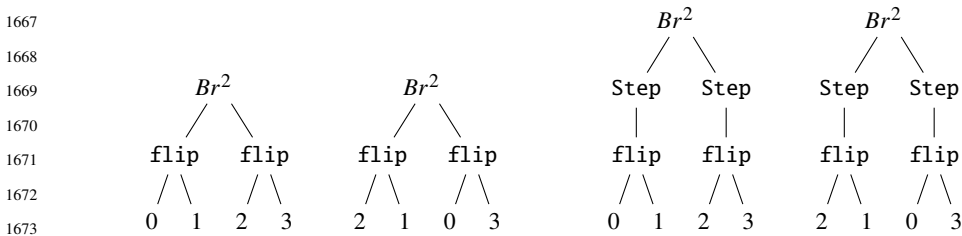


Fig. 27: Two strongly bisimilar trees before interpretation (left), but not after (right)

the target monad but also be able to internalize non-determinism by providing a stuck state (`mStuck`), an observable tick (`mStep`), and an internal branching (`mBr`). We provide straightforward instances for CTrees and stateful interpretations.

Perhaps more surprisingly, the requirement that `interp h` defines a monad morphism unearths interesting subtleties. Let us consider the elementary case where the interface `E` is implemented in terms of (possibly pure) uninterpreted computations, that is when $M := \text{ctree } F B$. The requirement becomes: $\forall t u, t \sim u \rightarrow \text{interp } h t \sim \text{interp } h u$. But this result does not hold for an arbitrary `h`: intuitively, our definition for `sbisim` has implicitly assumed that implementations of external events may eliminate reachable states in the computation's induced LTS—through pure implementations—but should not be allowed to introduce new ones.

The counter-example in Figure 27, where `flip` is the binary event introduced in Section 2.2, fleshes out this intuition. Indeed, both trees are strongly bisimilar: each of them can emit the label `obs flip false` by stepping to either the `Ret 0` or `Ret 2` node, or emit the label `obs flip true` by stepping to either the `Ret 1` or `Ret 3` node. However, they are strongly bisimilar because the induced LTS processes the question to the environment—`flip`— and its answer—`false/true`—in a single step, such that the computations never observe that they have had access to distinct continuations. However, if one were to introduce in the tree a `Step` node before the external events, for instance using the handler $h := \lambda e \Rightarrow \text{Step } (\text{trigger } e)$, a new state allowing for witnessing the distinct continuations would become available in the LTSs, leading to non-bisimilar interpreted trees.

We hence say a handler $h : E \rightsquigarrow \text{ctree } F B$ is *quasi-pure* if it implements each event `e` either as a pure computation, i.e., $\forall l t', h e \xrightarrow{l} t' \implies \exists r, l = \text{val } r$, or always performs exactly one step before returning, i.e., $\forall l t', h e \xrightarrow{l} t' \implies \exists r, t' = \text{Ret } r$. Similarly, we say

that $h : E \rightsquigarrow \text{stateT } S \text{ (ctree } F \ B)$ is quasi-pure if it is point-wise quasi-pure. We show that we recover the desired property for the subclass of quasi-pure handlers:

Theorem 2 (Quasi-pure handlers interpret into monad morphisms (♣)).

- If $h : E \rightsquigarrow \text{ctree } F \ B$ is quasi-pure, then

$$\forall t \ u, t \sim u \rightarrow \text{interp } h \ t \sim \text{interp } h \ u.$$

- If $h : E \rightsquigarrow \text{stateT } (\text{ctree } F \ B)$ is quasi-pure, then

$$\forall t \ u, t \sim u \rightarrow \forall s, \text{interp } h \ t \ s \sim \text{interp } h \ u \ s.$$

The same is true for \lesssim .

The stateful version is, in particular, sufficient to transport `ImpBr` equations established before interpretation—such as the theory of `br _ or _`—through interpretation (♣). More generally, we can reason after interpretation to establish equations relying both on the nondeterminism and state algebras, for instance to establish the equivalence $p_3 \equiv \text{br } p_2 \text{ or } p_3$ mentioned in Section 2.2 (♣).

The former theorem links `CTrees` before and after interpretation. It can also be interesting to compare alternative implementations of a handler for the same kind of event, e.g. different memory models for memory access events. In this case, we provide a theorem to lift a simulation result on handlers to a simulation result on interpreted `CTrees`.

Theorem 3 (A simulation between handlers can be lifted through interpretation (♣)).

- $\forall (h \ h' : E \rightsquigarrow \text{ctree } F) \ t, (\forall e, h \ e \lesssim h' \ e) \implies \text{interp } h \ t \lesssim \text{interp } h' \ t.$
- $\forall (h \ h' : E \rightsquigarrow \text{stateT } (\text{ctree } F)) \ t, (\forall e \ s, h \ e \ s \lesssim h' \ e \ s) \implies \text{interp } h \ t \ s \lesssim \text{interp } h' \ t \ s.$

This theorem is presented here in the homogeneous case for simplicity, but we provide it for heterogeneous simulations, thus the heavily-parameterized Coq statement of the mechanized theorem. Note that unlike Theorem 2, we do not need any assumption on the handlers or the trees.

6.2 Refinement

Interpretation provides a general theory for the implementation of external events. Importantly, `CTrees` also support an analogous facility for the *refinement* of its internal branches: one can shrink the set of accessible paths in a computation—and, in particular, determinize it.

We provide, to this end, a new combinator, `refine`, defined in Figure 28. The definition is very similar¹⁹ to `interp`, except that it takes as an argument a handler specifying how

¹⁹ In our development, `interp` and `refine` are defined as special case of a fold operator allowing for the simultaneous implementation of both external events and internal branches.

```

1749 Definition refine (h : B  $\rightsquigarrow$  M) : ctree E B  $\rightsquigarrow$  M :=  $\lambda$ R  $\Rightarrow$ 
1750 miter ( $\lambda$  t  $\Rightarrow$  match t with
1751 | Ret r  $\Rightarrow$  ret (inr r)
1752 | Stuck  $\Rightarrow$  mStuck
1753 | Guard t  $\Rightarrow$  ret (inl t)
1754 | Step t  $\Rightarrow$  bind mstep ( $\lambda$  _  $\Rightarrow$  inl t)
1755 | Br b k  $\Rightarrow$  bind (h b) ( $\lambda$  x  $\Rightarrow$  ret (inl (k x)))
1756 | Vis e k  $\Rightarrow$  bind (mTrigger e) ( $\lambda$  x  $\Rightarrow$  ret (inl (k x)))
1757 end).

```

Fig. 28: Refining CTrees (class constraints omitted) (🔥)

to implement branches rather than external events into a monad M . The target monad must naturally still be iterative, able to provide a stuck state, and an observable tick, but must additionally explain how it can re-embed an uninterpreted event (`mtrigger`)—a device already used in (Yoon et al., 2022).

As hinted at by the combinator’s name, the source program should be able to simulate the refined program. Fixing M to `ctree F B`, this is expressed as $\forall t, \text{refine } h t \lesssim t$. However, one cannot hope to obtain such a result for an arbitrary h , because it could implement internal branches with an observable computation that can’t be simulated by t . We hence prove this result for refinement handlers implementing branches in terms of *finite pure CTrees* (🔥): finite-height CTrees that do not contain any `Step` or `Vis` node.

Lemma 10 (Finite pure refinements are proper refinements (🔥) ²⁰).

- $\forall h t, h \text{ pointwise pure finite} \implies \text{refine_cst } h t \lesssim t$
- $\forall h t s, h \text{ pointwise finite pure} \implies \text{refine_state } h t s \lesssim_{st} t$

Unlike `interp`, `refine` is *not* a monad morphism. The CTrees Br^2 (`Ret 0`) (`Ret 1`) and Br^2 (`Ret 1`) (`Ret 0`) are clearly bisimilar, but refining them by always choosing the left branch of `Br` nodes gives `Ret 0` and `Ret 1`, which are not bisimilar. This highlights the major difference between `Vis` and `Br`: `Vis` nodes represent external events whose response from the environment has a strong semantic value, while `Br` nodes represent internal nondeterminism, with semantically indistinguishable branches.

6.3 Extraction

The shallow nature of CTrees also offers testing opportunities. Xia et al. (2020) describe how external events such as IO interactions can alternatively be implemented in OCaml and linked against at extraction. Similarly, we demonstrate on `ImpBr` how to execute a CTree by running an impure refinement implemented in OCaml by picking random branches along the execution.

In comparison with ITrees, the random execution of CTrees requires some care because of their nondeterministic nature. The naïve approach (🔥) of randomly choosing a branch

²⁰ The second theorem is an heterogeneous simulation (see Section 4.7): the return types of the trees are not identical — the refined tree maintains an additional state that we ignore.

```

1795 let rec run t =
1796   match observe t with
1797     | RetF r -> print_int (int_of_nat r); true
1798     | BrF (_, k) ->
1799       let b = Random.bool() in
1800       if run (k (Obj.magic b)) then true
1801       else run (k (Obj.magic (not b)))
1802     | GuardF t -> run t
1803     | StuckF -> false
1804     | _ -> failwith "unreachable";;
1805
1806 let rec collect t =
1807   match observe t with
1808     | RetF r -> [int_of_nat r]
1809     | GuardF t -> collect t
1810     | StuckF -> []
1811     | BrF (_, k) ->
1812       collect (k (Obj.magic true)) @ collect (k (Obj.magic false))
1813     | _ -> failwith "unreachable";;

```

Fig. 29: A random interpreter and a collecting interpreter for ImpBr, implemented in OCaml.

when encountering a *Br* node is not semantically correct because of stuck branches: the semantics of $Br^2 \emptyset t$ should be the semantics of t . We provide a correct implementation for the ImpBr example (🔥) that backtracks when it encounters a stuck branch.

This fixed version still chooses *Br* branches randomly, but if it subsequently reaches a stuck node (materialized by a `false` return value), it explores the other branch. Again on the ImpBr example, we can define a collecting interpreter that, given an ImpBr program, crawls its CTree to build the list of its possible return values.

We observe that these interpreters exhibits correct behavior on an example CTree (🔥). However, a limitation of the proposed implementations is that they may loop when given a program with an infinite chain of *Guard* or *Br* nodes (e.g., the `spin` CTree from Figure 6). If it reaches a `spin`, the random interpreter will loop on the *Guard* case and never terminate. As for the collecting interpreter, it will always loop as it always explores every branch of the CTree. For the random interpreter, performing a breadth-first search instead of a depth-first search would solve this limitation. But this is not an option for the collecting interpreter. Instead, the maximal exploration depth could be limited as a workaround.

6.4 ITree embedding

We have used CTrees directly as a domain to represent the syntax of ImpBr, as well as in our case studies (see Section 7 and 8). CTrees can, however, fulfill their promise sketched in Section 2, and be used as a domain to host the monadic implementation of external representations of nondeterministic events in an ITree.

To demonstrate this approach, we consider the family of events `choose (n: nat) : Choose (fin n)`, and aim to define an operator `embed` taking an ITree computation modeling nondeterministic branching using these external events, and


```

1841 Definition inject {E} : itree E  $\rightsquigarrow$  ctree E := interp ( $\lambda e \Rightarrow$  trigger e).
1842 Definition internalize {E} : ctree (Choose +' E)  $\rightsquigarrow$  ctree E :=
1843   interp ( $\lambda e \Rightarrow$  match e with | inl1 (choose n)  $\Rightarrow$  BrSn | inr1 e  $\Rightarrow$  trigger
1844     e).
1845 Definition embed {E} : itree (Choose +' E)  $\rightsquigarrow$  ctree E :=
1846    $\lambda \_ t \Rightarrow$  internalize (inject t).

```

Fig. 30: Implementing external branching events into the CTree monad (👉)

implementing them as branches indexed similarly into a CTree. This operator, defined in Figure 30, is the composition of two transformations. First, we `inject` ITrees into CTrees by (ITree) interpretation. This injection rebuilds the original tree as a CTree, where `Stepi` nodes have become `Guard` nodes, and an additional `Guard` has been introduced in front of each external event. Second, we `internalize` the external branching contained in a CTree implementing a `Choose` event, using the isomorphic stepping branch. The resulting embedding forms a monad morphism transporting `eutt` ITrees into `sbisim` CTrees:

Lemma 11. `embed` respects `eutt` (👉) $\forall t u, \text{eutt } t u \implies \text{embed}(t) \sim \text{embed}(u)$

The proof of this theorem highlights how `Stepi` nodes in ITrees (adding a subscript to distinguish them from their CTree homonym) collapse two distinct concepts that nondeterminism forces us to unravel in CTrees. The `eutt` relation is defined as the greatest fixpoint of an inductive endofunction `euttF`. In particular, one can recursively and *asymmetrically* strip finite amounts of `Stepi`—the corecursion is completely oblivious to these nodes in the structures. Corecursively, however, `Stepi` nodes can be matched *symmetrically*—a construction that is useful in exactly one case, namely to relate the silently spinning computation, `Stepiω`, to itself. From the CTree perspective, recursing in `euttF` corresponds to recursion in the definition of the LTS: `Stepi` nodes are `Guard` nodes. But corecursing corresponds to a step in the LTS: `Stepiω` corresponds to `Stepω`. ITrees' `Stepi` thus corresponds to either a `Guard` or a `Step`. Nondeterminism forces us to separate both concepts, as whether a node in the tree constitutes an accessible state in the LTS becomes semantically relevant. We obtain two "dual" notions that have no equivalent in ITrees: we may have finite amounts of asymmetric corecursive choices, i.e., finitely many `Steps`, and structurally infinite stuck processes, i.e., `Guardω`.

In the proof of Lemma 11, this materializes by the fact that an induction on `euttF` leaves us disappointed in the symmetric `Stepi` case: we have no applicable induction hypothesis, but expose in our embedding a `Guard`, which does not allow us to progress in the bisimulation. We must resolve the situation by proving that being able to step in `embed t` implies that `t` is not `Stepiω`, i.e., that we can inductively reach a `Vis` or a `Ret` node (👉).

Limitations: on guarding recursive calls using `Guard`. We have shown that CTree equipped with `iter` as recursor and strong bisimulation as equivalence form an iterative monad. Furthermore, building interpretation atop this `iter` combinator gives rise to a monad morphism respecting `eutt`, hence is suitable for implementing non-deterministic effects represented as external in an ITree.

1887 However, we stress that the underlying design choice in the definition of `iter`, guarding
 1888 recursion using `Guard`, is not without consequences. It has strong benefits, mainly that a lot
 1889 of reasoning can be performed against strong bisimulation. More specifically, this choice
 1890 allows the user to reserve weak bisimulation for the purposes of ignoring domain-specific
 1891 steps of computations that may be relevant both seen under a stepping or non-stepping
 1892 lens—e.g., synchronizations in `ccs`—but it does not impose this behavior on recursive
 1893 calls. However, it also leads to a coarse-grained treatment of silent divergence: in particular,
 1894 the silently diverging `ITree` (an infinite chain of `Stepi`) is embedded into an infinite chain of
 1895 `Guard`, which, we have seen, corresponds to a stuck LTS. For some applications—typically,
 1896 modeling other means of being stuck and later interpreting them into a nullary branch—
 1897 one would prefer to embed this tree into the infinite chain of `Step` to avoid equating both
 1898 computations. While one could rely on manually introducing a `Step` in the body iterated
 1899 upon when building the model, that approach is a bit cumbersome.

1900 Instead, a valuable avenue would be to develop the theory accompanying the alter-
 1901 nate iterator mentioned in Section 4.4 and guarding recursion using `Step`. Naturally, the
 1902 corresponding monad would not be iterative with respect to strong bisimulation, but we con-
 1903 jecture that it would be against weak bisimulation. From this alternate iterator would arise
 1904 an alternative embedding of `ITrees` into `CTrees`: we conjecture it would still respect `eutt`,
 1905 but seen as a morphism into `CTrees` equipped with weak bisimulation. The development
 1906 accompanying this paper does not yet support this alternate iterator, we leave implementing
 1907 it to future work.

1908 Currently, the user has the choice between (1) not observing recursion at all, but getting
 1909 away with strong bisimulation in exchange, (2) manually inserting `Step` at recursive calls
 1910 that they chose to observe. With support for this alternate iterator, the user would be given
 1911 additional option to (3) systematically `tau`-observe recursion, at the cost of working with
 1912 weak bisimulation everywhere.

1913 7 Case study: a model for `ccs`

1914 We claim that `CTrees` form a versatile tool for building semantic models of nondeterministic
 1915 systems, concurrent ones in particular. In this section, we illustrate the use of `CTrees` as
 1916 a model of concurrent communicating processes by providing a semantics for Milner’s
 1917 Calculus of Communicating Systems (`ccs`) (Milner, 1989). The results we obtain—the
 1918 usual algebra, up-to principles, and precisely the same equivalence relation as the usual
 1919 operational-based strong bisimulation—are standard, per se, but they are all established by
 1920 exploiting the generic notion of bisimilarity of `CTrees`. The result is a shallowly embedded
 1921 model for `ccs` in Coq that could be easily, and modularly, combined with other language
 1922 features.
 1923
 1924
 1925
 1926

1927 7.1 Syntax and operational semantics

1928 The syntax and operational semantics of `ccs` are shown in Figure 31. The language assumes
 1929 a set of *names*, or communication channels, ranged over by c . For any name c , there is a
 1930 co-name \bar{c} satisfying $\bar{\bar{c}} = c$. An *action* is represented by a label l ; it is either a communication
 1931
 1932

$$\begin{array}{l}
1933 \quad l ::= \tau \mid c \mid \bar{c} \quad P ::= 0 \mid l \cdot P \mid P \oplus Q \mid P \parallel Q \mid \nu c \cdot P \mid !P \\
1934 \\
1935 \quad \frac{}{a \cdot P \xrightarrow{\text{ccs}}^a P} \quad \frac{P \xrightarrow{\text{ccs}}^l P'}{P \oplus Q \xrightarrow{\text{ccs}}^l P'} \quad \frac{Q \xrightarrow{\text{ccs}}^l Q'}{P \oplus Q \xrightarrow{\text{ccs}}^l Q'} \quad \frac{P \xrightarrow{\text{ccs}}^l P'}{P \parallel Q \xrightarrow{\text{ccs}}^l P' \parallel Q} \\
1936 \\
1937 \\
1938 \\
1939 \quad \frac{Q \xrightarrow{\text{ccs}}^l Q'}{P \parallel Q \xrightarrow{\text{ccs}}^l P \parallel Q'} \quad \frac{P \xrightarrow{\text{ccs}}^c P' \quad Q \xrightarrow{\text{ccs}}^{\bar{c}} Q'}{P \parallel Q \xrightarrow{\text{ccs}}^\tau P' \parallel Q'} \quad \frac{P \xrightarrow{\text{ccs}}^l P' \quad l \notin \{c, \bar{c}\}}{\nu c \cdot P \xrightarrow{\text{ccs}}^l \nu c \cdot P'} \\
1940 \\
1941 \\
1942 \\
1943 \quad \frac{P \parallel !P \xrightarrow{\text{ccs}}^l P'}{!P \xrightarrow{\text{ccs}}^l P'} \\
1944 \\
1945
\end{array}$$

Fig. 31: Syntax for ccs (🍌) and its operational semantics (🍌)

label c or \bar{c} , representing the sending/reception of a message on a channel, or the reserved action τ , which represents an internal action.

The standard operational semantics, shown in the figure, is expressed as a labeled transition system, where states are terms P and labels are actions l . The ccs operators are the following: 0 is the process with no behavior. A prefix process $l \cdot P$ emits an action l and then becomes the process P . The internal choice $P \oplus Q$ behaves either like the process P or like the process Q , in the same fashion as the `BRINTERNAL` semantics for `br` in Section 2.2. The parallel composition of two processes $P \parallel Q$ interleaves the behavior of the two processes, while allowing the two processes to communicate. If the process P emits a name c and the process Q emits its co-name \bar{c} , then the two processes can progress simultaneously and the parallel composition emits an internal action τ . Channel restriction $\nu c \cdot P$ prevents the process P from emitting an action c or \bar{c} : the operational rule states that any emission of another action is allowed. Finally the replicated process $!P$ behaves as an unbounded replication of the process P . Operationally, $!P$ has the behavior of $P \parallel !P$.

7.2 Model

We define a denotational model for ccs using `ctree actE ccsB void` as domain, written `ccs#` in the following. As witnessed by this type, processes do not return any value, but may emit actions modeled as external events expecting `unit` for answer: **Inductive** `actE ::= | act a : actE unit`. They can exhibit only binary, ternary, or quaternary branch, captured in `ccsB`. Figure 32 defines the semantic operators associated with each construct of the language. They are written as over-lined versions of their syntactic counterparts, and defined over `ccs#`.

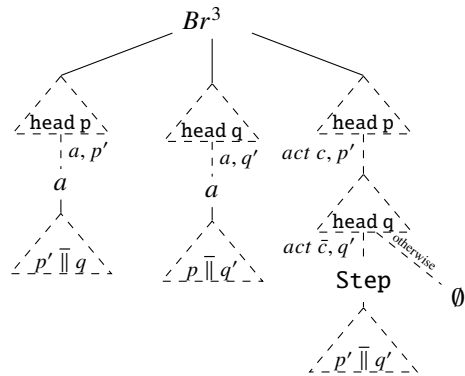
The empty process is modeled as a stuck tree—we cannot observe it. Actions are directly defined as visible events, and thus the prefix triggers the action, and continues with the remaining of the process. As discussed in Section 2.2, the delayed branching node fits exactly with the semantics of the choice operator in ccs, only progressing if one of the composed terms progresses. Restriction raises a minor issue: the compositional definition

$$\begin{aligned}
1979 \quad & \bar{0} \triangleq \emptyset & a\bar{c}p \triangleq \text{trigger } a ;; p & p \bar{\oplus} q \triangleq Br^2 p q & \bar{!}p \triangleq p \bar{\parallel} !p \\
1980 & & & & \\
1981 \quad & & & & \text{vc}P \triangleq \text{interp h_new } c P \\
1982 & & & & \text{where h_new } c e = \begin{cases} \emptyset & \text{if } e = \text{act } c \text{ or } e = \text{act } \bar{c} \\ \text{trigger } e & \text{otherwise} \end{cases} \\
1983 & & & & \\
1984 & & & & \\
1985 \quad & p \bar{\parallel} q \triangleq \text{cofix } F p q \cdot Br^3 & (p' \leftarrow \text{head } p ;; \text{actL } F q p') & & \\
1986 & & (q' \leftarrow \text{head } q ;; \text{actR } F p q') & & \\
1987 & & (p' \leftarrow \text{head } p ;; q' \leftarrow \text{head } q ;; \text{actLR } F p' q') & & \\
1988 & & & & \\
1989 & & & & \text{actL } F q (AStep t) \triangleq Step (F t q) \\
1990 & & & & \text{actL } F q (AVis e k) \triangleq Vis e (\lambda i \cdot F (k i) q) \\
1991 & & & & \\
1992 & & & & \text{actR } F p (AStep t) \triangleq Step (F p t) \\
1993 & & & & \text{actR } F p (AVis e k) \triangleq Vis e (\lambda i \cdot F p (k i)) \\
1994 & & & & \\
1995 & & & & \text{actLR } F r r' \triangleq \\
1996 & & & & \begin{cases} Step \cdot F (k ()) (k' ()) & \text{if } \exists a. r = Vis (act a) k \wedge r' = Vis (act \bar{a}) k' \\ \emptyset & \text{otherwise} \end{cases} \\
1997 & & & & \\
1998 & & & & \\
1999 & & & & \\
2000 & & & & \\
2001 & & & & \\
2002 & & & & \\
2003 & & & & \\
2004 & & & & \\
2005 & & & & \\
2006 & & & & \\
2007 & & & & \\
2008 & & & & \\
2009 & & & & \\
2010 & & & & \\
2011 & & & & \\
2012 & & & & \\
2013 & & & & \\
2014 & & & & \\
2015 & & & & \\
2016 & & & & \\
2017 & & & & \\
2018 & & & & \\
2019 & & & & \\
2020 & & & & \\
2021 & & & & \\
2022 & & & & \\
2023 & & & & \\
2024 & & & &
\end{aligned}$$

Fig. 32: Denotational model for ccs using $\text{ccs}^\#$ as a domain (🔴)

implies that the CTree for the restricted term has already been produced when we encounter the restriction and, a priori, that tree might contain visible actions on the name being restricted. We enforce scoping by replacing those actions by a stuck tree, \emptyset , effectively cutting these branches. This is done using the `interp` operator from CTrees, with `h_new`, a handler that does the substitution.

Parallel composition is more intricate, as the operator requires significant introspection of the composed terms. The traditional operational semantics of ccs is not explicitly constructive: each of the three reduction rules depends on the existence of specific transitions in the sub-processes. We perform this necessary introspection, in a constructive way, defining the tree as an explicit cofixpoint, and using the `head` operator introduced in Section 3.2. While the operation `head p` precisely captures the desired set of actions that p may perform, computing this set could, in general, silently diverge. We therefore cannot bluntly

Fig. 33: Depiction of the tree resulting from $p \bar{\parallel} q$

$$\bar{!}p \triangleq p \parallel \bar{!}p$$

$$p \parallel \bar{!}q \triangleq \text{cofix } F \ p \ q \cdot Br^4 \quad \begin{array}{l} (p' \leftarrow \text{head } p \ ; \ ; \ \text{actL } F \ q \ p') \\ (q' \leftarrow \text{head } q \ ; \ ; \ \text{pbR } F \ q \ p \ q') \\ (p' \leftarrow \text{head } p \ ; \ ; \ q' \leftarrow \text{head } q \ ; \ ; \ \text{pbLR } F \ q \ p' \ q') \\ (q' \leftarrow \text{head } q \ ; \ ; \ q'' \leftarrow \text{head } q \ ; \ ; \ \text{pbRR } F \ q \ q' \ q'') \end{array}$$

$$\begin{array}{l} \text{actL } F \ q \ (AStep \ t) \triangleq Step(F \ t \ q) \\ \text{actL } F \ q \ (Vis \ e \ k) \triangleq Vis \ e \ (\lambda i \cdot F \ (k \ i) \ q) \end{array}$$

$$\begin{array}{l} \text{pbR } F \ q \ p \ (AStep \ t) \triangleq Step(\lambda i \cdot F \ (p \ \bar{\parallel} \ t) \ q) \\ \text{pbR } F \ q \ p \ (Vis \ e \ k) \triangleq Vis \ e \ (\lambda i \cdot F \ (p \ \bar{\parallel} \ k \ i) \ q) \end{array}$$

$$\text{pbLR } F \ q \ r \ r' \triangleq$$

$$\begin{cases} Step \cdot F \ (k \ () \ \bar{\parallel} \ k' \ ()) \ q & \text{if } \exists a. r = Vis \ (act \ a) \ k \wedge r' = Vis \ (act \ \bar{a}) \ k' \\ \emptyset & \text{otherwise} \end{cases}$$

$$\text{pbRR } F \ q \ r \ r' \triangleq$$

$$\begin{cases} Step \cdot F \ (p \ \bar{\parallel} \ k \ () \ \bar{\parallel} \ k' \ ()) \ q & \text{if } \exists a. r = Vis \ (act \ a) \ k \wedge r' = Vis \ (act \ \bar{a}) \ k' \\ \emptyset & \text{otherwise} \end{cases}$$

Fig. 34: Denotational model for $\bar{!}p$ (👉)

initiate the computation by sequencing the heads of p and q , as divergence in the former may render inaccessible valid transitions in the latter.²¹ Instead, we initiate the computation with a ternary delayed choice: the left (resp. middle) branch captures the behaviors starting with an interaction by p (resp. q), while the right branch captures the behaviors starting with a synchronisation between p and q . Essentially, the tree nondeterministically explores the set of applicable instances of the three operational rules for parallel composition. In particular, if the operational rule to step in the left (resp. right) process is non-applicable, the left (resp. middle) branch of the resulting tree silently diverges. The right branch silently diverges if neither process can step, but, in general, it also contains branches considering the interaction of incompatible actions; we cut these branches by inserting \emptyset . In all cases, the operator continues corecursively, having progressed in either or both processes. Figure 33 shows the CTree resulting from $p \ \bar{\parallel} \ q$.

The last operator to consider is the replication $\bar{!}$. In theory, it could be expressed in terms of parallel composition directly, as the cofix $\bar{!}p \triangleq p \ \bar{\parallel} \ \bar{!}p$. Unfortunately, although it is sound, defining the $\bar{!}$ operator in this way is too involved for Coq's syntactic criterion on cofixes to recognize that the corecursive call is guarded under $\bar{\parallel}$. To circumvent this

²¹ Technically, the variant of CCS considered here actually cannot generate such a computation, so we could therefore rule this case out extensionally. The case could, however, easily arise in a variant of CCS relying on recursive processes rather than replication, and in other calculi, so we therefore favor this more general, reusable, approach.

difficulty, we use an auxiliary operator, defined on Figure 34. The intuition behind this operator, $p \overline{\parallel} !q$, is to capture the parallel composition of a process p with a replicated process $!q$. By extending the domain of the function, we manage to recover a syntactically guarded cofix, therefore accepted by Coq. The operator $p \overline{\parallel} !q$ nondeterministically explores the four kinds of interactions that such a process could exhibit: a step in p ; the creation of a copy of q performing a step to q' , before being composed in parallel with p ; the creation of a copy of q synchronizing with p ; or the emission of two copies of q synchronizing one with another. We finally define the replication operator as $\bar{!}p \triangleq p \overline{\parallel} !p$.

With these tools at hand, the model $\llbracket \cdot \rrbracket : \text{ccs} \rightarrow \text{ccs}^\#$ is defined by recursion on the syntax.

Equational Theory We provide a first validation of our model by proving that it satisfies the expected equational theory with respect to CTrees's notion of strong bisimulation, enabling the usual algebraic reasoning advocated for process calculi. In particular, we prove that our definition for the replication is sane in that it validates equationally the expected definition: $\bar{!}p \sim \bar{!}p \overline{\parallel} p$ (🔥). We also prove an illustrative collection of expected equations satisfied by our operators (🔥):

$$\begin{array}{ccccccc} p \overline{\oplus} q \sim q \overline{\oplus} p & p \overline{\oplus} (q \overline{\oplus} r) \sim (p \overline{\oplus} q) \overline{\oplus} r & p \overline{\oplus} \bar{0} \sim p & p \overline{\oplus} p \sim p \\ p \overline{\parallel} \bar{0} \sim p & p \overline{\parallel} q \sim q \overline{\parallel} p & p \overline{\parallel} (q \overline{\parallel} r) \sim (p \overline{\parallel} q) \overline{\parallel} r \end{array}$$

To facilitate these proofs, we first prove sound up-to principles at the level of ccs for each constructor: strong bisimulation up-to $c^\tau[\cdot]$, $[\cdot] \overline{\oplus} [\cdot]$, $[\cdot] \overline{\parallel} [\cdot]$, $\bar{!}[\cdot]$, and $\nu c^\tau[\cdot]$ are all valid principles, allowing us to rewrite sbisim under semantic contexts during bisimulation proofs. Additionally to these language-level up-to principles, we inherit the ones generically supported by sbisim (Lem. 3).

7.3 Equivalence with the operational strong bisimilarity

In addition to proving that we recover in our semantic domain the expected up-to principles and the right algebra, we furthermore show that the model is sound and complete with respect to strong bisimulation compared to its operational counterpart. We do so by first establishing an asymmetrical bisimulation between ccs and $\text{ccs}^\#$, matching operational steps over the syntax to semantic steps in the CTree. We write \bar{l} for the obvious translation of labels between both LTSs.

Definition 6 (Strong bisimulation between ccs and $\text{ccs}^\#$). *A relation $\mathcal{R} : \text{rel}(\text{ccs}, \text{ccs}^\#)$ is a strong bisimulation if and only if, for any label l , ccs term P , and $\text{ccs}^\#$ tree q .*

$$\begin{array}{ccc} P \mathcal{R} q \wedge P \xrightarrow{\text{ccs}} P' \implies \exists q', P' \mathcal{R} q' \wedge q \xrightarrow{\bar{l}} q' & \begin{array}{ccc} P & \mathcal{R} & q \\ l \downarrow_{\text{ccs}} & & \downarrow \bar{l} \\ P' & \mathcal{R} & q' \end{array} \\ \text{and conversely} & & \\ P \mathcal{R} q \wedge q \xrightarrow{\bar{l}} q' \implies \exists P'. P' \mathcal{R} q' \wedge P \xrightarrow{l} P' & & \end{array}$$

Lemma 12 (🔥). *The relation $R \triangleq \{(P, q) \mid \llbracket P \rrbracket \sim q\}$ is a strong bisimulation.*

We derive from this result that the operational and semantic strong bisimulations define exactly the same relation over CCS:

Lemma 13 (🔥). $\forall P Q, \llbracket P \rrbracket \sim \llbracket Q \rrbracket$ iff $P \sim_{\text{CCS}} Q$

8 Case study: modelling cooperative multithreading

As a second case study, we consider cooperative multithreading. Cooperative scheduling is used in languages such as Javascript, async Rust, or Akka/Scala actors, but is also a very general model, as preemptive multi-tasking is equivalent to cooperative scheduling where threads are willing to yield (i.e., let other threads run) at any time. We extend the syntax of `imp` with two additional constructs:

$$\text{comm} \triangleq \text{skip} \mid x ::= e \mid c1; c2 \mid \text{while } b \text{ do } c \mid \text{fork } c1 \ c2 \mid \text{yield}$$

The command `fork` forks the current thread into two, running respectively `c1` and `c2`. Importantly, during such a fork, the thread running `c2` retains control, and the one running `c1` will only get a chance to execute after `c2` voluntarily yields control, or terminates. This yielding of control is achieved by the `yield` statement, which signals that the current thread suspends its execution, and lets a new thread be scheduled—possibly the same one again.

This semantics implements a mechanism akin to the `fork` system call, which duplicates the current process, but without a possibility for joining threads. For example, the program

$$(\text{fork } (x ::= 1) (\text{yield}; x ::= 2)); y ::= x$$

forks two copies of the program, with the “main” thread immediately yielding, allowing for either thread to run next. Its semantics is to first spawn a thread for `x ::= 1`, then have the main thread reach the `yield`, giving `x ::= 1` a chance to run. Assuming the spawned thread goes next, it runs in sequence `x ::= 1` and `y ::= x`, after which the main thread recovers control and finishes its execution. `y ::= x` is part of both threads and is thus executed twice, after each assignment to `x`.

Alternatively, some cooperative scheduling languages (Abadi and Plotkin, 2010) consider a `spawn` operator that simply spawns an independent thread: we can encode this behavior in several ways. Notably, if `spawn` always occurs in tail position, there is no continuation to duplicate. For instance, the program

$$\text{fork } x ::= 1 (\text{fork } (x ::= 2) \text{ skip})$$

spawns two threads that set `x` to different values, and terminates. The two spawned threads can then be scheduled in either order, resulting in `x = 1` or `x = 2` in the final state. This constraint could be syntactically enforced in the language if relevant. Alternatively, one can use the command `while true do yield`²² to “terminate” a thread; for instance to prevent the first thread above from reaching `y ::= x`. With fancier encodings using reserved shared-variables, nested “joins” and other synchronization operations can be modeled. In

²² Or more elegantly, introduce `block` in the language.

this case study, we do not concern ourselves with such extensions, and restrict ourselves to the formalization of the syntax described above.

8.1 Model

The model for `ImpBr`, described in Section 2, was defined in two stages: a representation into a nondeterministic `Ctree` with interaction with memory, and then a stateful interpretation. We proceed this time in three stages: a representation into a deterministic `Ctree` with concurrent interaction represented as external events, a scheduling pass introducing nondeterminism by interleaving all the valid executions, and finally a stateful interpretation.

Representation First, we represent statements as computations of type `ctree (YieldE + SpawnE + MemE) voidB unit`. At this stage of representation, they are modeled as *deterministic* computations,²³ as highlighted by the use of the empty interface `voidB` for branches. The computation can however perform two more classes of events than the traditional memory ones: yielding and spawning:

```
Variant YieldE : Type → Type :=      Variant ForkE : Type → Type :=
  | Yield : YieldE unit.              | Fork : ForkE bool.
```

`Yield` carries no additional information and acts purely as a signal to yield control, and `Fork` introduces a binary branch in the `Ctree`, allowing us to store the asynchronous thread in one branch and the main thread that continues running in the other branch. These events are used to represent the corresponding statements:

$$\llbracket \text{yield} \rrbracket \triangleq \text{trigger (Yield)}$$

$$\llbracket \text{fork } c1 \ c2 \rrbracket \triangleq b \leftarrow \text{trigger (Fork)} \ ; \ ; \text{ if } b \text{ then } \llbracket c1 \rrbracket \text{ else } \llbracket c2 \rrbracket$$

The remaining of the representation is entirely standard.

We write $\llbracket p \rrbracket$ the representation of p .

Interleaving The model's second pass introduces the non-determinism implicitly induced by the `Fork` events (extending the thread pool) and the `Yield` events (non-deterministically picking a new active thread to schedule). At a high-level, our goal is hence to write a function:

```
schedule1: ctree (YieldE + ForkE + MemE) voidB unit → ctree MemE Bn unit
```

where `Bn` allows arbitrary finite branching—at run time, we may pick an identifier out of the current pool set, i.e., out of an unbounded finite set.

This combinator, like in the case of parallel composition for `ccs`, cannot be simply defined via `interp`. We hence craft this function by co-recursion, but need to generalize it first to this end. Let us pose a couple of definitions:

```
Variant SpawnE : Type → Type := | Spawn : SpawnE unit.
Notation thread := ctree (YieldE + ForkE + MemE) voidB unit.
Notation prog := ctree (YieldE + SpawnE + MemE) Bn unit.
```

²³ We could have alternatively used `ITrees` as semantic domain at this stage.


```

2209  schedule v0 [-] ≐ ret ()
2210  schedule vn [-] ≐ Vis Yield (branchn (λi · schedule vn [i]))    for n > 0
2211  schedule vn [i] ≐
2212    if v[i] =          then
2213      ret ()          Guard (schedule v[-i]n-1 [-])
2214      Guard t        Guard (schedule v[i ↦ t]n [i])
2215      Step t          Step (schedule v[i ↦ t]n [i])
2216      Stuck           Stuck
2217      Vis Yield k     Guard (schedule v[i ↦ k] ())n [-]
2218      Vis Fork k      Vis Spawn (λ_ · schedule (k true :: v[i ↦ k false])n+1 [i + 1])
2219      Vis e k         Vis e (λx · schedule (v[i ↦ k x])n [i])

```

Fig. 35: The definition of schedule (👉)

We introduce an external event `Spawn` containing no information that we will use to keep track of points where a fork happened. We write `thread` as a shorthand for the datatype of represented threads, i.e., intermediate, deterministic, models of pieces of code that have not been scheduled yet. In contrast, we write `prog` for the second semantic domain we aim, the datatype of already scheduled (and therefore nondeterministic) computations.

Our updated goal is therefore to craft a cofixpoint

```
schedule n (pool: fin n → thread) (curr : option (fin n)): prog,
```

where `n` is the arity of the current set of threads left to interpret, and `curr` is the index of the thread currently under focus, if any. The result is a scheduled computation, i.e., a `prog`. Once we have this function, we shall define our second stage of interpretation by simply starting with the singleton thread pool:

Definition `schedule1 t := schedule 1 (λ _ ⇒ t)` (Some F1)

Figure 35 defines the function formally.²⁴ We write v_n for a vector of size n , and vector operations for removing the i -th element as $v[-i]$, updating the i -th element to x as $v[i \mapsto x]$, and adding an element x to the front as $x :: v$ (so x is the new 0-th element in the resulting vector). The traditional constructors of the option type `None` and `Some v` are written respectively $[-]$ and $[v]$. References to `schedule` in its body should be interpreted as corecursive calls—we abuse notations to lighten the presentation.

The first two cases cover the situation where no thread is active, i.e., the second argument is $[-]$. If the thread pool is empty, the computation simply terminates. Otherwise, it picks a thread to be scheduled: a `Yield` event is inserted, followed by a branching node of arity the cardinality of the thread pool, and sets the chosen thread active.

If there is an active thread, $[i]$, the schedule makes progress in that thread, analyzing the corresponding tree in the pool. If the active thread has terminated, it is removed from the pool, and out of focus. `Guard`, `Step`, and memory event nodes are simply kept, the active thread updated, and scheduling continues without changing focus. A stuck thread blocks the whole computation: this justifies encoding variant fork operators that do not distribute

²⁴ The implementation relies on Sozeau and Mangin (2019)'s Equations library given the heavy reliance on dependent pattern matching on vectors.

over sequence by killing one of the forked threads as suggested in the introduction of this Section—it has however the inelegant side effect that the dead threads are never garbage collected from the pool. Remain finally to treat the concurrency events.²⁵ The `Yield` nodes are substituted for an invisible `Guard`, and remove the current focus—the event is hence reintroduced right after in the focusing rule. The `Fork` nodes are replaced by a simple unary `Spawn` marker, and corecursion occurs over the vector extended with the new thread, and the updated thread that remains under focus (note that this shifts the active thread from index i to $i + 1$).

The acute reader may wonder why we keep `Yield` and `Spawn` events in the `prog` datatype since they do not contain any data. And indeed, we follow this scheduling process by an interpretation phase simply removing these events. They however offer an intermediate semantic domain at which less programs are equated, but that respects more contexts.

We write $\mathcal{S}[[p]] \triangleq \text{schedule } [[p]]_1 [-] : \text{ctree MemE unit}$ and $\bar{\mathcal{S}}[[p]] : \text{ctree MemE Bn unit}$ the resulting tree after clean up of the `Yield` and `Spawn` events.

Stateful interpretation. Remains only to interpret the memory events. No difficulty remains, as already informally described over `ImpBr` and formalized in Section 2.1, we can simply use the generic facility for stateful interpretation over `CTrees`. The resulting, final, semantic domain is therefore `stateT mem (ctree voidE Bn) unit`.

8.2 Equational theory

The model described in Section 8.1 allows us to derive some program equivalences at source-level w.r.t. weak bisimilarity of their models. For example, the following programs all just run c , though some of them first perform some “invisible” steps related to concurrency (👉):

$$\bar{\mathcal{S}}[[\text{fork } c \text{ skip}]] \sim \bar{\mathcal{S}}[[\text{yield}; c]] \sim \bar{\mathcal{S}}[[c]]$$

We emphasize that these equations are not compositional, they only hold in the absence of additional concurrent threads, hence why `yield` behaves as a no op. In contrast, we conjecture that the monadic equivalence between $\mathcal{S}[[\cdot]]$ representations is a congruence, albeit we have not proved it formally at the moment.

Other equations, especially ones that make use of multiple threads in nontrivial ways, rely on the stability of `schedule` under `sbsim`:

Lemma 14 (schedule preserves \sim (👉)). *If the delayed branches of every element of vectors v_n and w_n have arity less than 2, and the elements of both vectors are strongly bisimilar up to a permutation ρ , then `schedule` $v_n [i] \sim$ `schedule` $w_n [\rho i]$.*

The arity requirement is satisfied by all denotations of programs in this language. This condition greatly simplifies the proof by constraining the shape that the strongly bisimilar `CTrees` can take.

²⁵ Note that the typing of `thread` rules out statically the branching case, which simplifies greatly the proof of the meta-theory of `schedule`.

Lemma 14 allows us to permute the thread pool, which is useful in examples such as (🔥):

$$S[\text{fork } c1 \text{ (fork } c2 \text{ skip)}] \approx S[\text{fork } c2 \text{ (fork } c1 \text{ skip)}]$$

This program spawns two asynchronous threads then yields control to one of the two. This equivalence captures the natural fact that it does not matter which thread is spawned first, since neither can run until both are spawned.

Furthermore, Lemma 14 allows us to validate some simple optimizations that do not directly involve reasoning about concurrency or memory, such as (🔥):

$$\begin{aligned} & S[\text{fork } c1 \text{ (fork (while true do yield) skip)}] \\ \approx & S[\text{fork (yield; while true do yield)(fork } c1 \text{ skip)}] \end{aligned}$$

Here, one of the spawned threads is a while loop, which we wish to unroll by one iteration. Crucially, the loop and its unrolled form are strongly bisimilar, so this equivalence follows from Lemma 14 just as in the previous example. Other optimizations that can be done before interpreting events, such as constant folding or dead code elimination, can be proven sound similarly.

Finally, equivalences involving memory operations are still valid as well (🔥):

$$\text{fork } (x::= 2) \ (x::= 1) \equiv x::= 2$$

where \equiv here refers to equivalence (\approx in this case) after interpreting both concurrency and memory events. This result follows from the result in Section 6.1, which allows us to transport equations made before interpreting state events into computations in the state monad after interpretation.

9 Related Work

Since Milner's seminal work on ccs (Milner, 1989) and the π -calculus (Milner et al., 1992), process algebras have been the topic of a vast literature (Bergstra et al., 2001). We mention only a few parts of it that are most relevant to our work. In the Coq realm, Lenglet and Schmitt (2018) have formalized $\text{HO}\pi$, a minimal π -calculus, notably exhibiting the difficulty inherent to the formal treatment of name extrusion. Beyond its formalization, dealing with scope extrusion as part of a compositional semantics is known to be a challenging problem (Crafa et al., 2012; Cristescu et al., 2013). By restricting to ccs in our case study, we have side-stepped this difficulty. Foster et al. (2021) formalize in Isabelle/HOL a semantics for CSP and the Circus language using a variant implementation of ITrees, where continuations to external events are partial functions. However, they only model deterministic processes, leaving nondeterministic ones for future work. This paper introduces the tools to address that problem. CSP has also been extensively studied by Brookes (2002) by providing a model based on the compositional construction of infinite sets of traces: CTrees offer a complementary coinductive model to this more set-theoretic approach. Brookes tackles questions of fairness, an avenue that we have not yet explored in our setup.

Formal semantics for nondeterminism are especially relevant when dealing with low-level concurrent semantics. In shared-memory-based programming languages, rather than

message passing ones, concurrency gives rise to the additional challenge of modeling their memory models, a topic that has received considerable attention. Understanding whether monadic approaches such as the one proposed in this paper are viable to tackle such models vastly remains to be investigated. Early suggestions that they may include Lesani et al. (2022): the authors prove correct concurrent objects implemented using ITrees, assuming a sequentially consistent model of shared memory. They relate the ITrees semantics to a trace-based one to reason about refinement, something that we conjecture would not be necessary when starting from CTrees. Operationally specified memory models, in the style of which increasingly relaxed models have been captured and sometimes formalized, intuitively seem to be a better fit. Major landmarks in this axis include the work by Sevcík et al. on modeling TSO using a central synchronizing transition system linking the program semantics to the memory model in the CompCertTSO compiler (Sevcík et al., 2013); or Kang et al.’s promising semantics (Kang et al., 2017; Lee et al., 2020) that have captured large subsets of the C++11 concurrency model without introducing out-of-thin-air behaviors. On the other side of the spectrum, axiomatic models in the style of Alglave et al.’s (Alglave et al., 2014, 2021) framework appear less likely to transpose to our constructive setup.

Our model for cooperative multithreading is partially reminiscent of Abadi and Plotkin’s work (Abadi and Plotkin, 2010): they define a denotational semantics based on partial traces that they prove fully abstract, and satisfying an algebra of stateful concurrency. The main difference between the two approaches is that partial traces use the memory state explicitly to define the composition of traces, where CTrees can express the semantics of a similar language independently of the memory model. The formal model we describe here tackles a slightly different language than theirs, but we should be able to adapt it reasonably easily to obtain a formalization of their work. More recently, Din et al. (Din et al., 2017, 2022) have suggested a novel way to define semantics based on the composition of symbolic traces, partially inspired by symbolic execution (King, 1976). They use it, in particular, to formalize actor languages, which rely on cooperative scheduling, with a similar modularity as the one we achieve (orthogonal semantic features can be composed), but not in a compositional way.

Our work brings proper support for nondeterminism to monadic interpreters in Coq. As with ITrees however, the tools we provide are just right to conveniently build denotational models of first order languages, such as CCS, but have difficulty retaining compositionality when dealing with higher-order languages. In contrast, on paper, game semantics has brought a variety of techniques lifting this limitation. In particular in a concurrent setup, event structures have spawned a successful line of work (Rideau and Winskel, 2011; Castellan et al., 2017) from which inspiration could be drawn for further work on CTrees.

Comparison with the original CTrees paper and subsequent related work

Differences in design compared to Chappe et al. (2023). The original CTrees paper (Chappe et al., 2023) is the basis for the present one. We detail below the main differences and limitations of this previous iteration.

At the time, the structure was not parameterized by a signature B . Rather, Br nodes were limited to finite branching in $\text{fin } n$. This limitation was triple. First most obviously, it prevented infinite branching (which is heavily used in Chappe et al. (2024)). Second,

it did not allow carrying information about the origin of a given internal choice: when encountering a branching on `fin n`, there was no way to know whether it represents the generation of a random number or the choice of a thread to schedule, for instance. The new parameterization allows for the theory of `refine` we develop in Section 6.2, and more generally enable the possibility of writing interesting schedulers based on source-level information. Finally, the case study presented in Section 8 had to rely on an extrinsic coinductive invariant to express and exploit the fact that the first level of representation is deterministic. We can now capture it statically with an empty branching interface.

On the other hand, `∅` and `Guard` used to be particular cases of `Br`, respectively nullary and unary branching nodes. However, with this parameterization, maintaining this encoding required class constraints to every definition relying on them. This overhead led us to expose a sort of canonical encoding of these two constructs as dedicated constructors.

Finally, `BrS` nodes used to be their own construction, and it was proved that they could be equivalently encoded as guarded `Br` nodes. Since we also add to introduce a dedicated `Step` node for the same reason as `∅` and `Guard`, we removed the `BrS` constructors and directly work with the encoding.

Without the alternative definition of the LTS from Section 5, some results, especially from Section 6, were significantly harder to establish and were proved in more restrictive cases. In particular, the monad morphism result for `interp` was only proved for handlers reduced to `trigger` or `Ret`. Similarly, the simulation result for `refine` was only proved for constant handlers. The introduction of the alternative LTS enables more concise and more general proofs for these results.

The equational theory was based on an older version of Pous' coinduction library that relied on the companion (Pous, 2016) instead of tower induction. These two theories are proved equivalent in the library, but tower induction is more comfortable to work with, especially when up-to principles are involved.

Comparison to Bahr and Hutton (2023). Following the original publication of Chappe et al. (2023), Bahr and Hutton (2023) have proposed an implementation in Agda of a variant on the `CTree` structure, and extensively compared it to the original paper on `CTrees`. The most important difference they introduce is to statically prevent infinite chain of `Br` nodes, i.e., in particular infinite structures denoting stuck processes, by defining the datatype through mutual induction-coinduction: intuitively, a `Step` guard must always be finitely reachable. Their definition as is would not be accepted by `Coq`, but investigating alternate encoding would be an interesting perspective. Another distinction came from one of the observations in the original paper: to avoid the necessary restriction on handlers described in Section 6, visible events generate two successive transitions in the LTS: one deterministically labeled with the event (a question to the environment), and a second one labeled by the response from the environment. This split makes the definition of the LTS more cumbersome, since the domain of states is no longer the datastructure itself, but it does recover a (tighter) equivalence unconditionally preserved by interpretation. Moving our `CTree` library to such a finer LTS would make sense, but we leave it to future work as we have not yet encountered a concrete case where our simpler definition does not suffice.

The paper focuses on concurrency for functional programming, relying notably on a binary parallel operator. In this setting, the definition of their parallel operator relies

on a codensity monad, and their notion of program equivalence relies on step-indexed bisimilarity. These various theories are exploited for calculating concurrent compilers.

Comparison to Cho et al. (2023). Cho et al. (2023) introduces DTrees, a generalization of ITrees with support for nondeterminism: the \mathbb{D} Tau sort of node is the counterpart of our BrS nodes. DTrees are equipped with a novel notion of weak simulation that they call *freely-stuttering*. The key element of this notion of simulation is that its simulation game operates not only on 2 programs, but also on additional indices that enable more powerful reasoning principles, especially for the asymmetric stripping of Taus. In this regard, this approach has some similarities with our alternative characterization of strong bisimulation that relies on an extra boolean in the bisimulation game. Interestingly, Cho et al. (2023) observe that this notion is not only weaker than the notions of forward and backward simulations in CompCert (Leroy, 2009), but that *each step of these notions of simulation* can be replayed by one step of freely-stuttering simulation. Unfortunately, this notion is not strong enough to formally compare the standard and alternative notions of strong similarity from the present paper, as a step of our strong simulation from Section 4.5.1 corresponds to *several* steps of our strong simulation from Section 5.1. The paper additionally studies new ATau nodes, representing angelic nondeterminism, with no equivalent in CTrees. Cho et al. (2023) defined freely-stuttering simulations in a generic way so that they can be applied to LTSs that are not defined as DTrees. Note that due to their lack of equivalent to Br nodes, some LTSs that can be represented as CTrees cannot be represented as DTrees, in particular LTSs with nondeterminism that stems from non- τ nodes.

10 Conclusion and Perspectives

We have introduced CTrees, a model for nondeterministic, recursive, and impure programs in Coq. Inspired by ITrees, we have introduced two kinds of nondeterministic branching nodes, and designed a toolbox to reason about these new computations. Beyond the various strong (bi)similarity game that we studied in depth, future extensions of our work could develop the meta-theory around weak (bi)similarity. More ambitiously, we could refactor our library to clearly separate our various reasoning principles on LTSs from the CTree structure, so that they can be used more widely.

We have illustrated the expressiveness of the framework through two significant case studies. Both nonetheless offer avenues for further work, notably through an extension of ccs to name passing *à la* π -calculus, and to further extend the equational theory for cooperative multithreading that we currently support.

With this extended version, we have presented how the library has evolved during the two years that separates us from its introduction. Not only has the library evolved in the meantime, but it has been put to great use. In particular, Chappe et al. (2024) has shown that it is expressive enough to model complex weak memory models, paving realistically the long term goal to leveraging the library as the basis for a verified compiler with support for weak memory models.

The library has been greatly enriched, from introducing additional equivalences and refinements, generalizing their meta-theory, but also to reimplementing its structure based

on slightly tweaked design points. These design choices matter greatly. It is particularly interesting to see at around the same time alternate proposals for similar structures, in particular in Bahr and Hutton (2023) and Cho et al. (2023). Conducting an in-depth comparison of these approaches could fuel the next iteration towards a formal library for building monadic model of concurrent programming languages.

Data-Availability Statement

Acknowledgements

We are grateful to the anonymous reviewers from POPL'23 for their in-depth comments that helped both improve this paper, and open avenues of further work. We thank Gabriel Radanne for providing assistance with TikZ. Finally, we are most thankful to Damien Pous for developing the coinduction library this formal development crucially relies on, and for the numerous advice he provided us with.

References

- Abadi, M. & Plotkin, G. D. (2010) A model of cooperative threads. *Logical Methods in Computer Science*. **6**(4), 1–39.
- Abramsky, S. & Melliès, P.-A. (1999) Concurrent games and full completeness. Proceedings of the 14th Annual ACM/IEEE Symposium on Logic in Computer Science. IEEE Computer Society Press.
- Alglave, J., Deacon, W., Grisenthwaite, R., Hacquard, A. & Maranget, L. (2021) Armed cats: Formal concurrency modelling at arm. *ACM Trans. Program. Lang. Syst.* **43**(2), 8:1–8:54.
- Alglave, J., Maranget, L. & Tautschnig, M. (2014) Herding cats: modelling, simulation, testing, and data-mining for weak memory. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014. ACM. p. 40.
- Altenkirch, T., Danielsson, N. A. & Kraus, N. (2017) Partiality, revisited. *Foundations of Software Science and Computation Structures*. Berlin, Heidelberg. Springer Berlin Heidelberg. pp. 534–549.
- Bahr, P. & Hutton, G. (2023) Calculating compilers for concurrency. *Proc. ACM Program. Lang.* **7**(ICFP).
- Beck, C., Yoon, I., Chen, H., Zakowski, Y. & Zdancewic, S. (2024) A two-phase infinite/finite low-level memory model.
- Bergstra, J., Ponse, A. & Smolka, S., editors. (2001) *Handbook of Process Algebra*. Elsevier Science.
- Bloom, B., Istrail, S. & Meyer, A. R. (1988) Bisimulation can't be traced. Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York, NY, USA. Association for Computing Machinery. p. 229–239.
- Brandauer, S., Castegren, E., Clarke, D., Fernandez-Reyes, K., Johnsen, E. B., Pun, K. I., Tarifa, S. L. T., Wrigstad, T. & Yang, A. M. (2015) Parallel objects for multicores: A glimpse at the parallel language encore. *Formal Methods for Multicore Programming - 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Bertinoro, Italy, June 15-19, 2015, Advanced Lectures*. Springer. pp. 1–56.
- Brookes, S. D. (2002) Traces, pomsets, fairness and full abstraction for communicating processes. *Proc. 13th Intl. Conf. on Concurrency Theory (CONCUR 2002)*. Berlin Heidelberg. Springer. pp. 466–482.
- Capretta, V. (2005) General recursion via coinductive types. *Logical Methods in Computer Science*. **1**(2), 1–18.
- Castellan, S., Clairambault, P., Rideau, S. & Winskel, G. (2017) Games and strategies as event structures. *Log. Methods Comput. Sci.* **13**(3).

- 2531 Chappe, N., He, P., Henrio, L., Zakowski, Y. & Zdancewic, S. (2023) Choice trees: Representing
 2532 nondeterministic, recursive, and impure programs in coq. *Proc. ACM Program. Lang.* **7**(POPL).
- 2533 Chappe, N., Henrio, L. & Zakowski, Y. (2024) A concurrency model based on monadic interpreters:
 2534 executable semantics for a concurrent subset of LLVM IR. working paper or preprint.
- 2535 Cho, M., Song, Y., Lee, D., Gähler, L. & Dreyer, D. (2023) Stuttering for free. *Proc. ACM Program.*
 2536 *Lang.* **7**(OOPSLA2).
- 2537 Crafa, S., Varacca, D. & Yoshida, N. (2012) Event structure semantics of parallel extrusion in the
 2538 pi-calculus. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial
 2539 Intelligence and Lecture Notes in Bioinformatics). pp. 225–239.
- 2540 Cristescu, I., Krivine, J. & Varacca, D. (2013) A Compositional Semantics for the Reversible π -
 2541 Calculus. Proceedings - Symposium on Logic in Computer Science. pp. 388–397.
- 2542 Din, C. C., Hähnle, R., Johnsen, E. B., Pun, K. I. & Tapia Tarifa, S. L. (2017) Locally abstract,
 2543 globally concrete semantics of concurrent programming languages. Automated Reasoning with
 2544 Analytic Tableaux and Related Methods. Cham. Springer International Publishing. pp. 22–43.
- 2545 Din, C. C., Hähnle, R., Henrio, L., Johnsen, E. B., Pun, V. K. I. & Tarifa, S. L. T. (2022) Lage
 2546 semantics of concurrent programming languages.
- 2547 Foster, S., Hur, C. & Woodcock, J. (2021) Formally verified simulations of state-rich processes using
 2548 interaction trees in isabelle/hol. 32nd International Conference on Concurrency Theory, CONCUR
 2549 2021, August 24-27, 2021, Virtual Conference. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.
 2550 pp. 20:1–20:18.
- 2551 Harper, R. (2016) *Practical Foundations for Programming Languages*. Cambridge University Press.
 2552 second edition.
- 2553 Henrio, L., Madelaine, E. & Zhang, M. (2016) A theory for the composition of concurrent pro-
 2554 cesses. Formal Techniques for Distributed Objects, Components, and Systems. Cham. Springer
 2555 International Publishing. pp. 175–194.
- 2556 Hur, C.-K., Neis, G., Dreyer, D. & Vafeiadis, V. (2013) The power of parameterization in coinductive
 2557 proof. Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of
 2558 Programming Languages. New York, NY, USA. ACM. pp. 193–206.
- 2559 Kang, J., Hur, C., Lahav, O., Vafeiadis, V. & Dreyer, D. (2017) A promising semantics for relaxed-
 2560 memory concurrency. Proceedings of the 44th ACM SIGPLAN Symposium on Principles of
 2561 Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. ACM. pp. 175–189.
- 2562 King, J. C. (1976) Symbolic execution and program testing. *Communications of the ACM.* **19**(7),
 2563 385–394.
- 2564 Kiselyov, O. & Ishii, H. (2015) Freer monads, more extensible effects. Proceedings of the 8th ACM
 2565 SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015.
 2566 pp. 94–105.
- 2567 Koenig, J. & Shao, Z. (2020) Refinement-based game semantics for certified abstraction layers.
 2568 Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science. New
 2569 York, NY, USA. Association for Computing Machinery. p. 633–647.
- 2570 Lampropoulos, L. & Pierce, B. C. (2018) *QuickChick: Property-Based Testing in Coq*. Software
 2571 Foundations series, volume 4. Electronic textbook.
- 2572 Lee, S., Cho, M., Podkopaev, A., Chakraborty, S., Hur, C., Lahav, O. & Vafeiadis, V. (2020) Promising
 2573 2.0: global optimizations in relaxed memory concurrency. Proceedings of the 41st ACM SIGPLAN
 2574 International Conference on Programming Language Design and Implementation, PLDI 2020,
 2575 London, UK, June 15-20, 2020. ACM. pp. 362–376.
- 2576 Lenglet, S. & Schmitt, A. (2018) Ho π in coq. Proceedings of the 7th ACM SIGPLAN International
 Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9,
 2018. ACM. pp. 252–265.
- Leroy, X. (2009) Formal verification of a realistic compiler. *Communications of the ACM.* **52**(7),
 107–115.
- Lesani, M., Xia, L., Kaseorg, A., Bell, C. J., Chlipala, A., Pierce, B. C. & Zdancewic, S. (2022) C4:
 verified transactional objects. *Proc. ACM Program. Lang.* **6**(OOPSLA), 1–31.
- Letan, T., Régis-Gianas, Y., Chifflier, P. & Hiet, G. (2018) Modular verification of programs with

2577 effects and effect handlers in coq. Formal Methods - 22nd International Symposium, FM 2018, Held
2578 as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings.
pp. 338–354.

2579 Levy, P. B. (2006) Infinitary howe’s method. *Electronic Notes in Theoretical Computer Science*.
2580 **164**(1), 85–104. Proceedings of the Eighth Workshop on Coalgebraic Methods in Computer Science
(CMCS 2006).

2581 Maillard, K., Hrițcu, C., Rivas, E. & Van Muylder, A. (2020) The next 700 relational program logics.
2582 *Proceedings of the ACM on Programming Languages*. **4**(POPL).

2583 Melliès, P.-A. & Mimram, S. (2007) Asynchronous games: Innocence without alternation. CONCUR
2584 2007 – Concurrency Theory. Berlin, Heidelberg. Springer Berlin Heidelberg. pp. 395–411.

2585 Michelland, S., Zakowski, Y. & Gonnord, L. (2024) Abstract Interpreters: a Monadic Approach to
2586 Modular Verification (DRAFT). working paper or preprint.

2587 Milner, R. (1989) *Communication and Concurrency*. Prentice-Hall, Inc. USA.

2588 Milner, R., Parrow, J. & Walker, D. (1992) A calculus of mobile processes, i. *Information and
Computation*. **100**(1), 1–40.

2589 Oliveira Vale, A., Melliès, P.-A., Shao, Z., Koenig, J. & Stefanescu, L. (2022) Layered and object-based
2590 game semantics. *Proc. ACM Program. Lang*. **6**(POPL).

2591 Parrow, J. & Sjödin, P. (1994) The complete axiomatization of cs-congruence. Proceedings of the 11th
2592 Annual Symposium on Theoretical Aspects of Computer Science. Berlin, Heidelberg. Springer-
Verlag. pp. 557–568.

2593 Pous, D. (2007) Complete lattices and up-to techniques. *Programming Languages and Systems*.
2594 Berlin, Heidelberg. Springer Berlin Heidelberg. pp. 351–366.

2595 Pous, D. (2016) Coinduction all the way up. Proceedings of the 31st Annual ACM/IEEE Symposium
2596 on Logic in Computer Science. New York, NY, USA. Association for Computing Machinery. p.
307–316.

2597 Pous, D. (2024) The coq-coinduction library .

2598 Pous, D. (2024) The coq-coinduction library: examples .

2599 Rideau, S. & Winskel, G. (2011) Concurrent strategies. Proceedings of the 26th Annual IEEE
2600 Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario,
2601 Canada. IEEE Computer Society. pp. 409–418.

2602 Sangiorgi, D. (1998) On the bisimulation proof method. *Mathematical Structures in Computer
Science*. **8**(5), 447–479.

2603 Sangiorgi, D. (2012) *Introduction to Bisimulation and Coinduction*. Cambridge University Press.
2604 USA. second edition.

2605 Sangiorgi, D. & Rutten, J. (2012) *Advanced Topics in Bisimulation and Coinduction*. Cambridge
2606 University Press. USA. second edition.

2607 Sangiorgi, D. & Walker, D. (2001) *The π -calculus*. Cambridge University Press. USA. first edition.

2608 Schäfer, S. & Smolka, G. (2017) Tower induction and up-to techniques for CCS with fixed points.
2609 *Relational and Algebraic Methods in Computer Science - 16th International Conference, RAMiCS
2017, Lyon, France, May 15-18, 2017, Proceedings*. pp. 274–289.

2610 Sevcik, J., Vafeiadis, V., Nardelli, F. Z., Jagannathan, S. & Sewell, P. (2013) Compcertso: A verified
2611 compiler for relaxed-memory concurrency. *J. ACM*. **60**(3), 22:1–22:50.

2612 Silver, L., He, P., Cecchetti, E., Hirsch, A. K. & Zdancewic, S. (2023) Semantics for noninterference
2613 with interaction trees. 37th European Conference on Object-Oriented Programming, ECOOP 2023,
2614 July 17-21, 2023, Seattle, Washington, United States. Schloss Dagstuhl - Leibniz-Zentrum für
Informatik. pp. 29:1–29:29.

2615 Smyth, M. (1976) Powerdomains. *Mathematical Foundations of Computer Science*. Springer.

2616 Sozeau, M. & Mangin, C. (2019) Equations reloaded: high-level dependently-typed functional
2617 programming and proving in coq. *Proc. ACM Program. Lang*. **3**(ICFP), 86:1–86:29.

2618 Team, T. C. D. (2022) The coq proof assistant .

2619 van Glabbeek, R. J. (1993) The linear time - branching time spectrum ii. Proceedings of the 4th
2620 International Conference on Concurrency Theory. Berlin, Heidelberg. Springer-Verlag. pp. 66–81.

2621 Xia, L., Zakowski, Y., He, P., Hur, C., Malecha, G., Pierce, B. C. & Zdancewic, S. (2020) Interaction

trees. *Proceedings of the ACM on Programming Languages*. **4**(POPL).

2623 Yoon, I., Zakowski, Y. & Zdancewic, S. (2022) Formal reasoning about layered monadic interpreters.

2624 *Proceedings of the ACM on Programming Languages*. **6**(ICFP).

2625 Zakowski, Y., Beck, C., Yoon, I., Zaichuk, I., Zaliva, V. & Zdancewic, S. (2021) Modular,
2626 compositional, and executable formal semantics for llvm ir. *Proc. ACM Program. Lang.* **5**(ICFP).

2627 Zakowski, Y., He, P., Hur, C.-K. & Zdancewic, S. (2020) An equational theory for weak bisimulation
2628 via generalized parameterized coinduction. Proceedings of the 9th ACM SIGPLAN International
2629 Conference on Certified Programs and Proofs (CPP).

2630

2631

2632

2633

2634

2635

2636

2637

2638

2639

2640

2641

2642

2643

2644

2645

2646

2647

2648

2649

2650

2651

2652

2653

2654

2655

2656

2657

2658

2659

2660

2661

2662

2663

2664

2665

2666

2667

2668