



**HAL**  
open science

# Choice Trees: Representing and Reasoning About Nondeterministic, Recursive, and Impure Programs in Rocq

Nicolas Chappe, Paul He, Ludovic Henrio, Eleftherios Ioannidis, Yannick  
Zakowski, Steve Zdancewic

## ► To cite this version:

Nicolas Chappe, Paul He, Ludovic Henrio, Eleftherios Ioannidis, Yannick Zakowski, et al.. Choice Trees: Representing and Reasoning About Nondeterministic, Recursive, and Impure Programs in Rocq. *Journal of Functional Programming*, 2025. <hal-05154458v2>

**HAL Id: hal-05154458**

**<https://hal.science/hal-05154458v2>**

Submitted on 22 Dec 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons CC BY 4.0 - Attribution - International License

# Choice Trees: Representing and Reasoning About Nondeterministic, Recursive, and Impure Programs in Rocq

NICOLAS CHAPPE

*ENS de Lyon, Univ 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

*CNRS, Univ Lyon, ENS de Lyon, UCBL, 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

*Inria, Univ Lyon, ENS de Lyon, UCBL, CNRS, 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 Rocq. 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 delayed branching. This structure allows us to provide shallow embedding of denotational models with nondeterministic choice in the style of ccs, while recovering an inductive LTS view of the computation. CTrees leverage a vast collection of bisimulation and refinement tools well-studied on LTSs, 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 Rocq’s CIC (The Coq Development Team, 2024). 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. Notably, small step operational semantics are straightforward to mechanize and extend, but offer little compositionality. On the contrary, powerdomains and game semantics for instance are denotational approaches, aiming to ensure compositionality by construction; however, the mathematical structures involved are themselves complex, typically involving set-theoretical relations and constraints (Abramsky and Melliès, 1999; Melliès and Mimram, 2007; Rideau and Winskel, 2011) that are not straightforward to implement in type theory; though there are some notable exceptions (Koenig and Shao, 2020; Oliveira Vale et al., 2022; Borthelle et al., 2025).

In the Rocq ecosystem, *interaction trees* (ITrees) (Xia et al., 2019) strike a sweet spot in this design space: they package in a library *reusable* components for defining *executable denotational* semantics. When applicable, they deliver easy to mechanize models supporting powerful reasoning principles and extraction to definitional interpreters.

This paper introduces an extension to the ITree framework to provide support nondeterminism, while retaining its benefits. The main technical contributions are to introduce the definition of this new structure, the CTrees (“choice trees”), and to develop the suitable metatheory and equational reasoning principles to accommodate that change. To do so, it turns out crucial to represent in the tree a notion of *delayed* branch: a syntactic node that *may* represent a non-deterministic branching in the computation, depending on whether the sub-trees exhibit observable behaviors. Moreover, we demonstrate how giving a meaning to these trees in terms of labelled transition systems (LTS) is the key to put in light the distinction between *stepping* choices (which correspond to  $\tau$  transitions and introduce new LTS states) and *delayed* choices (which don’t correspond to a transition and don’t create a state in the LTS). It furthermore allows us to leverage the process algebra literature and define equivalence (resp. refinement) of CTrees as bisimilarity (resp. similarity).

47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92

93 The net result of our contributions is a library, entirely formalized in Rocq, that offers  
94 flexible building blocks for constructing *executable denotational models* of nondeterministic,  
95 and notably concurrent, computations. To demonstrate the applicability of this library,  
96 we use it to implement the semantics from two formalisms: ccs (Milner, 1989), a process  
97 calculus that we use along the paper to illustrate our definitions, as well as a shared-memory  
98 language with cooperative threads inspired from the literature (Abadi and Plotkin, 2010).  
99 Crucially, in both of these scenarios, we are able to define the necessary parallel composition  
100 operator such that the semantics of the programming language can be defined fully  
101 compositionally (i.e., by straightforward induction on the syntax). Moreover, we recover the  
102 notion of bisimilarity of ccs process based on its operational semantics directly from the  
103 equational theory induced by the encoding of the semantics using CTrees; for the language  
104 with cooperative threading, we prove some standard program equivalences.

105 To summarize, this paper makes the following contributions:

- 106 • We introduce CTrees, a novel data structure for defining nondeterministic computa-  
107 tions in type theory, along with a set of combinators for building semantic objects  
108 using CTrees.
- 109 • We develop a theory of strong bisimilarity (resp. similarity) of CTrees as equivalence  
110 (resp. refinement), and establish their metatheory. Doing so, we connect their  
111 semantics to standard notions from labeled transition systems.
- 112 • We demonstrate that, as with ITrees, CTrees support interpretation of events, and  
113 introduce a novel notion of implementation of branching nodes, opening the way to  
114 reason formally about schedulers, and giving a framework to tweak the executability  
115 of CTree models.
- 116 • We demonstrate how to use CTrees in two case studies: (1) to define a semantics for  
117 Milner’s classic ccs and prove that the resulting derived equational theory coincides  
118 with the one given by the standard operational semantics, and (2) to model in stages  
119 cooperative multithreading with support for `fork` and `yield` operations and prove  
120 nontrivial program equivalences.
- 121 • We develop theories for more notions of equivalence and refinement: an alternate  
122 characterization of strong (bi)similarity easing some proofs, as well as notions of  
123 weak bisimilarity, complete similarity, and heterogeneous (bi)similarity.
- 124 • Finally, our library show cases at scale that modern libraries (Hur et al., 2013; Pous,  
125 2016) enable elegant coinductive reasoning in Rocq.

126 All of our results have been implemented in the Rocq prover (formerly known as the Coq  
127 proof assistant), and all claims in this paper are fully mechanically verified. For expository  
128 purposes, we stray away from Rocq’s syntax in the body of this paper, but systematically  
129 link our claims to their formal counterpart via hyperlinks represented as (🔗).

130 The remainder of the paper is organized as follows. The next section gives some back-  
131 ground about interaction trees and monadic interpreters, along with a discussion of the  
132 challenges of modeling nondeterminism in such context. We introduce the CTrees data  
133 structure and its main combinators in Section 3. Section 4 describes our first case study,  
134 a model for ccs. Section 5 introduces (coinductive) equality, strong bisimilarity, strong  
135 similarity, and trace equivalence for CTrees—and establishes its core equational theory.  
136 Section 6 applies this equational theory to the ccs case study. Section 7 describes how  
137  
138

```

139 CoInductive itree (E: Type → Type) (R: Type) : Type :=
140   (* computation terminating with value r *)
141   | Ret (r: R)
142   (* event e yielding an answer in A *)
143   | Vis {A: Type} (e : E A) (k : A → itree E R)
144   (* "silent tau" transition with child t *)
145   | Step (t: itree E R).

```

Fig. 1: Interaction trees: definition

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 8 describes our second case study, a model for the `imp` language extended with cooperative multithreading. Section 9 gives alternative characterization of strong bisimilarity and strong similarity for CTrees, enabling new proof techniques. Section 10 studies finer notions of (bi)similarity for CTrees: heterogeneous relations, complete similarity, and weak bisimilarity. Finally, Section 11 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 reimplementaion of our library.

## 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 Rocq ecosystem, interaction trees (Xia et al., 2019) 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 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. 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

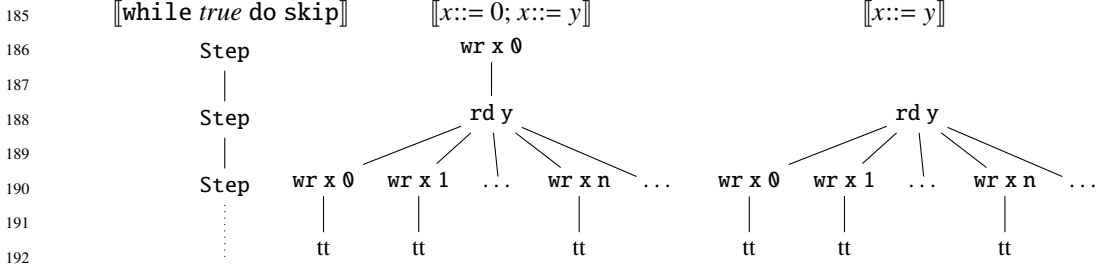


Fig. 2: Example ITrees denoting the `imp` programs  $p_1$ ,  $p_2$ , and  $p_3$ .

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 a non-observable step of computation, allowing the representation of silently diverging computations; it is also used for guarding corecursive definitions.<sup>1</sup>

We illustrate the ITrees approach by defining the semantics for a simple imperative programming language, `imp`. The language contains a `skip` construct, assignments, sequential composition, and loops—we assume a simple language of expressions,  $e$ .

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

Consider the following `imp` programs:

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

In the style of Xia et al. (2019), 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, as 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 equivalence

<sup>1</sup> The ITree library uses `Tau` to represent `Step` nodes. `Tau` and  $\tau$  are overloaded in our context, so we rename it to `Step` here to avoid ambiguity.

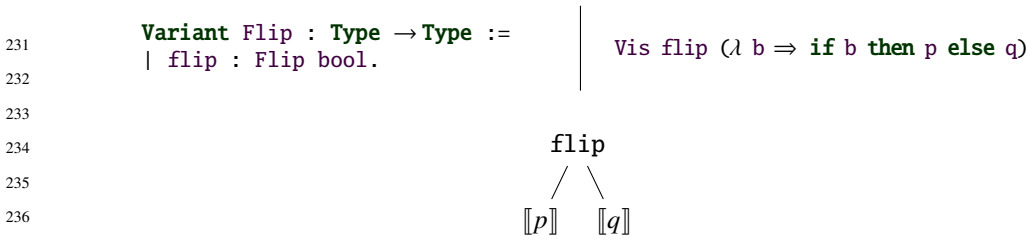


Fig. 3: A boolean event, an example of its use, and the corresponding CTree.

used for computations in the ITree monad is a *weak bisimulation*, dubbed *equivalence up-to taus* (`eutt`), which ignores finite sequences of `Step` nodes. It is 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, their trees are not `eutt`. 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)` where `voidE` is the “empty” event signature. 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. For  $p_2$  and  $p_3$ , assuming an initial state  $m$ , the computations become:<sup>2</sup>

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

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 in Figure 3

Naturally, as with memory events, `flip` does not come with its expected algebra: associativity, commutativity and idempotence. We therefore seek to interpret `flip` into an executable monad in which we recover these necessary equations, and furthermore combine them with the state algebra in order to establish program equivalences such as  $p_3 \equiv \text{br } p_2 \text{ or } p_3$ .

CTrees will form a suitable monad for modeling such nondeterministic effects. The core idea is to build an ITree-like coinductive data structure, with an additional kind of node to represent nondeterminism. The resulting definitions are very expressive. As foreseen, they form a proper monad, validating all monadic laws up-to coinductive structure equality, they

<sup>2</sup> Writing the trees horizontally to save space

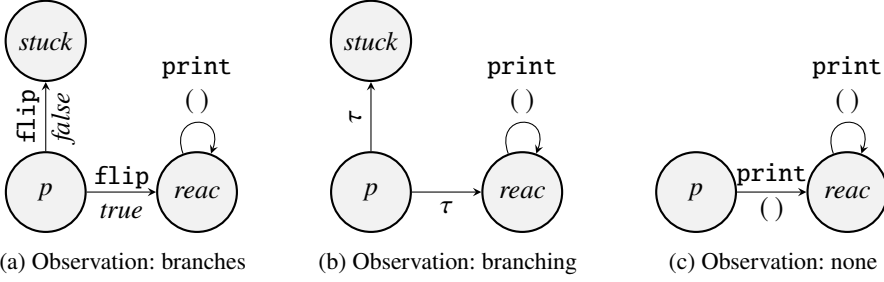


Fig. 4: Three possible semantics for the program  $p$ , from an LTS perspective

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`.

$$\text{comm} \triangleq \text{skip} \mid x ::= e \mid c_1; c_2 \mid \text{while } b \text{ do } c \mid \text{br } c_1 \text{ or } c_2 \mid \text{block} \mid \text{print}$$

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. The former corresponds in the literature to the same kind of choice that exists in `ccs` (Milner, 1989). The latter can be thought of as a form of *internal* choice, found natively as well in CSP, and encoded in `ccs` by guarding processes by an internal  $\tau$  step. We shall emphasize this analogy in Section 4 when providing a model for `ccs`.

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, we think of `imp` programs as labeled transition systems. 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  $\llbracket \text{br } p \text{ or } q \rrbracket \triangleq \text{Vis } \text{flip } \lambda b \cdot \text{if } b \text{ then } \llbracket p \rrbracket \text{ else } \llbracket q \rrbracket$ .

Figure 4b corresponds to  $\text{BR}_{\text{INTERNAL}}$ : 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  $\tau$  step of process algebra. This situation could<sup>3</sup> be captured by introducing a new kind of node in our data structure, a  $Br_S$  branch, that maps in our bisimulations defined in Section 5 to a nondeterministic *internal step*. For this semantics, we thus have  $\llbracket \text{br } p \text{ or } q \rrbracket \triangleq Br_S^2 \lambda b \cdot \text{if } b \text{ then } \llbracket p \rrbracket \text{ else } \llbracket q \rrbracket$ .<sup>4</sup>

Figure 4c corresponds to  $\text{BR}_{\text{DELAYED}}$ , but raises the question: how do we build such a behavior? It could be the responsibility of the model, i.e., the function mapping `imp`'s syntax to the semantic domain, `CTrees`, to explicitly compute this LTS. Here,  $\llbracket p \rrbracket$  would be an infinite sequence of `Vis print` nodes, containing no other node. But recall we seek to *compute* our models, typically by recursion on the syntax. But to build this LTS directly, the model for `br p or q` would then need 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. In general, statically determining whether a given program is semantically equivalent to `block` is intractable: consider a computation that checks the convergence of the Collatz sequence on its input before performing an observable event for instance. The introspection required to directly compute this LTS is therefore hard (or impossible) to implement in general.

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  $\text{BR}_{\text{DELAYED}}$  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.

### 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 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. This `B` parameter is an improvement in expressiveness over the first iteration of `CTrees` (Chappe et al., 2023), which only supported unlabelled finite choice; It will be useful in Section 7.2. We discuss in depth the differences between both implementations in Section 11.

`CTrees` are coinductive trees<sup>5</sup> with four main kinds of nodes: pure computations (`Ret`), external events (`Vis`), unary node with an implicitly associated  $\tau$  step (`Step`), and delayed

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

<sup>4</sup> The 2 indicates the arity of the branching.

<sup>5</sup> The implementation uses a negative style with primitive projections. We omit this technical detail in the presentation.

```

369 (* Core datatype *)
370 CoInductive ctree (E B : Type → Type) (R : Type) :=
371 | Ret (r : R) (* pure computation *)
372 | Step (t : ctree) (* internal step *)
373 | Vis {X : Type} (e : E X) (k : X → ctree) (* external event *)
374 | Br {X : Type} (c : B X) (k : X → ctree) (* delayed branching *)
375 | Guard (t : ctree) (* unary Br *)
376 | Stuck (* stuck process *)
377
378 (* Bind, sequencing computations *)
379 CoFixpoint bind {E T U} (t : ctree E B T) (k : T → ctree E B U)
380 : ctree E B U :=
381 match u with
382 | Ret r ⇒ k r
383 | Stuck ⇒ Stuck
384 | Step t ⇒ Step (bind t k)
385 | Guard t ⇒ Guard (bind t k)
386 | Vis e h ⇒ Vis e (λ x ⇒ bind (h x) k)
387 | Br b h ⇒ Br b (λ x ⇒ bind (h x) k)
388 end
389
390 (* Stepping branching *)
391 Definition BrS {X : Type} (c : B X) (k : X → ctree) :=
392 Br c (λ x ⇒ Step (k x))
393
394 (* Main fixpoint combinator *)
395 CoFixpoint iter {I : Type} (body : I → ctree E B (I + R))
396 : I → ctree E B R :=
397 bind (body i) (λ lr ⇒ match lr with
398 | inr r ⇒ Ret r
399 | inl i ⇒ Guard (iter body i)
400 end)
401
402 Notation "E ~ F" := (∀ X, E X → F X)
403 (* Atomic ctree triggering a single event *)
404 Definition trigger : E ~ ctree E B := λR (e : E R) ⇒ Vis e (λ x ⇒ Ret x)
405 (* Atomic branching ctrees *)
406 Definition branch : B ~ ctree E B := λR (b : B R) ⇒ Br b (λ x ⇒ Ret x)
407 Definition branchS : B ~ ctree E B := λR (b : B R) ⇒ BrS b (λ x ⇒ Ret x)

```

Fig. 5: CTree structure definition (👉)

branching (**Br**). Nullary (**Stuck** constructor) and unary (**Guard** constructor) delayed 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 delayed branching is indexed by the return type specified by the emitted branch. When using finite branching in examples, we assume a suitable branching interface **B** that includes such finite indexed types, and we abusively write, for instance,  $Br^2 t u$  for a computation branching on a finite type with two inhabitants, and  $B_2$  the corresponding signature it draws from, rather than explicitly spelling out an event with a boolean signature and the continuation that branches on the boolean index:  $\lambda i \Rightarrow \text{match } i \text{ with } 0 \Rightarrow t \mid 1 \Rightarrow u \text{ end}$ . We write **Stuck** nodes as “0” in equations.

```

415  (* Stuck processes *)
416  Definition stuck : ctree E B void := Stuck
417  Definition stuckE (e : E void) : ctree E B void :=
418    trigger e
419  Definition stuckB (b : B void): ctree E B void :=
420    branch b
421  CoFixpoint stagnate : ctree E B R := Guard stagnate
422  CoFixpoint stagnate_nary n (bn : B (fin n)) : ctree E B R
423    := branch bn ;; stagnate_nary n bn
424
425  (* Spinning processes *)
426  CoFixpoint spinS : ctree E B R := Step spinS
427  CoFixpoint spinS_nary n (bn : B (fin n)) : ctree E B R :=
428    branchS bn ;; spinS_nary n
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460

```



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

The remainder of Figure 5 displays (superficially simplified) definitions of the core combinators. In particular, step branching that materializes internal choice, denoted  $Br_S$  and informally introduced in Section 2.2, is defined as the composition of  $Br$  and  $Step$ . The minimal computations respectively triggering an event  $e$ , delaying a branch, or generating observable branching, are defined as `trigger`, `branch` and `branchS`.

As expected, `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` insert a `Step` node between iterations for this purpose, treated weakly by the `eutt` equivalence built on the structure. Here, we instead use the `Guard` constructor, encoding a unary non-observable branch. A `Step` node would also be a valid choice, with different resulting semantics, this option will be further discussed in Section 7.4.

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 chooses among none. More convoluted, the `stagnate` and `stagnate_nary` trees are infinitely deep structures made of delayed branches; they yet never find in their structure a transition to take. In contrast, the `spinS` and `spinS_nary` processes exhibit a different behavior, depicted in the `spinS` LTS: they generate infinite traces of  $\tau$ s.

Section 5 will introduce the necessary notions of equivalence on `CTrees`, bisimilarity in particular, to formally express and prove the semantic equivalence between these various representations of stuck processes.

**Example: ImpBr.** In this example, and the remaining of the paper, we consider the `BRDELAYED` operational semantics for the `ImpBr`'s branching operator, as per Section 2.2.

We define a representation `repr_imp : comm → ctree (MemE + PrintE) B2 unit` by recursion on the syntax: `ImpBr` programs are computations that may diverge, draw from binary choices, and emit read and write events to the memory, or printing signals. Assuming that we have already defined similarly the representation for expressions `repr_expr`:

```

467 Fixpoint repr_imp (s : comm) : computation unit :=
468   match s with
469     | Assign x e ⇒ v ← repr_expr e ;; trigger (wr x v) (* x ::= e *)
470     | Print      ⇒ trigger print                      (* print *)
471     | Seq c1 c2 ⇒ repr_imp c1 ;; repr_imp c2          (* a ; b *)
472     | While b c ⇒
473       while (v ← repr_expr b ;;
474             if is_true v
475             then repr_imp c ;; ret (inl tt)
476             else ret (inr tt))
477     | Branch c1 c2 ⇒ Br2 (repr_imp c1) (repr_imp c2) (* br c1 or c2 *)
478     | Skip ⇒ Ret tt                                (* skip *)
479     | Block ⇒ Stuck                                (* block *)
480   end.

```

Similarly to `ITrees`, `trigger` is used for emitting events, monadic binds and returns for sequences and terminated programs, and the `iter` combinator is used to encode a suitable `while` looping combinator.<sup>6</sup> Finally, the two unusual constructs for an `Imp` language, branches and stuck programs, directly map to two new primitive concepts in `CTrees`.

### 3.2 Interpretation

`ITrees` support interpretation: provided a *handler*  $h : E \rightsquigarrow M$  implementing a signature of events  $E$  into a suitable monad  $M$ , the `(interp h) : itree E → M` combinator provides an implementation of any computation into  $M$ .<sup>7</sup> The only restriction imposed on the target monad  $M$  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  $M$ . For this implementation to be sensible and amenable to verification in practice, one must, however, check an additional property: `interp h` should form a monad morphism—in particular, it should map `eutt` `ITrees` to equivalent monadic computations in  $M$ .

Unsurprisingly, given their structure, `CTrees` enjoy their own `interp` combinator. Its definition, provided in Figure 7, 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, the target monad must also be able to internalize nondeterminism by providing a stuck state (`mStuck`), an observable tick (`mStep`), and a nondeterministic branching (`mBr`). `CTrees` and stateful `CTrees` are both valid target monads, we provide straightforward instances for them.

We shall see that handlers must satisfy some conditions for the induced interpretation to be well-behaved: we defer this discussion to Section 7 once all the necessary tools have been introduced.

<sup>6</sup> We omit its definition and refer the reader to our formal development.

<sup>7</sup> Recall that  $E \rightsquigarrow F$  is defined as  $\forall X, E X \rightarrow F X$ .

```

507 Definition interp (h : E  $\rightsquigarrow$  M) : ctree E B  $\rightsquigarrow$  M :=  $\lambda$ R  $\Rightarrow$ 
508   iter ( $\lambda$  t  $\Rightarrow$  match t with
509     | Ret r  $\Rightarrow$  ret (inr r)
510     | Stuck  $\Rightarrow$  mStuck
511     | Guard t  $\Rightarrow$  ret (inl t)
512     | Step t  $\Rightarrow$  bind mstep ( $\lambda$  _  $\Rightarrow$  inl t)
513     | Br n k  $\Rightarrow$  bind (mBr n) ( $\lambda$  x  $\Rightarrow$  ret (inl (k x)))
514     | Vis e k  $\Rightarrow$  bind (h e) ( $\lambda$  x  $\Rightarrow$  ret (inl (k x)))
515   end).

```

Fig. 7: Interpreter for CTrees (class constraints omitted) (👉)

**Example: ImpBr.** We can now complete our model for ImpBr by implementing read and write events using a stateful handler:

```

518 Definition h_imp : (MemE + PrintE)  $\rightsquigarrow$  stateT env (ctree PrintE B2) :=
519    $\lambda$  _ e s  $\Rightarrow$ 
520     match e with
521     | rd x  $\Rightarrow$  Ret (s, lookup_default x 0 s)
522     | wr x v  $\Rightarrow$  Ret (Maps.add x v s, tt)
523     | print  $\Rightarrow$  trigger print
524   end.

```

Read and write events are implemented concretely over a domain of maps `env`, while printing events are retrIGGERED, and hence left uninterpreted. The final model is therefore  $\llbracket c \rrbracket \triangleq \text{interp } h\_imp \text{ (repr\_imp } c)$ .

### 3.3 A Hint of Introspection: Heads of Computations

*Br* nodes prevent the need for introspection over trees when modeling 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 (i.e., the non-*Br* nodes immediately reachable from the root after a series of *Br* nodes). This is the case for many parallel operators, including the one of `CCS` that we model in Section 4. 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 in Figure 8, 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.

As such, `head` provides a constructive computation of the set of visible actions a CTREE computation can draw. We demonstrate in the next section how it is used to build a model for `CCS`, but illustrate it first on an artificial example.

```

553 Variant action E B R :=
554   | ARet (r : R)
555   | AStep (t : ctree E B R)
556   | AVis {X} (e : E X) (k : X → ctree E B R).
557
558 CoFixpoint head {E B R} (t : ctree E B R) : ctree E B (action E B R) :=
559   match t with
560   | Ret r ⇒ Ret (ARet r)
561   | Stuck ⇒ Stuck
562   | Step t ⇒ Ret (AStep t)
563   | Guard t ⇒ Guard (head t)
564   | Vis e k ⇒ Ret (AVis e k)
565   | Br b k ⇒ Br b (λ x ⇒ head (k x))
566 end.

```

Fig. 8: Lazily computing the set of reachable observable nodes (👉)

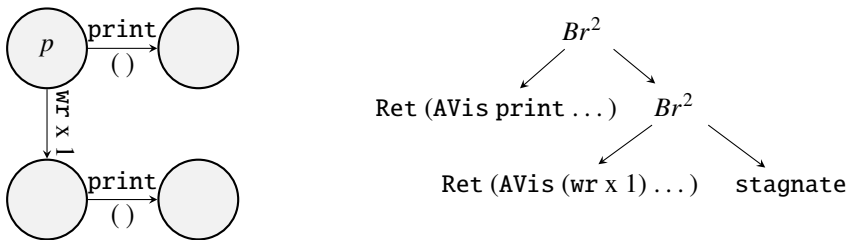


Fig. 9: The LTS for program  $p$  (on the left) and its head CTree (on the right)

**Example: ImpBr.** Consider  $p \triangleq \text{br}(\text{print})$  or  $\text{br}(x := 1; \text{print})$  or  $(\text{while } \text{true} \text{ do skip})$ , a program that may have three behaviors: printing, writing in memory then printing, or looping. Figure 9 represents on the left the LTS of  $\text{repr\_imp}(p)$ : the looping behavior is not observable, as its model is *stagnate*—we revisit formally this representation in terms of LTS in Section 5. On the right, it shows the CTree  $\text{head}(\text{repr\_imp}(p))$ : notice that detecting that *stagnate* will exhibit no observable behavior is not computable in general, the head therefore also exhibits a stagnating branch. The two other branches return respectively the printing and writing events. The dots in the Ret nodes stand for the continuations. In particular, the continuation for the  $\text{wr } x \ 1$  event contains the subsequent print event.

#### 4 Case Study: ccs, a Model

We claim that CTrees form a versatile tool for building semantic models of nondeterministic systems, concurrent ones in particular. In this section, we illustrate the use of CTrees as a model of concurrent communicating processes by providing a denotational semantics for Milner’s Calculus of Communicating Systems (ccs) (Milner, 1989). The result is a shallowly embedded model for ccs in Rocq that could be easily, and modularly, combined with other language features.

$$\begin{array}{c}
599 \quad l ::= \tau \mid c \mid \bar{c} \quad P ::= 0 \mid l.P \mid P+Q \mid P \parallel Q \mid \nu c.P \mid !P \\
600 \\
601 \quad \frac{}{a.P \xrightarrow{\text{ccs}} P} \quad \frac{P \xrightarrow{l} P'}{P+Q \xrightarrow{l} P'} \quad \frac{Q \xrightarrow{l} Q'}{P+Q \xrightarrow{l} Q'} \quad \frac{P \xrightarrow{l} P'}{P \parallel Q \xrightarrow{l} P' \parallel Q} \\
602 \\
603 \\
604 \\
605 \quad \frac{Q \xrightarrow{l} Q'}{P \parallel Q \xrightarrow{l} P \parallel Q'} \quad \frac{P \xrightarrow{c} P' \quad Q \xrightarrow{\bar{c}} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'} \quad \frac{P \xrightarrow{l} P' \quad l \notin \{c, \bar{c}\}}{\nu c.P \xrightarrow{l} \nu c.P'} \\
606 \\
607 \\
608 \\
609 \quad \frac{P \parallel !P \xrightarrow{l} P'}{!P \xrightarrow{l} P'} \\
610 \\
611
\end{array}$$

Fig. 10: Syntax for ccs (👉) and its operational semantics (👉)

### 4.1 Syntax and Operational Semantics

The syntax and operational semantics of ccs are shown in Figure 10. The language assumes a set of *names*, or communication channels, ranged over by  $c$ . For any name  $c$ , there is a co-name  $\bar{c}$  satisfying  $\bar{\bar{c}} = c$ . An *action* is represented by a label  $l$ ; it is either a communication label  $c$  or  $\bar{c}$ , representing the sending/reception of a message on a channel, or the reserved action  $\tau$ , 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  $l.P$  emits an action  $l$  and then becomes the process  $P$ . The choice operator  $P+Q$  behaves either like the process  $P$ , or like the process  $Q$ , in the same fashion as the  $\text{BR}_{\text{DELAYED}}$  semantics for  $\text{br}$  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  $\tau$ . Channel restriction  $\nu c.P$  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  $P \parallel !P$ .

### 4.2 Model

We define a denotational model for ccs using  $\text{ctree actE ccsB void}$  as domain, written  $\text{ccs}^\#$  in the following. As witnessed by this type, processes do not return any value, but may emit actions modeled as external events expecting  $\text{unit}$  for answer: **Inductive**  $\text{actE} ::= \mid \text{act } a : \text{actE unit}$ . Because the semantics of CCS limits the possible different outcomes at each program point, its model can exhibit only binary, ternary, or quaternary branches, captured in  $\text{ccsB}$ . Specifically, a ternary branch is used to express the behaviour of the parallel operator where either the left sub-term or the right sub-term progresses, or a synchronisation between the two sub-terms happens. Quaternary branches are

$$\begin{aligned}
645 \quad & \bar{0} \triangleq \emptyset & a:p \triangleq \text{trigger } a ;; p & \quad p \bar{+} q \triangleq Br^2 p q & \quad \bar{!}p \triangleq p \bar{||} !p \\
646 & & & & & \\
647 \quad & \nu c.P \triangleq \text{interp } h_{\text{new}} c P & \quad \text{where } h_{\text{new}} c e = \begin{cases} \emptyset & \text{if } e = \text{act } c \text{ or } e = \text{act } \bar{c} \\ \text{trigger } e & \text{otherwise} \end{cases} \\
648 & & & & & \\
649 & & & & & \\
650 \quad & p \bar{||} q \triangleq \text{cofix } F p q \cdot Br^3 & \quad (p' \leftarrow \text{head } p ;; \text{actL } F q p') \\
651 & & & & & \quad (q' \leftarrow \text{head } q ;; \text{actR } F p q') \\
652 & & & & & \quad (p' \leftarrow \text{head } p ;; q' \leftarrow \text{head } q ;; \text{actLR } F p' q') \\
653 & & & & & \\
654 & & & & & \text{actL } F q (AStep t) \triangleq Step (F t q) \\
655 & & & & & \text{actL } F q (AVis e k) \triangleq Vis e (\lambda i \cdot F (k i) q) \\
656 & & & & & \\
657 & & & & & \text{actR } F p (AStep t) \triangleq Step (F p t) \\
658 & & & & & \text{actR } F p (AVis e k) \triangleq Vis e (\lambda i \cdot F p (k i)) \\
659 & & & & & \\
660 & & & & & \text{actLR } F r r' \triangleq \\
661 & & & & & \begin{cases} \text{Step} \cdot F (k ()) (k' ()) & \text{if } \exists a. r = \text{Vis } (act a) k \wedge r' = \text{Vis } (act \bar{a}) k' \\ \emptyset & \text{otherwise} \end{cases} \\
662 & & & & & \\
663 & & & & & \\
664 & & & & & \\
665 & & & & & \\
666 & & & & & \\
667 & & & & & \\
668 & & & & & \\
669 & & & & & \\
670 & & & & & \\
671 & & & & & \\
672 & & & & & \\
673 & & & & & \\
674 & & & & & \\
675 & & & & & \\
676 & & & & & \\
677 & & & & & \\
678 & & & & & \\
679 & & & & & \\
680 & & & & & \\
681 & & & & & \\
682 & & & & & \\
683 & & & & & \\
684 & & & & & \\
685 & & & & & \\
686 & & & & & \\
687 & & & & & \\
688 & & & & & \\
689 & & & & & \\
690 & & & & &
\end{aligned}$$

Fig. 11: Denotational model for ccs using  $\text{ccs}^\#$  as a domain (👉)

required for the semantics of the replication operator—we detail its model below. Naturally, nested operators will lead to nested branches.

Figure 11 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  $\text{ccs}^\#$ .

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 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 `hnew`, 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 3.3. 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 initiate the computation by sequencing the heads of  $p$  and  $q$ , as divergence in the former

691  
692  
693  
694  
695  
696  
697  
698  
699  
700  
701  
702  
703  
704  
705  
706  
707  
708  
709  
710  
711  
712  
713  
714  
715  
716  
717  
718  
719  
720  
721  
722  
723  
724  
725  
726  
727  
728  
729  
730  
731  
732  
733  
734  
735  
736

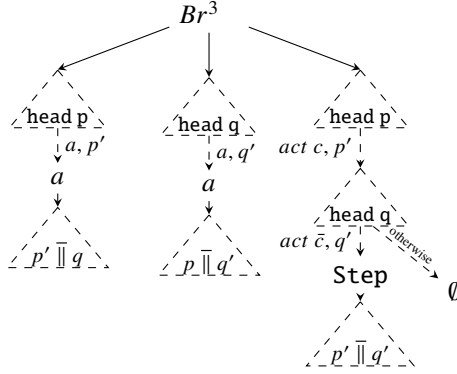


Fig. 12: Depiction of the tree resulting from  $p \parallel q$

may render inaccessible valid transitions in the latter.<sup>8</sup> 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 12 shows the CTree resulting from  $p \parallel q$ .

The last operator to consider is the replication  $\bar{\cdot}$ . In theory, it could be expressed in terms of parallel composition directly, as the cofix  $\bar{!}p \triangleq p \parallel \bar{!}p$ . Unfortunately, although it is sound, defining the  $\bar{\cdot}$  operator in this way is too involved for Rocq's syntactic criterion on cofixes to recognize that the corecursive call is guarded under  $\parallel$ . To circumvent this difficulty, we use an auxiliary operator, defined in Figure 13, and define the replication operator as  $\bar{!}p \triangleq p \parallel !q$ . The intuition behind this operator,  $p \parallel !q$ , 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 Rocq. The operator  $p \parallel !q$  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.

Finally, we define the model  $\llbracket \cdot \rrbracket : \text{ccs} \rightarrow \text{ccs}^\#$  by recursion on the syntax.

<sup>8</sup> 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.

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

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

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

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

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

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

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

Fig. 13: Definition of the auxiliary operator  $p \overline{\parallel}!$  (👉)

## 5 Strong (Bi)similarity and Associated Equational Theory for CTrees

Section 3 introduced CTrees as domain of computations, 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.

Readers familiar with ITrees may recall that their notions of equivalence, strong and weak bisimilarity, are defined syntactically on ITrees, and that strong bisimulation even corresponds to structural coinductive equality on ITrees. Because of the more subtle semantics of  $Br$  nodes, equality and strong bisimulation are distinct notions on CTrees, and definitions are more involved. In particular, our notions of refinement and equivalence use a formalization of the LTS representation of a CTree that we have sketched.

Through this section, we first introduce a (coinductive) syntactic equality of CTrees (Section 5.1). We then take an LTS view on CTrees (Section 5.2) to lift standard relations on processes to ctrees: namely, strong bisimilarity (Section 5.3) and strong similarity (Section 5.4). We equip both of these notions with primitive up-to principles and establish an equational theory for CTrees. Section 5.5 illustrates equational reasoning on CTrees with a detailed example of a similarity proof.

The theories discussed throughout this section provide the building blocks necessary for deriving domain-specific equational theories, such as the ones established in Section 6 for ccs, and in Section 8 for cooperative scheduling. Furthermore, additional reasoning principles for strong (bi)similarity are introduced in Section 9, and other equivalences and refinements are defined in Section 10.

$$\begin{aligned}
\text{REFL}^{up} R &\triangleq \{(x, x)\} & \text{SYM}^{up} R &\triangleq \{(y, x) \mid R x y\} \\
\text{TRANS}^{up} R &\triangleq \{(x, z) \mid \exists y, R x y \wedge R y z\} \\
\text{BIND}^{up}(\text{EQUIV}) R &\triangleq \{(x \ggg k, y \ggg l) \mid \text{EQUIV } x y \wedge \forall v, R (k v) (l v)\} \\
\text{UPTO}^{up}(\text{EQUIV}) R &\triangleq \{(x, y) \mid \exists x' y', \text{EQUIV } x x' \wedge R x' y' \wedge \text{EQUIV } y' y\}
\end{aligned}$$

Fig. 14: Main generic up-to principles used for relations of CTrees where  $R : \text{rel } (\text{ctree } E B X)$

**Conventions and preliminary definitions.** We define in this section some *coinductive* relations, and establish their meta-theory. A coinductive relation is defined as the greatest relation of an adequate family of relations: for instance, bisimilarity, written  $\sim$ , is the greatest bisimulation.

The meta-theory of such relations is essentially composed of two levels. On one hand, top-level proof rules give means of proving that two computations are bisimilar. On the other hand, *up-to principles* establish additional proof rules *valid during a proof by coinduction of bisimilarity*, resulting in an *enhanced coinduction principle*.

To distinguish the latter, we write  $\sim_{\mathcal{R}}$  the goal of an ongoing proof of bisimulation with candidate  $\mathcal{R}$ . From there, such proof rules can be of two natures: if the premise is expressed in terms of  $\sim_{\mathcal{R}}$ , its application is valid, but does not give access to the coinduction hypothesis; otherwise, its premise is expressed in terms of  $\mathcal{R}$  and it gives access to the coinduction hypothesis.

We refer the interested reader to Appendix A for a more detailed introduction to these concepts, and to their mechanization in Rocq.

Figure 14 describes some main generic up-to principles we use for our relations on CTrees.<sup>9</sup> Recall that over  $C$ , these potential up-to functions are endofunctions of relations: for instance,  $\text{REFL}^{up}$  is the constant diagonal relation,  $\text{SYM}^{up}$  builds the symmetric relation,  $\text{TRANS}^{up}$  is the composition of relations. The validity of  $\text{REFL}^{up}$ ,  $\text{SYM}^{up}$  and  $\text{TRANS}^{up}$  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 during a proof by coinduction up-to companion: the former as our goal, the latter as our coinduction hypothesis. The  $\text{BIND}^{up}(\_)$  up-to function helps when reasoning structurally, allowing to cross through `bind` constructs during proofs by coinduction. Finally, validity of the  $\text{UPTO}^{up}(\text{EQUIV})$  principle allows for rewriting via the *equiv* relation during coinductive proofs for  $b$ .

### 5.1 Coinductive Equality for CTree

Rocq'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

<sup>9</sup> 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.

therefore define, as is standard, a structural equality<sup>10</sup> by coinduction  $\text{equ} : \text{rel}(\text{ctree } E \text{ B } A)$  (written  $\cong$  in infix). The endofunction simply matches head constructors and behaves extensionally on continuations.

**Definition 1** (Structural equality (☞)).

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

The  $\text{equ}$  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  $\text{equ}$ .

**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  $\text{eq}$  comes at the usual cost: all constructions introduced over CTrees must be proved to respect  $\text{equ}$  (in Rocq parlance, they must be **Proper**), allowing us to work painlessly with setoid-based rewriting.<sup>11</sup>

Finally, we establish some enhanced coinduction principles for  $\text{equ}$ .

**Lemma 2** (Enhanced coinduction for  $\text{equ}$  (☞)).  $\text{REFL}^{up}$ ,  $\text{SYM}^{up}$ ,  $\text{TRANS}^{up}$ ,  $\text{BIND}^{up}(\cong)$  (☞) and  $\text{UPTO}^{up}(\cong)$  (☞) provide valid up-to principles for  $\text{equ}$ .

With these lemmas, one has the ability, *in the middle of a proof by coinduction aiming to establish that two trees are  $\text{equ}$* , to invoke reflexivity, symmetry, transitivity, congruence for  $\text{bind}$ , and to rewrite previously established equations.

While  $\text{equ}$ , 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 branching nodes, 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.

<sup>10</sup> Note that for ITrees, this relation corresponds to what Xia et al. dub as *strong bisimulation*, and name  $\text{eq\_itree}$ . We carefully avoid this nomenclature here to reserve this term for the relation we define in Section 5.3

<sup>11</sup> As is the case for the  $\text{eq\_itree}$  over ITrees, postulating as an axiom that  $\text{equ}$  coincide with definitional equality would be sound in Rocq. We are however doomed to embrace setoids anyway for all other relations, we therefore avoid doing so.

$$\begin{array}{c}
875 \qquad \text{label} \triangleq \text{tau} \mid \text{obs } e \ v \mid \text{val } v \\
876 \\
877 \qquad \frac{k \ v \xrightarrow{l} t}{\text{Br}^b \ k \xrightarrow{l} t} \qquad \frac{t \xrightarrow{l} u}{\text{Guard } t \xrightarrow{l} u} \qquad \frac{}{\text{Step } t \xrightarrow{\text{tau}} t} \qquad \frac{}{\text{Vis } e \ k \xrightarrow{\text{obs } e \ v} k \ v} \qquad \frac{}{\text{Ret } v \xrightarrow{\text{val } v} \emptyset} \\
878 \\
879
\end{array}$$

Fig. 15: Inductive characterization of the LTS induced by a CTree (👉)

## 5.2 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 15. This LTS exhibits three kinds of labels: a  $\text{tau}$ <sup>12</sup> witnesses a stepping branch, an  $\text{obs } e \ x$  observes the encountered event together with the answer from the environment considered, and a  $\text{val } v$  is emitted when returning a value. Note that there is a significant mismatch between the structure of the tree and the induced LTS: each state of the LTS corresponds, in the CTree, to a node *that is not immediately preceded by a Br node*. Accordingly, the definition of the transition relation between states inductively iterates over delayed branches. On the contrary, stepping branches and visible nodes map immediately to a set of transitions, one for each outgoing edge; finally a return node generates a single  $\text{val}$  transition, moving onto a stuck state, encoded as the  $\emptyset$  constructor. These rules formalize the intuition we gave in Section 2.2 and 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:  $t \rightarrow \triangleq \forall u, \neg(t \xrightarrow{l} u)$ , we can make the depictions of stuck processes from Figure 6 precise:  $\emptyset$  itself and nullary nodes are stuck by construction, since stepping would require a branch, while  $\text{stagnate\_nary}$  and  $\text{stagnate}$  are proven to be stuck by induction on the transition relation (👉). It follows that  $\emptyset$ ,  $\text{stagnate\_nary}$  and  $\text{stagnate}$  are semantically undistinguishable.

$$\emptyset \rightarrow \qquad \text{Br}^0 \rightarrow \qquad \text{stagnate\_nary } n \ b_n \rightarrow$$

The stepping relation interacts slightly awkwardly with  $\text{bind}$ : indeed, although a unit for  $\text{bind}$ , the  $\text{Ret}$  construct is not inert from the perspective of the LTS: it has one outgoing transition labeled with the return value. Non  $\text{val}$  transitions can therefore be propagated below the left-hand side of a  $\text{bind}$ , while a  $\text{val}$  transition in the prefix does not entail the existence of a transition in the  $\text{bind}$ . Figure 16 describes lemmas capturing this intuition, by distinguishing cases of a  $\text{val } v$  transition ( $\text{trans\_bind\_r}$  for backward reasoning and the second case of the conclusion of  $\text{trans\_bind\_inv}$  for forward reasoning about  $\text{bind}$ ) or another kind of transition ( $\text{trans\_bind\_l}$  and the first case of  $\text{trans\_bind\_inv}$ ).

<sup>12</sup> We warn again the reader accustomed to ITrees to think of  $\text{tau}$  under the lens of the process algebra literature, and not as a representation of ITree's  $\text{Tau}$  constructor.

$$\begin{array}{c}
 921 \quad \frac{t \xrightarrow{l} u \quad l \neq \text{val } v}{t \gg k \xrightarrow{l} u \gg k} \text{ (trans\_bind\_l)} \qquad \frac{t \xrightarrow{\text{val } v} \emptyset \quad k \xrightarrow{l} u}{t \gg k \xrightarrow{l} u} \text{ (trans\_bind\_r)} \\
 922 \\
 923 \\
 924 \\
 925 \quad \frac{t \gg k \xrightarrow{l} u}{(l \neq \text{val } v \wedge \exists t', t \xrightarrow{l} t' \wedge u \cong t' \gg k) \vee (\exists v, t \xrightarrow{\text{val } v} \emptyset \wedge k \xrightarrow{l} u)} \text{ (trans\_bind\_inv)} \\
 926 \\
 927
 \end{array}$$

Fig. 16: Lemmas for transitions under bind (🔴)

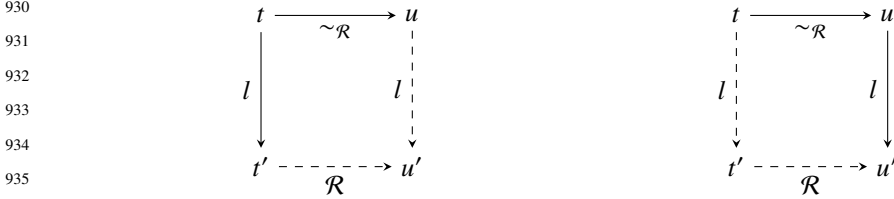


Fig. 17: The bisimulation game  $\sim_{\mathcal{R}}$

### 5.3 Bisimilarity

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

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

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

and conversely

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

The bisimulation game is represented visually in Figure 17. Solid lines represent universal hypotheses and dashed ones represent existential conclusions. Bisimilarity, written  $t \sim u$ , is defined as the greatest fixpoint of  $sb$ :  $\text{sbisim} \triangleq \text{gfp } sb$ .

All the traditional tools surrounding bisimilarity can be transferred to our setup. The rest of this subsection shows the proof rules and up-to principles that are proved valid for CTree bisimilarity.

#### 5.3.1 Core Equational Theory

Bisimilarity forms an equivalence relation satisfying a collection of primitive laws for CTrees summed up in Figure 18. 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.

$$\begin{array}{c}
967 \quad \frac{x = y}{\text{Ret } x \sim \text{Ret } y} \quad \frac{\forall x, h x \sim k x}{\text{Vis } e h \sim \text{Vis } e k} \quad \frac{(\forall x, \exists y, h x \sim k y) \wedge (\forall y, \exists x, h x \sim k y)}{\text{Br}^b h \sim \text{Br}^c k} \\
968 \\
969 \\
970 \quad \frac{t \sim u}{\text{Step } t \sim \text{Step } u} \quad \frac{(\forall x, \exists y, h x \sim k y) \wedge (\forall y, \exists x, h x \sim k y)}{\text{Br}_S^b h \sim \text{Br}_S^c k} \\
971 \\
972 \\
973 \\
974 \quad \frac{t \sim u \quad (\forall x, g x \sim k x)}{t \gg g \sim u \gg k} \\
975 \\
976 \\
977 \quad \text{Guard } t \sim t \quad \frac{u \leftrightarrow}{\text{Br}^2 t u \sim t} \quad \text{Br}^2 t (\text{Br}^2 u v) \sim \text{Br}^2 (\text{Br}^2 t u) v \quad \text{Br}^2 t u \sim \text{Br}^2 u t \\
978 \\
979 \\
980 \quad \text{Br}^2 t t \sim t \quad \text{Br}^2 (\text{Br}^2 t u) v \sim \text{Br}^3 t u v \quad \text{Br}_S^2 t u \sim \text{Br}_S^2 u t \quad \text{Br}_S^2 t t \sim \text{Step } t \\
981 \\
982 \quad \text{Step } t \not\sim t \quad \text{stagnate\_nary } n \sim \text{stagnate\_nary } m \\
983 \\
984 \quad \frac{(n > 0 \wedge m > 0) \vee (n = m = 0)}{\text{spinS\_nary } n \sim \text{spinS\_nary } m} \\
985 \\
986 \\
987 \\
988 \\
989 \\
990
\end{array}$$

Fig. 18: Elementary equational theory for CTrees (👉)

991 The first four rules recover some structural reasoning on the syntax of the trees from  
992 its semantic interpretation. These rules recover reasoning principles close to what eutt  
993 provides by construction for ITrees: leaves are bisimilar if they are equal, computations  
994 performing the same external interaction must remain point-wise bisimilar, and unary steps  
995 can be matched against one another. Delayed branches, potentially of distinct arity, can be  
996 matched one against another if both domains of indexes can be injected into the other to  
997 reestablish bisimilarity. This is only an implication and not an equivalence since the points  
998 of the continuation structurally immediately accessible do not correspond to accessible  
999 states in the LTS. By contrast, this same condition is necessary and sufficient for *stepping*  
1000 branches, as their Step nodes induce additional  $\tau$  transitions, thus new states in the LTS.  
1001 Finally, bisimilarity is a congruence for `bind`.

1002 Unlike ITrees, two CTrees can be strongly bisimilar despite different head constructors.  
1003 The simplest example is that `sbisim` can ignore (finite numbers of) Guard nodes, making  
1004 e.g. Guard (Ret `tt`) bisimilar to Ret `tt`. Another example is that stuck processes behave as  
1005 a unit for delayed branching nodes. The fact that `Br` and Guard nodes are treated *weakly* by  
1006 *strong* bisimilarity can be slightly disconcerting, but our definition of *strong* bisimilarity is  
1007 standard. Rather, this mismatch is due to the fact that we completely collapse `Br` nodes in  
1008 the LTSs we build from CTrees. This choice has far-reaching consequences that Section 9  
1009 will further discuss.

1010 We obtain the equational theory that we expect for nondeterministic effects. Delayed  
1011 branching is associative, commutative, idempotent, and can be merged into delayed  
1012

branching nodes of larger arity w.r.t. `sbisim`.<sup>13</sup> By contrast, stepping branches are only commutative, and almost idempotent, provided we introduce an additional `Step`. This `Step` cannot be ignored w.r.t.  $\sim$ , but one can move to weak bisimilarity to this end: we discuss this setup in Section 10.3.

Finally, two stagnating processes 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 here, but we additionally prove that the `iter` combinator deserves its name: the Kleisli category of the `ctree E B` monad is iterative w.r.t. strong bisimulation ( $\Rightarrow$ ). Concretely, we prove that the four equations described in Section 4 of Xia et al. (2019) 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`. One could provide an alternate iterator guarding recursion by a `Step`, and recover the iterative laws w.r.t. weak bisimulation, but we 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 10.3.

**Example: `ImpBr`.** Naturally, this equational theory gets trivially lifted at the language level for `ImpBr`. More specifically, the equational theory over `CTrees` is lifted to the syntax through `repr_imp` ( $\Rightarrow$ ). Furthermore, anticipating on Section 7, the theory will be transported for free by interpretation, and hence hold on the full model  $\llbracket \cdot \rrbracket$  as well, in which we can additionally reason about state, and in particular establish the equation  $p_3 \equiv \text{br } p_2 \text{ or } p_3$  ( $\Rightarrow$ ) suggested in Section 2.2.

The acute reader may notice that in exchange for being able to work with strong bisimulation, we have mapped the silently looping program to `stagnate`, hence identifying it with stuck processes. Indeed, both of these programs have the same observable behavior (no observable transition). For an alternate model observing recursion, one would need to investigate the use of the alternate iterator mentioned above and work with weak bisimilarity.

### 5.3.2 Proof System for Bisimulation Proofs

As is usual, the laws in Figure 18, 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  $\mathcal{R}$  t u`, i.e., the proof goal in which  $\mathcal{R}$  is the bisimulation candidate, and this coinduction hypothesis is not yet accessible. We depict the main rules we use in Figure 19. From the standpoint of the proof scientist, these rules notably avoid the exponential explosion in the number of subgoals that would arise by playing each side the bisimulation game separately: for all `Ctree` constructors, there is a reasoning rule with no 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 between two cases. In the first case, applying the rule

<sup>13</sup> 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.

$$\begin{array}{c}
1059 \\
1060 \\
1061 \\
1062 \\
1063 \\
1064 \\
1065 \\
1066 \\
1067 \\
1068 \\
1069 \\
1070 \\
1071 \\
1072 \\
1073 \\
1074 \\
1075 \\
1076 \\
1077 \\
1078 \\
1079 \\
1080 \\
1081 \\
1082 \\
1083 \\
1084 \\
1085 \\
1086 \\
1087 \\
1088 \\
1089 \\
1090 \\
1091 \\
1092 \\
1093 \\
1094 \\
1095 \\
1096 \\
1097 \\
1098 \\
1099 \\
1100 \\
1101 \\
1102 \\
1103 \\
1104
\end{array}$$

$$\begin{array}{c}
\text{Ret } v \sim_{\mathcal{R}} \text{Ret } v \qquad \frac{\forall v, (k \ v) \ \mathcal{R} \ (k' \ v)}{\text{Vis } e \ k \sim_{\mathcal{R}} \text{Vis } e \ k'} \\
\\
\frac{(\forall x, \exists y, (k \ x) \sim_{\mathcal{R}} (k' \ y)) \wedge (\forall y, \exists x, (k \ x) \sim_{\mathcal{R}} (k' \ y))}{\text{Br}^b \ k \sim_{\mathcal{R}} \text{Br}^c \ k'} \qquad \frac{\forall v, (k \ v) \sim_{\mathcal{R}} (k' \ v)}{\text{Br}^b \ k \sim_{\mathcal{R}} \text{Br}^b \ k'} \\
\\
\frac{t \sim_{\mathcal{R}} u}{\text{Guard } t \sim_{\mathcal{R}} \text{Guard } u} \qquad \frac{t \ \mathcal{R} \ u}{\text{Step } t \sim_{\mathcal{R}} \text{Step } u} \\
\\
\frac{(\forall x, \exists y, (k \ x) \ \mathcal{R} \ (k' \ y)) \wedge (\forall y, \exists x, (k \ x) \ \mathcal{R} \ (k' \ y))}{\text{Br}_S^b \ k \sim_{\mathcal{R}} \text{Br}_S^c \ k'} \qquad \frac{\forall v, (k \ v) \ \mathcal{R} \ (k' \ v)}{\text{Br}_S^b \ k \sim_{\mathcal{R}} \text{Br}_S^b \ k'}
\end{array}$$

Fig. 19: Proof rules for coinductive proofs of  $\text{sbisim}$  (👉)

soundly acts as playing the game. i.e., the premises refer to  $\mathcal{R}$ , allowing to conclude using the coinduction hypothesis. In the second case, the premises still refer to  $\sim_{\mathcal{R}}$  and the proof rule strip off delayed branches from the structure of our trees. In this second case, on either side of the bisimulation, the rule does not entail any step in the corresponding LTSs, but rather corresponds to recursive calls to its inductive constructor.

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

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

In particular, the  $\text{UPTO}^{up}(\sim)$  rewriting principle can be used with the equation  $\text{Guard } t \sim t$  during bisimulation proofs, allowing for asymmetric stripping of guards.

We always strive to provide powerful up-to principles, but from a library user perspective  $\sim_{\mathcal{R}}$  is still harder to work with than  $\sim$ . All the results that are valid for  $\sim_{\mathcal{R}}$  are also valid for  $\sim$ , it is thus desirable to reason as much as possible at the  $\sim$  level. Proofs involving e.g. loops are by nature coinductive, but this layer of complexity can sometimes be abstracted away. Section 7 provides a few high-level proof principles for that purpose, in particular around the `interp` combinator.

We omit the details, but our library additionally provides a characterization of the traces of a CTree, defined as the set of colists<sup>14</sup> of labels induced by the LTS of the tree. Trace-equivalence (written  $\equiv_{\text{tr}}$ ) (👉) is hence defined as the extensional equality of these sets of traces. With this infrastructure, we prove the standard result that  $\text{sbisim}$  entails trace-equivalence (👉).

<sup>14</sup> Colists are coinductively defined possibly infinite lists.

### 5.4 Similarity

Similarity is a specific, well-studied, notion of refinement for LTSs. In particular, similarity entails behavior inclusion, making it a popular proof tool for verified compilation (a compiled program should be simulated by its corresponding source program), but also when reasoning about scheduling (a scheduler is correct when it shrinks the determinism induced in a process by the set of all valid schedulers).

There exists a variety of subtly distinct variants of similarity. We focus in this section on strong similarity, the relation defined as the greatest fixpoint of the left half-game of bisimulation as depicted on Figure 17.

**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:*

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

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

The coinductive proof rules for simulation are depicted in Figure 20. The rules are similar to those for bisimulation (Figure 19), 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 18. 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 20.

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

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

### 5.5 Proof Example

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

**VARIANT PrintE** : **Type** → **Type** := **print** : **bool** → **PrintE** **unit**.

**COFIXPOINT t** : **ctree** **PrintE** **B2** **void** :=

$$\begin{array}{c}
1156 \quad \frac{t \rightarrow}{t \lesssim_{\mathcal{R}} u} \text{ (ss\_stuck)} \quad \text{Ret } v \lesssim_{\mathcal{R}} \text{Ret } v \text{ (ss\_ret)} \quad \frac{\forall v, (k v) \mathcal{R} (k' v)}{\text{Vis } e k \lesssim_{\mathcal{R}} \text{Vis } e k'} \text{ (ss\_vis\_id)} \\
1157 \\
1158 \\
1159 \quad \frac{\forall x, (k x) \lesssim_{\mathcal{R}} u}{\text{Br}^b k \lesssim_{\mathcal{R}} u} \text{ (ss\_br\_l)} \quad \frac{\exists y, t \lesssim_{\mathcal{R}} (k' y)}{t \lesssim_{\mathcal{R}} \text{Br}^b k'} \text{ (ss\_br\_r)} \\
1160 \\
1161 \\
1162 \quad \frac{\forall x, \exists y, (k x) \lesssim_{\mathcal{R}} (k' y)}{\text{Br}^b k \lesssim_{\mathcal{R}} \text{Br}^c k'} \text{ (ss\_br)} \quad \frac{\forall v, (k v) \lesssim_{\mathcal{R}} (k' v)}{\text{Br}^b k \lesssim_{\mathcal{R}} \text{Br}^b k'} \text{ (ss\_br\_id)} \\
1163 \\
1164 \\
1165 \\
1166 \quad \frac{t \lesssim_{\mathcal{R}} u}{\text{Guard } t \lesssim_{\mathcal{R}} u} \text{ (ss\_guard\_l)} \quad \frac{t \lesssim_{\mathcal{R}} u}{t \lesssim_{\mathcal{R}} \text{Guard } u} \text{ (ss\_guard\_r)} \\
1167 \\
1168 \\
1169 \quad \frac{t \lesssim_{\mathcal{R}} u}{\text{Guard } t \lesssim_{\mathcal{R}} \text{Guard } u} \text{ (ss\_guard)} \quad \frac{t \mathcal{R} u}{\text{Step } t \lesssim_{\mathcal{R}} \text{Step } u} \text{ (ss\_step)} \\
1170 \\
1171 \\
1172 \quad \frac{\forall x, \exists y, (k x) \mathcal{R} (k' y)}{\text{Br}_S^b k \lesssim_{\mathcal{R}} \text{Br}_S^c k'} \text{ (ss\_brS)} \quad \frac{\forall v, (k v) \mathcal{R} (k' v)}{\text{Br}_S^b k \lesssim_{\mathcal{R}} \text{Br}_S^b k'} \text{ (ss\_brS\_id)} \\
1173 \\
1174 \\
1175
\end{array}$$

Fig. 20: Rules for coinductive proofs of `ssim`, with their names in our Rocq library – rules with a double bar are equivalences (👉)

```

1176
1177
1178
1179     Vis (print true) (λ _ =>
1180       Vis (print false) (λ _ => t)).
1181
1182 CoFixpoint u : ctree PrintE B2 void :=
1183   Br2
1184     (Vis (print true) (λ _ => u))
1185     (Vis (print false) (λ _ => u)).
1186
1187 CoFixpoint u' : ctree PrintE B2 void :=
1188   Br2
1189     (trigger (print true))
1190     (trigger (print false))
1191   ;;
1192   Guard u'.
1193
1194
1195
1196
1197
1198
1199
1200
1201

```

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` nondeterministically 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 Rocq needs a syntactic guard before co-recursive calls. Because of this `Guard`, `u` and `u'` are not coinductively equal, but this has no consequence on the underlying LTS: they are still in strong bisimulation. The LTS underlying `t` and `u` are represented in Figure 21.

Fig. 21: The LTS for  $t$  (left) and  $u / u'$  (right)

It is clear that program  $u$  simulates program  $t$ . We establish this fact,  $t \lesssim u$ , through the following detailed proof steps, which correspond exactly to the Rocq 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.

$$\begin{array}{l}
 1215 \quad \vdash t \lesssim u \\
 1216 \quad \xleftarrow{\text{coinduction}} t \mathcal{R} u \vdash t \lesssim_{\mathcal{R}} u \\
 1217 \\
 1218 \quad \xleftarrow{\text{unfold}} t \mathcal{R} u \vdash \text{Vis} (\text{print true}) (\lambda \_ \Rightarrow \text{Vis} (\text{print false}) (\lambda \_ \Rightarrow t)) \lesssim_{\mathcal{R}} \\
 1219 \quad \quad \text{br branch2} (\lambda b \Rightarrow \text{Vis} (\text{print } b) (\lambda \_ \Rightarrow u)) \\
 1220 \\
 1221 \quad \xleftarrow{\text{ss\_br\_r true}} t \mathcal{R} u \vdash \text{Vis} (\text{print true}) (\lambda \_ \Rightarrow \text{Vis} (\text{print false}) (\lambda \_ \Rightarrow t)) \lesssim_{\mathcal{R}} \\
 1222 \quad \quad \text{Vis} (\text{print true}) (\lambda \_ \Rightarrow u) \\
 1223 \\
 1224 \quad \xleftarrow{\text{ss\_vis\_id}} t \mathcal{R} u \vdash \text{Vis} (\text{print false}) (\lambda \_ \Rightarrow t) \mathcal{R} u \\
 1225 \\
 1226 \quad \xleftarrow{\text{step}} t \mathcal{R} u \vdash \text{Vis} (\text{print false}) (\lambda \_ \Rightarrow t) \lesssim_{\mathcal{R}} u \\
 1227 \\
 1228 \quad \xleftarrow{\text{unfold}} t \mathcal{R} u \vdash \text{Vis} (\text{print false}) (\lambda \_ \Rightarrow t) \lesssim_{\mathcal{R}} \\
 1229 \quad \quad \text{br branch2} (\lambda b \Rightarrow \text{Vis} (\text{print } b) (\lambda \_ \Rightarrow u)) \\
 1230 \\
 1231 \quad \xleftarrow{\text{ss\_br\_r false}} t \mathcal{R} u \vdash \text{Vis} (\text{print false}) (\lambda \_ \Rightarrow t) \lesssim_{\mathcal{R}} \\
 1232 \quad \quad \text{Vis} (\text{print false}) (\lambda \_ \Rightarrow u) \\
 1233 \\
 1234 \quad \xleftarrow{\text{ss\_vis\_id}} t \mathcal{R} u \vdash t \mathcal{R} u
 \end{array}$$

The proof proceeds by coinduction: taking the singleton pair  $(t, u)$  as the simulation candidate  $\mathcal{R}$ ,<sup>15</sup> 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  $u$ .<sup>16</sup> 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 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  $\text{Vis} (\text{print true})$  can be stripped using the `ss_vis_id` theorem. Notice that the relation in the goal is no longer  $\lesssim_{\mathcal{R}}$  but

<sup>15</sup> Note that this candidate would not be a valid simulation without any enhancement.

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

1248  $\mathcal{R}$ , as consuming  $\text{Vis}$  nodes performs a simulation step. But we want to perform one more  
 1249 simulation step to consume the second  $\text{Vis}$  of the left  $\text{CTree}$ , so we strengthen the proof  
 1250 goal from  $\mathcal{R}$  to  $\lesssim_{\mathcal{R}}$  again using the  $ss$  up-to  $ss$  principle through the  $\text{step}$  tactic. Then, the  
 1251 remaining  $\text{Vis}$  on the left can be matched using the same method as the first one, finally  
 1252 reducing to the goal  $\text{t } \mathcal{R} u$ , which is precisely our coinduction hypothesis, concluding the  
 1253 proof.

1254 This simple example emphasizes that our infrastructure is robust enough to formally  
 1255 conduct, in Rocq, a proof that closely mimics the detailed one we would write on paper.

1256 For a similar pedagogical proof of bisimilarity of  $u$  and  $u'$ , illustrating the use of  
 1257 bisimulation up-to bisimilarity, we refer the interested reader to our development (👉).

## 1261 6 Case Study: ccs, Equational Theory

1262 Recall the model for  $\text{ccs}$  introduced in Section 4: we show that the generic bisimilarity of  
 1263  $\text{CTrees}$  introduced in the previous section is directly suitable for this model. The results  
 1264 we obtain—the usual algebra, up-to principles, and precisely the same equivalence relation  
 1265 as the usual operational-based strong bisimulation—are standard, per se, but they are all  
 1266 established by exploiting the generic notion of bisimilarity of  $\text{CTrees}$ .  
 1267

### 1268 6.1 Equational Theory

1269 We provide a first validation of our model by proving that it satisfies the expected equational  
 1270 theory with respect to  $\text{CTrees}$ 's notion of strong bisimulation, enabling the usual algebraic  
 1271 reasoning advocated for process calculi. In particular, we prove that our definition for the  
 1272 replication is sane in that it validates equationally the expected definition:  $\bar{!}p \sim \bar{!}p \parallel p$  (👉).  
 1273 We also prove an illustrative collection of expected equations satisfied by our operators  
 1274 (👉):  
 1275

$$\begin{array}{l}
 1276 \quad p \bar{\vdash} q \sim q \bar{\vdash} p \qquad p \bar{\vdash} (q \bar{\vdash} r) \sim (p \bar{\vdash} q) \bar{\vdash} r \qquad p \bar{\vdash} \bar{0} \sim p \qquad p \bar{\vdash} p \sim p \\
 1277 \\
 1278 \quad p \parallel \bar{0} \sim p \qquad p \parallel q \sim q \parallel p \qquad p \parallel (q \parallel r) \sim (p \parallel q) \parallel r
 \end{array}$$

1279 To facilitate these proofs, we first prove sound up-to principles at the level of  $\text{ccs}$  for  
 1280 each constructor: strong bisimulation up-to  $c\text{-}[\cdot]$ ,  $[\cdot] \bar{\vdash} [\cdot]$ ,  $[\cdot] \parallel [\cdot]$ ,  $\bar{!}[\cdot]$ , and  $\nu c\text{-}[\cdot]$  are all  
 1281 valid principles, allowing us to rewrite  $\text{sbisim}$  under semantic contexts during bisimulation  
 1282 proofs. Additionally to these language-level up-to principles, we inherit the ones generically  
 1283 supported by  $\text{sbisim}$  (Lemma 3).  
 1284  
 1285  
 1286

### 1287 6.2 Equivalence with the Operational Strong Bisimilarity

1288 In addition to proving that we recover in our semantic domain the expected up-to principles  
 1289 and the right algebra, we furthermore show that the model is sound and complete with  
 1290 respect to strong bisimulation compared to its operational counterpart. We do so by first  
 1291 establishing an asymmetrical bisimulation between  $\text{ccs}$  and  $\text{ccs}^\#$ , matching operational  
 1292  
 1293

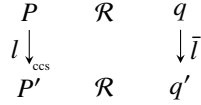


Fig. 22: The bisimulation game for ccs

steps over the syntax to semantic steps in the CTree. We write  $\bar{l}$  for the obvious translation of labels between both LTSs.

**Definition 4** (Strong bisimulation between ccs and ccs<sup>#</sup>).

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 ccs<sup>#</sup> tree  $q$ :

$$P \mathcal{R} q \wedge P \xrightarrow{l}_{\text{ccs}} P' \implies \exists q', P' \mathcal{R} q' \wedge q \xrightarrow{\bar{l}} q'$$

and conversely

$$P \mathcal{R} q \wedge q \xrightarrow{\bar{l}} q' \implies \exists P'. P' \mathcal{R} q' \wedge P \xrightarrow{l} P'$$

Figure 22 depicts this bisimulation game graphically.

**Lemma 5** (👉). 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 6** (Equivalence of ccs bisimulations (👉)).  $\forall P Q, \llbracket P \rrbracket \sim \llbracket Q \rrbracket$  iff  $P \sim_{\text{ccs}} Q$

This establishes CTrees as a fitting framework for reasoning about channel-based concurrency. We shall illustrate in Section 8 that they are flexible enough to model shared-memory-based concurrency as well.

## 7 Interpretation from and to CTrees

The ITree ecosystem fundamentally relies on the incremental interpretation of effects, represented as external events, into their monadic implementations. Section 3.2 has defined interpretation of CTrees, showing that they also fit into this narrative. Through this section, we go further down this path by developing the meta-theory of CTree interpretation (Section 7.1), and showing that they form a suitable target monad for ITrees, for the implementation of nondeterministic branching (Section 7.4). Section 7.2 studies the related CTree combinator `refine`, that can interpret `Br` branches of a CTree to reduce its nondeterminism. Finally, Section 7.3 discusses the extraction of CTrees, that enables their execution after their effects have been interpreted away.

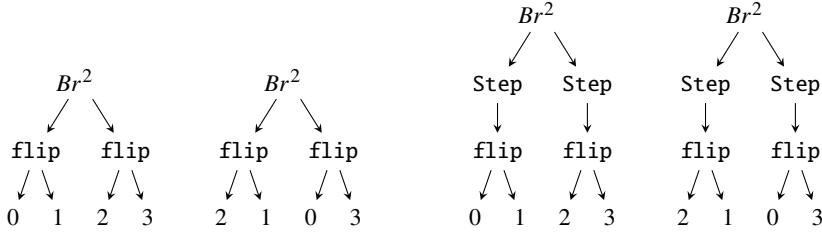


Fig. 23: Two strongly bisimilar trees before interpretation (left), but not after (right)

### 7.1 Interpretation

Section 3.2 introduced CTree interpretation. With the tools developed in Section 5, we can now turn to theoretical results around the `interp` combinator. Perhaps a bit 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  $M := \text{ctree } F \ B$  for some  $F$ . 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 23, where `flip` is the binary event introduced in Section 2.2, fleshes out this intuition. Indeed, the two trees on the left 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 as shown on the right in the figure. We hence prove that the desired property holds on a subclass of handlers.

**Definition 5** (Quasi-pure handlers). *Given signatures  $E, F, B$ , a handler  $h : E \rightsquigarrow \text{ctree } F \ B$  is said to be 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.,  $\forall l \ t', h \ e \xrightarrow{l} t' \implies \exists r, t' = \text{Ret } r$ .

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 monad morphisms when working with quasi-pure handlers:

**Theorem 1** (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  $h : E \rightsquigarrow \text{stateT } S (\text{ctree } F B)$  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 (👉). Bahr and Hutton (2023) have established that the restriction to quasi-pure handlers can be lifted by defining the LTS a bit differently (see Section 11). For the time being, we have decided against making this change in the `CTrees` library because it represents too much added complexity for little benefit.

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 2** (A simulation between handlers can be lifted through `interp` (👉)).

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

This theorem is presented here in the homogeneous case for simplicity, but we provide it for heterogeneous simulations, thus the heavily-parameterized `Rocq` statement of the mechanized theorem. Note that unlike Theorem 1, we do not need any assumption on the handlers or the trees.

## 7.2 Refinement

Interpretation provides a general theory for the implementation of external events. Importantly, `CTrees` also support an analogous facility for the implementation of its `Br` branches. In particular, one would typically use this process to shrink the set of accessible paths in a computation—and, notably, determinize it—hence leading to a new computation that refines by the original one. For this reason, we abusively refer to this process as the *refinement* of its `Br` branches, and capture sufficient conditions to ensure that it defines a computation simulated by the original one.

```

1432 Definition refine (h : B  $\rightsquigarrow$  M) : ctree E B  $\rightsquigarrow$  M :=  $\lambda$ R  $\Rightarrow$ 
1433 iter ( $\lambda$  t  $\Rightarrow$  match t with
1434 | Ret r  $\Rightarrow$  ret (inr r)
1435 | Stuck  $\Rightarrow$  mStuck
1436 | Guard t  $\Rightarrow$  ret (inl t)
1437 | Step t  $\Rightarrow$  bind mstep ( $\lambda$  _  $\Rightarrow$  inl t)
1438 | Br b k  $\Rightarrow$  bind (h b) ( $\lambda$  x  $\Rightarrow$  ret (inl (k x)))
1439 | Vis e k  $\Rightarrow$  bind (mTrigger e) ( $\lambda$  x  $\Rightarrow$  ret (inl (k x)))
1440 end .

```

Fig. 24: Refining CTrees (class constraints omitted) (🔗)

We provide, to this end, a new combinator, `refine`, defined in Figure 24. The definition is very similar<sup>17</sup> to `interp`, except that it takes as an argument a handler specifying how 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 by Yoon et al. (2022).

Similarly to `interp`, it can be desirable to write handlers that refine only part of the `Br` nodes. For instance, a CTree containing `Br` nodes for thread scheduling and other `Br` nodes for random number generation can be refined to schedule the threads in a round-robin way while not touching the `Br` nodes for random number generation. Note that this was not possible in the original CTrees library (Chappe et al., 2023) because there was no branching signature in the CTree type.

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 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 7** (Finite pure refinements are proper refinements (🔗)).

- $\forall e, h \text{ e finite pure} \implies \forall t, \text{refine } (M := \text{ctree } \_ \_) h t \lesssim t$
- $\forall e s, h \text{ e s finite pure} \implies \forall t s, \text{refine } (M := \text{stateT } \_ (\text{ctree } \_ \_)) h t s \lesssim_{st} t$

Note that the second statement introduces a heterogeneous simulation (see Section 10.2): the return types of the trees are not identical — the refined tree maintains an additional state that we ignore.

Unlike `interp`, `refine` is typically *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 respectively 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 nondeterminism, with branches indistinguishable for `sbisim`.

<sup>17</sup> 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 nondeterministic branches.

---

```

1478 let rec run t =
1479   match observe t with
1480   | RetF r -> print_int (int_of_nat r); true
1481   | BrF (_, k) ->
1482     let b = Random.bool() in
1483     if run (k (Obj.magic b)) then true
1484     else run (k (Obj.magic (not b)))
1485   | GuardF t -> run t
1486   | StuckF -> false
1487   | _ -> failwith "unreachable";;
1488
1489 let rec collect t =
1490   match observe t with
1491   | RetF r -> [int_of_nat r]
1492   | GuardF t -> collect t
1493   | StuckF -> []
1494   | BrF (_, k) ->
1495     collect (k (Obj.magic true)) @ collect (k (Obj.magic false))
1496   | _ -> failwith "unreachable";;

```

---

Fig. 25: A random interpreter and a collecting interpreter for ImpBr, implemented in OCaml.

### 7.3 Extraction

The shallow nature of CTrees also offers testing opportunities. Xia et al. (2019) 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 the locally angelic nondeterminism of *Br* nodes. Indeed, the naïve approach (🚫) of randomly choosing a branch 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 `run` for the ImpBr example (👉) that backtracks when it encounters a stuck branch, as shown in Figure 25.

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 `collect` (see Figure 25) that, given an ImpBr program, crawls its CTree to build the list of its possible return values.

We observe that these interpreters exhibit 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 stagnate CTree from Figure 6). If it reaches a stagnate, 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

```

1524 Definition inject {E} : itree E  $\rightsquigarrow$  ctree E := interp ( $\lambda$  e  $\Rightarrow$  trigger e).
1525 Definition internalize {E} : ctree (C + E) B  $\rightsquigarrow$  ctree E (B + C) :=
1526   interp ( $\lambda$  e  $\Rightarrow$  match e with | inl1 b  $\Rightarrow$  branchS b | inr1 e  $\Rightarrow$  trigger e).
1527 Definition embed {E} : itree (Choose + E)  $\rightsquigarrow$  ctree E :=
1528    $\lambda$  _ t  $\Rightarrow$  internalize (inject t).

```

Fig. 26: Implementing external branching events into the CTree monad (👉)

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.

### 7.4 ITree Embedding

We have used CTrees directly as a domain to represent the syntax of ImpBr, as well as in the ccs case study (see Section 4). CTrees can, however, fulfil 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 C and aim to define an operator `embed` taking an ITree computation modeling nondeterministic branching using these external events, and implementing them as branches indexed similarly into a CTree. This operator, defined in Figure 26, 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 (because of the Guard-based definition of `iter` on CTrees) `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 CTrees implementing a C event, using the isomorphic stepping branch. The resulting embedding forms a monad morphism transporting `eutt` ITrees into `sbisim` CTrees:

**Lemma 8** (`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 CTrees perspective, it is therefore natural to think of recursing in `euttF` as corresponding to the inductive case in the definition of the LTS: in this sense, `Stepi` nodes behave as `Guard` nodes. Semantically however, restricting ourselves to the deterministic case, i.e., where a CTree contains no `Br` nodes other than `Stuck` and `Guard`, the process of substituting `Guard` nodes for `Step` nodes lead to a strongly bisimilar computation.

Conversely, the corecursive case in `eutt` corresponds to both LTSs taking a synchronous internal step: we are tempted to think of  $\text{Step}_i^\omega$  as corresponding to  $\text{Step}^\omega$ . However, here again, we have actually more lenience. Indeed, ITrees do not have a notion of stuck computation,<sup>18</sup> contrary to CTrees: it turns out that it is just as sound, in the sense of Lemma 8, to map  $\text{Step}_i^\omega$  to  $\text{Guard}^\omega$ , i.e., to *Stuck*.

In the embedding we present here, we have taken this latter choice: to see this, one needs to unfold a couple of definitions. The `internalize` function relies on an `interp` that consumes ITrees and produce CTrees. The behavior of this `interp`, when faced with a  $\text{Step}_i$ , is to consume it and recurse w.r.t. the notion of iteration provided by CTrees: as introduced in Figure 5, this recursion is guarded by `Guard`.

In the proof of Lemma 8, the choice of embedding  $\text{Step}_i$  into `Guard` materializes by the fact that an induction on `euttF` leaves us disappointed in the symmetric  $\text{Step}_i$  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  $\text{Step}_i^\omega$ , i.e., that we can inductively reach a `Vis` or a `Ret` node (☞).

**Limitations: on guarding recursive calls using `Guard`.** We have shown that CTrees 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 nondeterministic effects represented as external in an ITree.

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

Instead, a valuable avenue would be to develop the theory accompanying the alternate iterator mentioned in Section 5.3 and guarding recursion using `Step`. Naturally, the corresponding monad would not be iterative with respect to strong bisimulation, but we conjecture that it would be against weak bisimulation. From this alternate iterator would arise an alternative embedding of ITrees into CTrees: we conjecture it would still respect `eutt`, but seen as a morphism into CTrees equipped with weak bisimulation. The development accompanying this paper does not yet support this alternate iterator, we leave its implementation for future work.

<sup>18</sup> It can typically be encoded as a nullary event, but this event is observable: such an encoding of stuck does not refine an arbitrary computation.

1616 Currently, the user has the choice between (1) not observing recursion at all, but getting  
 1617 away with strong bisimulation in exchange, (2) manually inserting `Step` at recursive calls  
 1618 that they chose to observe. With support for this alternate iterator, the user would be given  
 1619 additional option to (3) systematically  $\tau$ -observe recursion, at the cost of working with  
 1620 weak bisimulation everywhere.

## 1622 8 Case Study: Modelling Cooperative Multithreading

1624 Through the previous sections, we have used CCS to illustrate the use of CTrees. As a  
 1625 second case study, we now consider cooperative multithreading. Cooperative scheduling is  
 1626 used in languages such as Javascript, async Rust, or Akka/Scala actors, but is also a very  
 1627 general model, as preemptive multi-tasking is equivalent to cooperative scheduling where  
 1628 threads are willing to yield (i.e., let other threads run) at any time.

1629 We extend the syntax of `imp` with two additional constructs:

$$1630 \text{ comm} \triangleq \text{skip} \mid x ::= e \mid c1; c2 \mid \text{while } b \text{ do } c \mid \text{fork } c1 \ c2 \mid \text{yield}$$

1632 The command `fork c1 c2` forks the current thread into two, running respectively  $c1$  and  
 1633  $c2$ . Importantly, the thread running  $c2$  retains control, and the one running  $c1$  will only  
 1634 get a chance to execute *after*  $c2$  voluntarily yields control, or terminates. This yielding of  
 1635 control is achieved by the `yield` statement, which signals that the current thread suspends  
 1636 its execution, and lets a new thread be scheduled—possibly the same one again.

1637 This semantics implements a mechanism akin to the `fork` system call, which duplicates  
 1638 the current process, but without a possibility for joining threads. For example, the program

$$1639 (\text{fork } (x ::= 1) (\text{yield}; x ::= 2)); y ::= x$$

1641 forks two copies of the program, with the “main” thread immediately yielding, allowing  
 1642 for either thread to run next. Its semantics is to first spawn a thread for  $x ::= 1$ , then have  
 1643 the main thread reach the `yield`, giving  $x ::= 1$  a chance to run. Assuming the spawned  
 1644 thread goes next, it runs in sequence  $x ::= 1$  and  $y ::= x$ , after which the main thread recovers  
 1645 control and finishes its execution.  $y ::= x$  is part of both threads and is thus executed twice,  
 1646 after each assignment to  $x$ .

1647 Alternatively, some cooperative scheduling languages (Abadi and Plotkin, 2010) consider  
 1648 a `spawn` operator that simply spawns an independent thread: we can encode this behavior  
 1649 in several ways. Notably, if `spawn` always occurs in tail position, there is no continuation  
 1650 to duplicate. For instance, the program

$$1651 \text{fork } (x ::= 1) (\text{fork } (x ::= 2) \text{skip})$$

1653 spawns two threads that set  $x$  to different values, and terminates. The two spawned threads  
 1654 can then be scheduled in either order, resulting in  $x = 1$  or  $x = 2$  in the final state. This  
 1655 constraint could be syntactically enforced in the language if relevant. Alternatively, one  
 1656 can use the command `while true do yield`<sup>19</sup> to “terminate” a thread; for instance to  
 1657 prevent the first thread above from reaching  $y ::= x$ . With fancier encodings using reserved  
 1658 shared-variables, nested “joins” and other synchronization operations can be modeled. In

1659 <sup>19</sup> Or more elegantly, introduce `block` in the language.  
 1660  
 1661

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 + ForkE + MemE) voidB unit`. At this stage of representation, they are modeled as *deterministic* computations,<sup>20</sup> as highlighted by the use of the empty interface `voidB` for branches. The computation can however perform events from two additional signature, namely yielding and forking:

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

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

The remaining of the representation is entirely standard. We write  $\llbracket p \rrbracket$  the representation of  $p$ .

**Interleaving.** The model’s second pass makes explicit the nondeterminism implicitly induced by the `Fork` events (extending the thread pool) and the `Yield` events (nondeterministically 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.`

<sup>20</sup> We could have alternatively used `ITrees` as semantic domain at this stage.

```

1708   schedule v0 [-] ≐ ret ()
1709   schedule vn [-] ≐ Vis Yield (branchn (λi · schedule vn [i]))   for n > 0
1710   schedule vn [i] ≐
1711     if v[i] =      then
1712       ret ()      Guard (schedule v[-i]n-1 [-])
1713       Guard t     Guard (schedule v[i ↦ t]n [i])
1714       Step t      Step (schedule v[i ↦ t]n [i])
1715       Stuck       Stuck
1716   Vis Yield k     Guard (schedule v[i ↦ k] ()n [-])
1717   Vis Fork k      Vis Spawn (λ_ · schedule (k true :: v[i ↦ k false])n+1 [i + 1])
1718   Vis e k         Vis e (λx · schedule (v[i ↦ k x])n [i])

```

Fig. 27: 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 27 defines the function formally.<sup>21</sup> 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. The remaining cases correspond to events and in particular

<sup>21</sup> The implementation relies on Sozeau and Mangin (2019)'s `Equations` library given the heavy reliance on dependent pattern matching on vectors.

concurrency events.<sup>22</sup> 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 stays 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. However, they offer an intermediate semantic domain at which less programs are equated, but where there are more contexts inside which program equivalence is preserved: intuitively, equivalent programs are known to yield at the exact same points. We illustrate on an example the distinction between both equivalences in Section 8.2; leveraging this intermediate model to strengthen the language’s equational theory is left for future work.

We write  $\mathcal{S}[[p]] \triangleq \text{schedule } [[p]]_1 \lfloor - \rfloor : \text{ctree } (\text{YieldE} + \text{SpawnE} + \text{MemE}) \text{ unit}$  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`.

**Scheduling and Extraction.** Yielding models execution points where the control is handed back to the scheduler, that gets to pick the next thread. The nondeterminism in the model accounts for the fact that we are at a level of abstraction where we do not have any control on the scheduler: programs should be safe against an arbitrary scheduler. We do not go further in this case study, but we observe that the `refine` operator studied in Section 7.2 can be used to reduce the set of supported schedulers, and in particular to provide a deterministic one, resulting by extraction into a deterministic executable interpreter. Alternatively, one can extract the model as is, and implement a scheduler directly in OCaml. We refer to (Chappe et al., 2025) for a larger case study on modelling shared memory concurrency with `CTrees` that notably features the implementation of simple schedulers and the extraction of the model for testing purpose.

## 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 when `yield` behaves as a `noop`. Indeed, when

<sup>22</sup> Note that the typing of `thread` rules out statically the `Br` case, which simplifies greatly the proof of the meta-theory of `schedule`.

another thread is running, in the program in the middle `yield` lets the other thread run first, whereas the right program immediately runs `c` and only lets the other thread run afterwards. 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. Indeed when scheduling events are observable, only programs that allow thread interleaving to occur at the same time can be considered equivalent.

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

**Lemma 9** (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  $\rho$ , then  $\text{schedule } v_n [i] \sim \text{schedule } w_n [\rho i]$ .*

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.

Lemma 9 allows us to permute the thread pool, which is useful in examples such as (👉):

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

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 9 allows us to validate some simple optimizations that do not directly involve reasoning about concurrency or memory, such as (👉):

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

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 9 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 7.1, which allows us to transport equations made before interpreting state events into computations in the state monad after interpretation.

## 9 Reasoning on a Br-aware LTS

In some cases, (bi)simulation proofs that involve *Br* nodes can be unwieldy. Recall from Figure 6 that `stagnate` is defined as an infinite chain of `Guard` nodes. Since it can take no transition, it is semantically equivalent to  $\emptyset$ , and one may therefore consider proving that

it refines any computation:  $\forall t, \text{stagnate} \lesssim t$ . In order to prove this property, we could of course simply apply the rule `ss_stuck`, but in order to build intuition about the problem at hand, let us try to proceed by coinduction, using the rules from Figure 20.

$$\vdash \forall t, \text{stagnate} \lesssim t$$

$$\xleftarrow{\text{coinduction}} \forall t, \text{stagnate} \mathcal{R} t \vdash \forall t, \text{stagnate} \lesssim_{\mathcal{R}} t$$

$$\xleftarrow{\text{unfold}} \forall t, \text{stagnate} \mathcal{R} t \vdash \forall t, \text{Guard } \text{stagnate} \lesssim_{\mathcal{R}} t$$

$$\xleftarrow{\text{ss\_guard\_1}} \forall t, \text{stagnate} \mathcal{R} t \vdash \forall t, \text{stagnate} \lesssim_{\mathcal{R}} t$$

After initializing the coinduction, we can unfold one iteration of `stagnate` and use the relevant proof rule to remove the `Guard`, leading back to the original state. However, this process *did not* give access to the coinduction hypothesis: similarity asks of us to draw a transition in the LTS! Rather, in the middle of our game, we need to proceed by induction over the transition considered—in this case to realize there is no such transition, but more generally to capture enough information on the states that can be reached by a transition.

This need for inductive proofs is not surprising, as the definition for the transition relation (Figure 15) is inductive itself. While in this simple example, 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 7. In particular, the proof of Theorem 1 was very convoluted in Chappe et al. (2023), and was lacking in generality. The current version of the proof is much simpler and more general thanks to the theory that will be developed in the present section.

This need for nested coinductive and inductive reasoning is actually quite common in the realm of weak (bi)similarity defined as strong bisimilarity on weak transition systems (van Glabbeek, 1993; Sangiorgi, 2012). Consider the standard case of weak bisimilarity: a natural definition is to state it as strong bisimilarity over the weak LTS where  $a \xrightarrow{l} b$  is defined as  $a \xrightarrow{\tau^*} \xrightarrow{l} \xrightarrow{\tau^*} b$  for  $l \neq \tau$ . The challenge quantifying on inductively reachable observable transition leads to a similar proliferation of undesired inductions. In this case, the standard solution is to observe that the same relation can be generated by an asymmetric game over the strong LTS—a solution which we indeed follow ourselves in our definition of weak bisimilarity of CTrees (see Section 10.3).

We follow a similar intuition, albeit adapted to our non-standard setup. In Section 9.1, we first introduce a straightforward alternative characterization of strong similarity of CTrees. We then do the same for strong bisimilarity in Section 9.2: this characterization turns out more challenging, requiring a definition by mutual coinduction that we dub *intertwined bisimilarity*.

### 9.1 Alternative Characterization of CTree Similarity

Our definition of strong bisimilarity is based on a completely standard strong simulation game (see Definition 3), and we have established convenient proof rules and up-to principles over it. However, it is defined on an LTS that bears some distance to the CTrees data-structure

$$\begin{array}{c}
 1892 \qquad \text{label } \triangleq \epsilon \mid \text{tau} \mid \text{obs } e \ v \mid \text{val } v \\
 1893 \\
 1894 \\
 1895 \quad \frac{}{Br^b \ k \xrightarrow{\epsilon} k \ v} \quad \frac{}{\text{Guard } t \xrightarrow{\epsilon} t} \quad \frac{}{\text{Step } t \xrightarrow{\text{tau}} t} \quad \frac{}{Vis \ e \ k \xrightarrow{\text{obs } e \ v} k \ v} \\
 1896 \\
 1897 \\
 1898 \qquad \frac{}{\text{Ret } v \xrightarrow{\text{val } v} \emptyset} \\
 1899
 \end{array}$$

Fig. 28: Inductive characterization of the alternative  $\epsilon$ -LTS induced by a CTree. The *Br* and *Guard* cases differ from the original LTS.

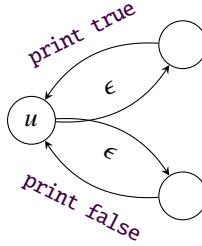


Fig. 29: The LTS built from the CTree *u* of Section 5.5, with explicit  $\epsilon$  transitions

itself: 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. In this section, we alternatively consider LTSs in which *Br* nodes generate a special  $\epsilon$  transition, and define a fitting simulation game that “ignores” these  $\epsilon$  transitions.

In the following, we refer to this new LTS as the  $\epsilon$ -LTS, or explicit LTS, and to the original one as the implicit LTS. Figure 28 shows the definition of the  $\epsilon$ -LTS: it is no longer inductive, each syntactic node in the CTree maps to transitions to its children in the LTS.<sup>23</sup> For instance, Figure 29 depicts the explicit LTS for the example CTree *u* from Section 5.5. We observe that given a CTree *c*, it admits an implicit transition  $c \xrightarrow{l} c'$  if and only if it admits an explicit sequence of transitions  $c \xrightarrow{\epsilon^*.l} c'$ , where  $\xrightarrow{\epsilon^*.l}$  is a shorthand for  $\xrightarrow{\epsilon^*} \xrightarrow{l}$ .

Working over this  $\epsilon$ -LTS, one must be careful about terminology: we seek to redefine *strong* similarity (meaning, strong w.r.t.  $\tau$  transitions), but will treat  $\epsilon$  transitions *weakly* in doing so. Figure 30 recalls the simulation game *ss* introduced in Section 5.4, but spelled out from the perspective of the  $\epsilon$ -LTS. We write  $t \xrightarrow{\epsilon} u$  for  $\epsilon$  transitions and  $t \xrightarrow{l} u$  for other transitions. This simply makes it explicit that the simulation challenge we have been considering may involve an arbitrary number of  $\epsilon$  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

<sup>23</sup> For historical reasons, this LTS is not explicitly defined in our Rocq development, but rather baked in the definition of the alternative simulation game. Making it explicit would make many Rocq definitions described in this section cleaner, and closer to the definitions in this paper, but it has no consequence on the theory: we leave this refactoring for future work.

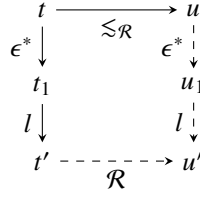


Fig. 30: The simulation game and bisimulation half-game  $t \lesssim_{\mathcal{R}} u$ , in the  $\epsilon$ -LTS

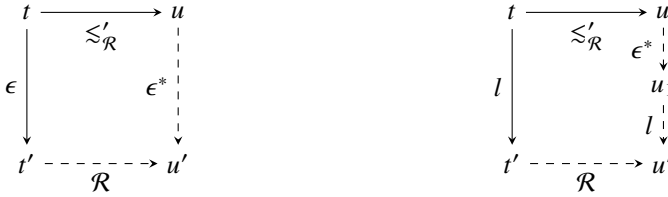


Fig. 31: The two cases of the simulation game  $t \lesssim'_{\mathcal{R}} u$

example proof above, the awkwardness in the game translates in the proof system: consider Figure 20, the proof rule for  $Br$  does not unlock the coinduction hypothesis.

We can now provide a definition of strong similarity of CTrees that is equivalent to  $ssim$ , while avoiding its drawbacks: Figure 31 depicts the corresponding game.

**Definition 6** (Simulation for CTrees on the explicit LTS (🔴)). *The progress function  $ss'$  for similarity over the explicit  $\epsilon$ -LTS maps a relation  $\mathcal{R}$  over CTrees to the relation such that  $ss' \mathcal{R} t u$  (also noted  $t \lesssim'_{\mathcal{R}} u$ ) holds if and only if:*

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

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

*Similarity, written  $t \lesssim' u$ , is defined as the greatest fixpoint of  $ss'$ :  $ssim' \triangleq \text{gfp } ss'$ .*

The key to the definition of this new game  $ss'$  is to distinguish apart  $\epsilon$ -transitions as a special kind of challenge. While other transitions are handled as in the original definitions,  $\epsilon$  transitions can be answered by any number of  $\epsilon$  transitions, possibly 0.

Our definition is very close to the standard notion of *weak simulation* Sangiorgi (2012). The definition of CTree simulation on the explicit LTS has two subtle differences: it is weak with respect to  $\epsilon$  transitions instead of  $\tau$ , and the answer to the non-epsilon challenge is different. Usually, classical weak simulation involves  $\xrightarrow{\epsilon^*.l.\epsilon^*}$ , but Definition 6 does not allow  $\epsilon$  transitions after the  $l$  transition. This is to match the semantics of the original CTree transition relation (Figure 15), which inductively skips  $Br$  nodes until it reaches a productive node, and stops just after. This slight difference is of no consequence on the greatest fixpoint:  $ssim'$  could equivalently be defined as the greatest fixpoint of either of these simulation games.

Crucially, we prove that both definitions of similarity coincide:

**Theorem 3** (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. But crucially, consuming a Guard node now unlocks the coinductive hypothesis, alleviating the need for a nested induction encountered when 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 `sb_guard` : `Guard t ~ t` would allow introducing a guard node and coinductively removing it using `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:

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

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

**Lemma 10** (Enhanced coinduction for  $\lesssim'$ ). *The functions `REFLup` (☞), `BINDup( $\lesssim$ )` (☞),  `$\epsilon_r^{up}$`  (☞), `ss` (☞) and `UPTOup( $\cong$ )` (☞) 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 11** ( $ss$  is a sub-chain of  $ss'$  (☞)). *Given  $\mathcal{R} : C_{ss'}$ , i.e., a chain for  $ss'$ , the following implication holds.*

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

With our alternative characterization of CTree simulation, we can now revisit our motivating example from the header of this section, and conclude via a coinductive proof (☞).

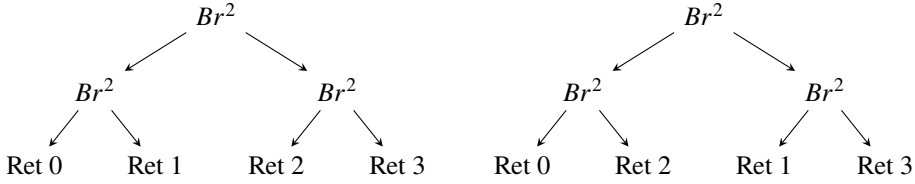


Fig. 32: Two bisimilar CTrees that motivate the definition of intertwined bisimilarity

$$\begin{aligned} & \vdash \forall t, \text{stagnate} \lesssim t \\ \Leftrightarrow & \vdash \forall t, \text{stagnate} \lesssim' t \end{aligned}$$

$$\xleftarrow{\text{coinduction}} \forall t, \text{stagnate } \mathcal{R} t \vdash \forall t, \text{stagnate} \lesssim'_{\mathcal{R}} t$$

$$\xleftarrow{\text{unfold}} \forall t, \text{stagnate } \mathcal{R} t \vdash \forall t, \text{Guard } \text{stagnate} \lesssim'_{\mathcal{R}} t$$

$$\xleftarrow{\text{ss\_guard\_1}} \forall t, \text{stagnate } \mathcal{R} t \vdash \forall t, \text{stagnate } \mathcal{R} t$$

### 9.2 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 inventiveness. Indeed, bisimulation games are usually defined as the intersection of a simulation game and its symmetrized version—that was the case for the bisimulation game in Section 5.3. However, this would not work for the game on the  $\epsilon$ -LTS. Consider the CTrees in Figure 32. 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  $\lesssim'$  (by intersecting two half games) would require an  $\epsilon$  challenge to be matched by *exactly* one  $\epsilon$  transition (as it would essentially amount to defining a standard weak bisimulation w.r.t.  $\epsilon$ ), which is not possible in 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 bisimulation over the  $\epsilon$ -LTS as 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. We dub this notion *intertwined bisimilarity*, to emphasize that it is an intersection of two interdependent simulation relations. This construction is reminiscent of *coupled simulations* (Parrow and Sjödin, 1994; Sangiorgi, 2012), which is defined as a pair of simulations and a coupling relation between them, though the principles are not comparable. To our knowledge, mutual coinduction has been used before as a key proof device in the context of applicative bisimilarity (Levy, 2006), but never in an actual definition of bisimulation.

To define formally intertwined 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 functions over binary CTree relations,  $\sim'_{\square}$  is a monotone function over ternary  $\text{bool} * \text{ctree} * \text{ctree}$  relations  $\mathcal{R}$ . In the following

we note  $\mathcal{R}_l$  for the left simulation  $\mathcal{R}$  true,  $\mathcal{R}_r$  for the right simulation  $\mathcal{R}$  false, and  $\mathcal{R}_{lr}$  for the intersection of  $\mathcal{R}_l$  and  $\mathcal{R}_r$ .

**Definition 7** (Bisimulation for CTrees on the explicit LTS ( $\bar{\square}$ )). *The progress function  $sb'$  for similarity over the explicit  $\epsilon$ -LTS maps a relation  $\mathcal{R}$  over a boolean and two CTrees to the relation such that*

- $sb' \mathcal{R} \text{ true } t u$  (also noted  $\sim'_{\mathcal{R}_l}$ ) holds if and only if:

$$\begin{aligned} \forall l t', l \neq \epsilon, t \xrightarrow{l} t' &\implies \exists u'. t' \mathcal{R}_{lr} u' \wedge u \xrightarrow{\epsilon^*.l} u' \\ \forall t', t \xrightarrow{\epsilon} t' &\implies \exists u'. t' \mathcal{R}_l u' \wedge u \xrightarrow{\epsilon^*} u' \end{aligned}$$

- $sb' \mathcal{R} \text{ false } t u$  (also noted  $\sim'_{\mathcal{R}_r}$ ) holds if and only if:

$$\begin{aligned} \forall u', l \neq \epsilon, u \xrightarrow{l} u' &\implies \exists t'. t' \mathcal{R}_{lr} u' \wedge t \xrightarrow{\epsilon^*.l} t' \\ \forall u', u \xrightarrow{\epsilon} u' &\implies \exists t'. t' \mathcal{R}_r u' \wedge t \xrightarrow{\epsilon^*} t' \end{aligned}$$

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.

This definition is similar to the one of  $\lesssim'_{\mathcal{R}}$  (Definition 6), but the  $\epsilon$  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  $\epsilon$  node, both the left and right half games need to be played (possibly several times) to get back to  $\mathcal{R}_l \cap \mathcal{R}_r$ .

Intuitively,  $t \sim'_{\mathcal{R}_l} u$  means that  $u$  can simulate one step of  $t$  (possibly skipping  $\epsilon$  transitions before reaching the matching transition), and the resulting  $t'$  and  $u'$  are bisimilar (in  $\sim'_{\mathcal{R}_{lr}}$ ). In case  $t$  admits an  $\epsilon$  transition, the  $u$  side can answer with any number of  $\epsilon$  transitions, or no transition at all. Symmetrically,  $t \sim'_{\mathcal{R}_r} u$  means that  $t$  can simulate one step of  $u$  (possibly skipping  $\epsilon$  transitions before reaching the matching transition), and the resulting  $t'$  and  $u'$  are bisimilar (in  $\sim'_{\mathcal{R}_{lr}}$ ). The name intertwined bisimulation comes from the fact that  $\mathcal{R}_l$  and  $\mathcal{R}_r$  can go separate ways when encountering  $\epsilon$  transitions, but converge back together when there is another transition.

For readers familiar with ITrees (Xia et al., 2019), 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, it is part of the coinduction domain and does evolve during a coinductive proof. Freely-stuttering simulation Cho et al. (2023) is another case of an extension of a coinductive domain in an ITree-adjacent setting that will be discussed in Section 11.

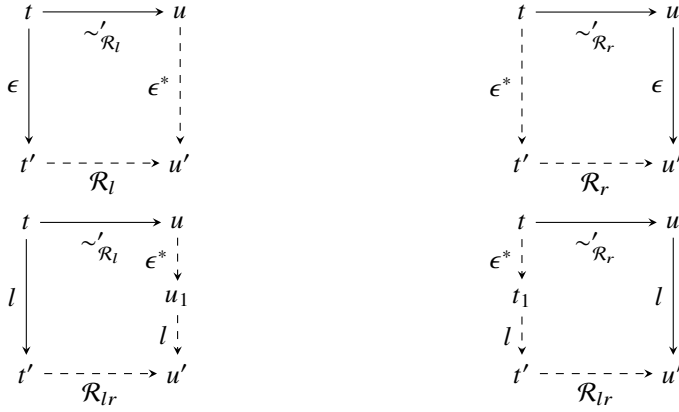


Fig. 33: The four cases of the intertwined bisimulation game  $t \sim'_{\mathcal{R}_{lr}} u$ . Note that the left and right half-games are symmetric, with  $\mathcal{R}_l$  and  $\mathcal{R}_r$  swapped.

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

Fig. 34: Proof rules for coinductive proofs of  $\sim'$  (🔴)

The main proof rules for intertwined bisimulation are shown in Figure 34.<sup>24</sup> 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  $\leq'$  in Lemma 10, we proved the validity of a ternary counterpart for  $\sim'$ .

**Lemma 12** (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 (🔴).*

<sup>24</sup> Note: for technical reasons, the rules involving  $\text{Br}_S$  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. 35: The two cases of the homogeneous alternative bisimulation game  $t \sim'_R u$ .  $\mathcal{R}^\circ$  represents the converse of  $\mathcal{R}$ .

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

The up-to negated symmetry principle is a ternary variant of the standard up-to symmetry principle. It swaps two CTree 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 5.3. Interestingly, the theorem statement can be split into a result on the left and right halves of  $\sim'$ .

**Theorem 4** (Equivalence of the two notions of bisimilarity (👉)).

$$\begin{aligned} \forall t \ u, \ t \sim u &\iff t \sim' u \\ \forall t \ u, \ ss \ (\text{sbisim } t \ u) &\iff \text{gfp } \sim'_\square \ \text{true } t \ u \\ \forall t \ u, \ ss \ (\text{sbisim } u \ t) &\iff \text{gfp } \sim'_\square \ \text{false } t \ u \end{aligned}$$

We have presented alternative characterizations of similarity and bisimilarity in the homogeneous case, but as with most definitions of Section 5, we implemented them in Rocq with support for heterogeneous relations (see Section 10.2). In fact, in the homogeneous case, the bisimulation game of Figure 33 degenerates to a simpler game (Figure 35) 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  $\mathcal{R} \ b \ t \ u \iff \mathcal{R} \ \neg b \ u \ t$  (👉).

## 10 More Notions of Equivalence and Refinement for CTrees

Section 5 extensively studied strong bisimilarity, strong similarity, and their equational theory. The aim of the present section is to demonstrate that the CTree data structure is not limited to these relatively simple notions, and that more involved notions of equivalence or refinement can be modeled in a similar way.

Section 10.1 defines a deadlock-sensitive notion of similarity, complete similarity. Section 10.2 shows how our notions of program comparison generalize to allow relating programs with different return types or signatures. Section 10.3 describes observations and preliminary results for weak bisimilarity. Numerous other equivalences or refinement relations exist in the literature (e.g., branching bisimilarity or stutter bisimulation); we believe most of them can easily be expressed over CTrees.



Fig. 36: The complete simulation game  $\lesssim_{\mathcal{R}}^C$

### 10.1 Complete Similarity

Section 5.4 introduced strong similarity, a standard notion of program refinement. This section studies complete similarity, a slightly more complex notion of similarity that behaves better in presence of stuck LTSs.

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 20.

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 36 depicts graphically the following complete simulation game. In these rules, we write  $t \rightarrow$  for  $\exists! t', t \xrightarrow{l} t'$ , i.e.  $t$  is not stuck.

**Definition 8** (Complete simulation for CTrees (🔴)). *The progress function  $css$  for complete similarity maps a relation  $\mathcal{R}$  over CTrees to the relation such that  $css \mathcal{R} t u$  (also noted  $t \lesssim_{\mathcal{R}}^C u$ ) holds if and only if*

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

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

The coinductive proof rules for complete simulation are depicted in Figure 37. 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. Cases that are identical to the rules for  $ssim$  are greyed out in the figure.

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.

$$\mathbf{BIND}_c^{up} (\text{EQUIV}) \ R \triangleq \{(x \ggg k, y \ggg l) \mid \text{EQUIV } x \ y \wedge \forall v, R (k \ v) (l \ v) \wedge k \ v \rightarrow\}$$

$$\begin{array}{c}
2260 \quad \text{Ret } v \lesssim_{\mathcal{R}}^C \text{Ret } v \\
2261 \\
2262 \\
2263 \quad \frac{\forall v, (k \ v) \ \mathcal{R} \ (k' \ v)}{\text{Vis } e \ k \lesssim_{\mathcal{R}}^C \text{Vis } e \ k'} \quad \frac{\forall x \in X, (k \ x) \lesssim_{\mathcal{R}}^C u \quad \text{inhabited } X}{\text{Br}^b \ k \lesssim_{\mathcal{R}}^C u} \quad \frac{\exists y, t \lesssim_{\mathcal{R}}^C (k' \ y) \quad t \rightarrow}{t \lesssim_{\mathcal{R}}^C \text{Br}^b \ k'} \\
2264 \\
2265 \\
2266 \quad \frac{\forall x, \exists y, (k \ x) \lesssim_{\mathcal{R}}^C (k' \ y) \quad \exists x, k \ x \rightarrow}{\text{Br}^b \ k \lesssim_{\mathcal{R}}^C \text{Br}^c \ k'} \quad \frac{\forall v, (k \ v) \lesssim_{\mathcal{R}}^C (k' \ v)}{\text{Br}^b \ k \lesssim_{\mathcal{R}}^C \text{Br}^b \ k'} \quad \frac{t \lesssim_{\mathcal{R}}^C u}{\text{Guard } t \lesssim_{\mathcal{R}}^C \text{Guard } u} \\
2267 \\
2268 \\
2269 \quad \frac{t \ \mathcal{R} \ u}{\text{Step } t \lesssim_{\mathcal{R}}^C \text{Step } u} \quad \frac{\forall x \in X, \exists y, (k \ x) \ \mathcal{R} \ (k' \ y) \quad \text{inhabited } X}{\text{Br}_S^b \ k \lesssim_{\mathcal{R}}^C \text{Br}_S^b \ k'} \quad \frac{\forall v, (k \ v) \ \mathcal{R} \ (k' \ v)}{\text{Br}_S^b \ k \lesssim_{\mathcal{R}}^C \text{Br}_S^b \ k'} \\
2270 \\
2271
\end{array}$$

Fig. 37: Proof rules for coinductive proofs of  $\text{cssim}$  (🔴)

**Lemma 13** (Enhanced coinduction for  $\text{cssim}$  (🔴)). *The functions  $\text{REFL}^{up}$ ,  $\text{TRANS}^{up}$ ,  $\text{BIND}_c^{up} (\lesssim^C)$  (🔴),  $\text{UPTO}^{up} (\cong)$  (🔴) and  $\text{UPTO}^{up} (\sim)$  (🔴) provide valid up-to principles for  $\text{cssim}$ .*

## 10.2 Heterogeneous (Bi)similarity

Throughout this paper, 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 10.3. From the perspective of program verification, this is insufficient:  $\text{val } v$  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. (2023) to introduce over `ITrees` the `rut` relation in order to express security invariants—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 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 9** (Bisimulation for `CTrees`, with arbitrary relation on labels (🔴)). *The progress function  $sb \ \mathcal{L}$  for bisimilarity maps a relation  $\mathcal{R}$  over `CTrees` to the relation such that  $sb \ \mathcal{L} \ R \ s \ t$  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  $sb \mathcal{L}$ :  $\text{sbisim } \mathcal{L} \triangleq \text{gfp}(sb \mathcal{L})$ .

Many of the proof rules, up-to principles, and theorems (in particular, Theorem 2) 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.

A class of  $\mathcal{L}$  relations is of particular interest: relations between labels that lift a relation  $\mathcal{V}$  of type  $X \rightarrow Y \rightarrow \text{Prop}$  between return values ( $\rightarrow$ ). Such a relation  $\uparrow \mathcal{V}$  of type  $\text{ctree } E B X \rightarrow \text{ctree } E B Y \rightarrow \text{Prop}$  coincides with equality for `obs` and `tau` labels, and relies on the  $\mathcal{V}$  relation for comparing `val` labels of possibly heterogeneous types. This is useful for comparing CTrees that are semantically related but typed differently.

### 10.3 Weak Bisimilarity

Weak bisimilarity is a bisimilarity relation that ignores silent steps (called  $\tau$  steps). It is an equivalence relation where programs are considered equivalent if they differ only in their number of finite  $\tau$  steps. Weak bisimilarity, written  $s \approx t$ , is derived from the definition of the weak transition ( $\rightarrow$ ). We will not detail our formal definition of weak bisimulation: it is based on the standard asymmetric game (see for instance Chapter 4 of (Sangiorgi, 2012)).

We first define the traditional *weak transition*  $s \xRightarrow{l} t$  on the LTS that can perform tau transitions before or after the  $l$  transition (and possibly no transition at all if  $l = \tau$ ). This part of the theory is so standard that we can directly reuse parts of the development for `ccs` that Pous developed to illustrate the companion (Pous, 2024). Weak bisimilarity enjoys additional equations compared to `sbisim` (See Figure 18): `Step` nodes can be ignored, and  $Br_S$  is therefore properly idempotent. Crucially however, it is still *not* associative.

$$\text{Step } t \approx t \quad Br_S^2 t t \approx t \quad Br_S^2 t (Br_S^2 u v) \not\approx Br_S^2 (Br_S^2 t u) v$$

Our library currently offers restricted support for weak bisimilarity. This situation stems in part by needs: existing applications of CTrees, including the examples in Sections 4 and 8, but also the artifact of Chappe et al. (2025) have so far only leveraged strong bisimilarity. While in the case of `ccs` it 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 definitions and careful definitions of the models, strong bisimilarity appears to be sufficient to recover a satisfying equivalence. This is partly explained by the fact that  $Br$

nodes exhibit a form of weak behavior w.r.t. strong bisimilarity. Section 9.1 will expand on this perspective.

But our limited support for weak bisimilarity also comes from interesting technical challenges. As is well-known in the field of process algebra (Milner, 1989, p.152), weak bisimilarity can be unwieldy. Specifically, it is not a congruence for the  $+$  operator of ccs (see Section 4) and of the  $\pi$ -calculus, 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 CTrees, imply the former result.

**Example 1** (Counter-example to the  $\approx$  up-to bind principle). *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  $t \gg k \approx t \gg k'$ .*

$$t := Br^2 (\text{Ret true}) (\text{Ret false})$$

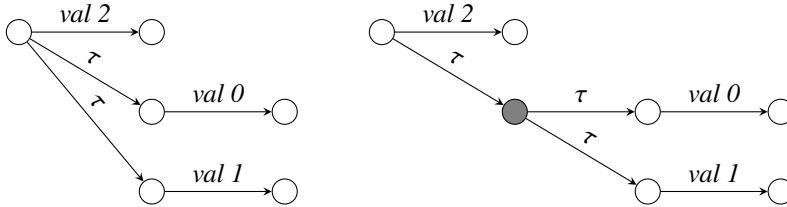
$$k \text{ true} := \text{Ret } 2$$

$$k \text{ false} := Br^2 (\text{Step} (\text{Ret } 0)) (\text{Step} (\text{Ret } 1))$$

$$k' \text{ true} := \text{Ret } 2$$

$$k' \text{ false} := \text{Step} (Br^2 (\text{Step} (\text{Ret } 0)) (\text{Step} (\text{Ret } 1)))$$

Below are depicted the LTSs for  $t \gg k$  (on the left) and  $t \gg k'$  (on the right). On the right, the grey state corresponds to the first Step of  $k' \text{ false}$ . It has no equivalent on the left, which explains why there is no bisimulation between these LTSs.



This limitation of weak bisimilarity is commonly circumvented in process algebra by requiring processes to be *guarded*:<sup>25</sup> the operands of the ccs  $+$  operator should begin with a deterministic transition. In a CTree setting, if we restrict ourselves to  $\tau$ -guarded parallel composition, 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 (👉).

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

<sup>25</sup> Beware that this use of “guarded” is not directly related to the notion of coinductive guard, nor to our Guard nodes.

This principle is however not completely satisfying: we can note that the hypothesis involves the plain relation  $\mathcal{R}$  (rather than  $\approx_{\mathcal{R}}$ ) and strips the leading `Step` nodes. Transferred to an enhanced principle, this would translate to a bind-rule for `Step`-guarded continuations that *does not* unlock the coinduction hypothesis. We leave the definition of a more comfortable proof principle to future work.

## 11 Related Work

Since Milner’s seminal work on `ccs` (Milner, 1989) and the  $\pi$ -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 `Rocq` realm, Ambal et al. (2020) have formalized `HO $\pi$` , a minimal  $\pi$ -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. The `CTrees` notion of “delayed choice” corresponds to `ccs`’s choice operator, and is named *mixed choice* in De Nicola (2014); it allows the encoding of both internal choice as a mixed choice where each branch starts by an invisible uncontrolled  $\tau$  transition; external choice (as found in `CSP` (Hoare, 1978)) is a mixed choice when each branch starts by a visible transition, but cannot be straightforwardly encoded for branches starting with  $\tau$  transitions. Encodings of the different types of choice and their properties and limitations is further discussed by van Glabbeek (1997).

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 Rocq. 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 Works adjacent to ITrees*

**Previous models of nondeterminism based on ITrees** In the Vellvm project (Zakowski et al., 2021), Zakowski et al. use non-deterministic events of an ITree for formalizing the nondeterministic features of the LLVM IR. They then interpret these events in order to reason on them. More specifically, their model consists of a propositionally-specified set of computations: ignoring other effects, the monad they use is  $\text{itree } E \_ \rightarrow \text{Prop}$ . The equivalence they build on top of it essentially amounts to a form of bijection up-to equivalence of the contained monadic computations. However, this approach suffers from several drawbacks. First, one of the monadic laws is broken: the `bind` operation does not associate to the left. Although stressed in the context of Vellvm (Zakowski et al., 2021) and Yoon et al.’s work on layered monadic interpreters (Yoon et al., 2022), this issue is not specific to ITrees but rather to a hypothetical “Prop Monad Transformer”, i.e. to  $\lambda M X \Rightarrow M X \rightarrow \text{Prop}$ , as pointed out previously in (Maillard et al., 2020). The definition is furthermore particularly difficult to work with. Indeed, the corresponding monadic equivalence is a form of bijection up-to setoid: for any trace in the source, we must existentially exhibit a suitable trace in the target. The inductive nature of this existential is problematic: one usually cannot exhibit upfront a coinductive object as witness, they should be produced coinductively. This challenge is particularly apparent in (Beck et al., 2024)

when proving that the change in perspective from an infinite domain of memory addresses to a finite one is sound.

Second, the approach is very much akin to identifying a communicating system with its set of traces, except using a richer structure, namely monadic computations, instead of traces: it forgets all information about *when* nondeterministic choices are made. While such a trace equivalence is well suited for some applications, it tends to equate more processes than desired, which in turn may lead to failing to be a congruence for some contexts. In a general purpose semantics library, we believe we should strive to provide as much compositional reasoning as possible, and offer flexibility to the user: thus, our structure supports both trace equivalence, and bisimulations (Bloom et al., 1988).

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

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

**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 `fin n`. This limitation was triple. First most obviously, it prevented infinite branching (which is heavily used by Chappe et al. (2025)). Second, it did not allow carrying information about the origin of a given nondeterministic 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 7.2, and more generally enables 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,  $\emptyset$  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,  $Br_S$  nodes used to have their own constructor, and it was proved that they could be equivalently encoded as  $Br$  nodes guarded with a `Step` (unary  $Br_S$ ) node. Since we

introduced a dedicated *Step* node for the same reason as  $\emptyset$  and *Guard*, we removed the  $Br_S$  constructors and directly work with *Step*-guarded *Br* nodes.

On the semantic side, without the alternative definition of the LTS from Section 9, some results, especially from Section 7, 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.

On the proof engineering side, 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 Pous' library, but tower induction is more comfortable to work with, especially when up-to principles are involved.

Last, stepping back from these two existing, concrete, incarnations of the CTree data-structure, we observe that all new CTree constructions *could* be encoded into ITree events. Actually, already in the design of ITrees, leaves could have been special events with empty return types, and  $Step_i$  nodes an identified event with a unary return type. While seeking such minimalism in the structure itself can occasionally avoid *some* code duplication, we argue that having dedicated constructors for crucial constructs is beneficial from an engineering standpoint. Indeed, with a shallow encoding into events, operators whose behavior depends on them must assume their presence (through a partially concrete signature, or a typeclass constraint), leading to a more complex management of the injection of signatures. Furthermore, the code itself would have to pattern match on events, and hence feature a lot more dependent programming, the type of the branches in the match having to reify the return type of the event.

**Comparison to Bahr and Hutton (2023).** Following the original publication of CTrees (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 chains 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 Rocq, but investigating alternate encodings 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 7, 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 compilers for concurrent programs.

**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 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 bisimilarity that relies on an extra boolean in the bisimulation game. 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- $\tau$  nodes. Finally, they focus on notions of similarity, but do not study notions of bisimilarity.

Interestingly, Cho et al. (2023) observe not only that the forward and backward similarity from CompCert (Leroy, 2009) are subrelations of freely-stuttering similarity, but that *each step of these notions of simulation from CompCert* can be replayed by one or several steps of freely-stuttering simulation. They formally study this kind of relation and dub it *replayability*. It is stronger than top-level relation inclusion. Our initial definition of CTree simulation (resp. bisimulation) can be replayed by our finer alternative characterization of CTree simulation (resp. bisimulation). Lemma 11 expresses this result for strong simulation. However, replayability is not a silver bullet, in particular it does not guarantee that up-to principles valid on the replayed relation are also valid on the replaying relation.

## 12 Conclusion and Perspectives

We have introduced CTrees, a model for nondeterministic, recursive, and impure programs in Rocq. 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 games 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*  $\pi$ -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. (2025) 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 realistic concurrency support.

The library has been greatly enriched, from introducing additional equivalences and simulations, 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 models of concurrent programming languages.

### Acknowledgements

We are grateful to the anonymous reviewers from POPL'23 and JFP 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 advices he has 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.
- Ambal, G., Lenglet, S. & Schmitt, A. (2020)  $HO\pi$  in Coq. *Journal of Automated Reasoning*.
- 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: Reconciling integer–pointer casts, finite space, and undef at the LLVM IR level of abstraction. *Proc. ACM Program. Lang.* **8**(ICFP).
- 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.
- Borthelle, P., Hirschowitz, T., Jaber, G. & Zakowski, Y. (2025) An abstract, certified account of operational game semantics. ESOP.
- 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.

- 2674 Castellan, S., Clairambault, P., Rideau, S. & Winskel, G. (2017) Games and strategies as event  
2675 structures. *Log. Methods Comput. Sci.* **13**(3).
- 2676 Chappe, N., He, P., Henrio, L., Zakowski, Y. & Zdancewic, S. (2023) Choice trees: Representing  
2677 nondeterministic, recursive, and impure programs in Coq. *Proc. ACM Program. Lang.* **7**(POPL).
- 2678 Chappe, N., Henrio, L. & Zakowski, Y. (2025) Monadic interpreters for concurrent memory models:  
2679 Executable semantics of a concurrent subset of LLVM IR. Proceedings of the 14th ACM SIGPLAN  
2680 International Conference on Certified Programs and Proofs. New York, NY, USA. Association for  
2681 Computing Machinery. p. 283–298.
- 2682 Cho, M., Song, Y., Lee, D., Gäher, L. & Dreyer, D. (2023) Stuttering for free. *Proc. ACM Program.*  
2683 *Lang.* **7**(OOPSLA2).
- 2684 Crafa, S., Varacca, D. & Yoshida, N. (2012) Event structure semantics of parallel extrusion in the  
2685 pi-calculus. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial  
2686 Intelligence and Lecture Notes in Bioinformatics). pp. 225–239.
- 2687 Cristescu, I., Krivine, J. & Varacca, D. (2013) A Compositional Semantics for the Reversible  $\pi$ -  
2688 Calculus. Proceedings - Symposium on Logic in Computer Science. pp. 388–397.
- 2689 De Nicola, R. (2014) A gentle introduction to process algebras. *Notes*.
- 2690 Din, C. C., Hähnle, R., Johnsen, E. B., Pun, K. I. & Tapia Tarifa, S. L. (2017) Locally abstract,  
2691 globally concrete semantics of concurrent programming languages. Automated Reasoning with  
2692 Analytic Tableaux and Related Methods. Cham. Springer International Publishing. pp. 22–43.
- 2693 Din, C. C., Hähnle, R., Henrio, L., Johnsen, E. B., Pun, V. K. I. & Tarifa, S. L. T. (2022) Lage  
2694 semantics of concurrent programming languages.
- 2695 Foster, S., Hur, C. & Woodcock, J. (2021) Formally verified simulations of state-rich processes  
2696 using interaction trees in Isabelle/HOL. 32nd International Conference on Concurrency Theory,  
2697 CONCUR 2021, August 24–27, 2021, Virtual Conference. Schloss Dagstuhl - Leibniz-Zentrum für  
2698 Informatik. pp. 20:1–20:18.
- 2699 Harper, R. (2016) *Practical Foundations for Programming Languages*. Cambridge University Press.  
2700 second edition.
- 2701 Hoare, C. A. R. (1978) Communicating sequential processes. *Commun. ACM.* **21**(8), 666–677.
- 2702 Hur, C.-K., Neis, G., Dreyer, D. & Vafeiadis, V. (2013) The power of parameterization in coinductive  
2703 proof. Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of  
2704 Programming Languages. New York, NY, USA. ACM. pp. 193–206.
- 2705 Kang, J., Hur, C., Lahav, O., Vafeiadis, V. & Dreyer, D. (2017) A promising semantics for relaxed-  
2706 memory concurrency. Proceedings of the 44th ACM SIGPLAN Symposium on Principles of  
2707 Programming Languages, POPL 2017, Paris, France, January 18–20, 2017. ACM. pp. 175–189.
- 2708 King, J. C. (1976) Symbolic execution and program testing. *Communications of the ACM.* **19**(7),  
2709 385–394.
- 2710 Kiselyov, O. & Ishii, H. (2015) Freer monads, more extensible effects. Proceedings of the 8th ACM  
2711 SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3–4, 2015.  
2712 pp. 94–105.
- 2713 Koenig, J. & Shao, Z. (2020) Refinement-based game semantics for certified abstraction layers.  
2714 Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science. New  
2715 York, NY, USA. Association for Computing Machinery. p. 633–647.
- 2716 Lee, S., Cho, M., Podkopaev, A., Chakraborty, S., Hur, C., Lahav, O. & Vafeiadis, V. (2020) Promising  
2717 2.0: global optimizations in relaxed memory concurrency. Proceedings of the 41st ACM SIGPLAN  
2718 International Conference on Programming Language Design and Implementation, PLDI 2020,  
2719 London, UK, June 15–20, 2020. ACM. pp. 362–376.
- 2720 Leroy, X. (2009) Formal verification of a realistic compiler. *Communications of the ACM.* **52**(7),  
2721 107–115.
- 2722 Lesani, M., Xia, L., Kaseorg, A., Bell, C. J., Chlipala, A., Pierce, B. C. & Zdancewic, S. (2022) C4:  
2723 verified transactional objects. *Proc. ACM Program. Lang.* **6**(OOPSLA), 1–31.
- 2724 Letan, T., Régis-Gianas, Y., Chifflier, P. & Hiet, G. (2018) Modular verification of programs with  
2725 effects and effect handlers in Coq. Formal Methods - 22nd International Symposium, FM 2018,  
2726 Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15–17, 2018,  
2727

Proceedings. pp. 338–354.

2720

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

2721

2722

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

2723

2724

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

2725

2726

Michelland, S., Zakowski, Y. & Gonnord, L. (2024) Abstract interpreters: A monadic approach to modular verification. *Proc. ACM Program. Lang.* **8**(ICFP).

2727

2728

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

2729

2730

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

2731

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

2732

2733

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

2734

2735

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

2736

2737

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

2738

2739

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

2740

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

2741

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

2742

2743

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

2744

2745

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

2746

2747

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

2748

2749

Sangiorgi, D. & Walker, D. (2001) *The  $\pi$ -calculus*. Cambridge University Press. USA. first edition.

2750

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

2751

2752

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

2753

2754

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

2755

2756

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

2757

2758

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

2759

2760

The Coq Development Team. (2024) The Coq proof assistant .

2761

2762

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

2763

2764

van Glabbeek, R. J. (1997) Notes on the methodology of CCS and CSP. *Theor. Comput. Sci.* **177**(2), 329–349.

2765

2766

Xia, L., Zakowski, Y., He, P., Hur, C.-K., Malecha, G., Pierce, B. C. & Zdancewic, S. (2019) Interaction

2766 trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* **4**(POPL).  
2767 Yoon, I., Zakowski, Y. & Zdancewic, S. (2022) Formal reasoning about layered monadic interpreters.  
2768 *Proceedings of the ACM on Programming Languages.* **6**(ICFP).  
2769 Zakowski, Y., Beck, C., Yoon, I., Zaichuk, I., Zaliva, V. & Zdancewic, S. (2021) Modular,  
2770 compositional, and executable formal semantics for LLVM IR. *Proc. ACM Program. Lang.* **5**(ICFP).  
2771 Zakowski, Y., He, P., Hur, C.-K. & Zdancewic, S. (2020) An equational theory for weak bisimulation  
2772 via generalized parameterized coinduction. Proceedings of the 9th ACM SIGPLAN International  
2773 Conference on Certified Programs and Proofs (CPP).  
2774  
2775  
2776  
2777  
2778  
2779  
2780  
2781  
2782  
2783  
2784  
2785  
2786  
2787  
2788  
2789  
2790  
2791  
2792  
2793  
2794  
2795  
2796  
2797  
2798  
2799  
2800  
2801  
2802  
2803  
2804  
2805  
2806  
2807  
2808  
2809  
2810  
2811

## Appendix

## A Coinductive proofs and up-to principles in Rocq

Working with CTrees requires the use of a swarm of coinductive predicates and relations to describe equivalences, simulations and invariants. Doing so at scale in Rocq 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 Rocq enforces through a syntactic guard checker. However, in the case of corecursion, guard checking is incompatible with any automation, and not compositional. Thankfully, coinduction is nowadays possible in Rocq thanks to library support (Hur et al., 2013; Zakowski et al., 2020; Pous, 2016). In essence, they all rely on an internalization of the theory of coinduction inside a library instead of using Rocq’s native coinduction support for defining the relation of interest,

Consequently, our development relies on Pous’s coinduction library (Pous, 2024) to define and reason about the various coinductive predicates and relations we manipulate. Rather than relying on Rocq’s native coinduction support, the library defines the necessary tools for coinductive reasoning from scratch, based on the *companion* (Pous, 2016) and *tower induction* (Schäfer and Smolka, 2017). This appendix presents the library we use in the rest of our developments.

**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 Rocq propositions  $\text{Prop}$  forms a complete lattice, as does 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  $C : \text{ctree } E \ B \ A \rightarrow \text{ctree } E \ B \ A \rightarrow \text{Prop}$  for fixed parameters  $E$ ,  $B$  and  $A$ . We write such binary relations as  $\text{rel}(A, B)$  for  $A \rightarrow B \rightarrow \text{Prop}$ , and  $\text{rel}(A)$  for  $\text{rel}(A, A)$ ; for instance:  $C := \text{rel}(\text{ctree } E \ B \ A)$ .

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):

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

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  $\mathbf{f} : \mathbf{X} \rightarrow \mathbf{X}$  defines a valid enhanced coinduction principle if the following reasoning principle is valid:

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

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  $f_{\text{refl}} R \triangleq R \cup \{(t, t)\}$  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 5.3. 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  $f_{\text{bisim}} R \triangleq \{(t, u) \mid \exists t' u', t \sim t' \wedge u' \sim u \wedge R t' u'\}$ , is sufficient to conclude.

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

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

In particular, Pous's library is built upon the so-called *companion*, a construction which relies on one particular class of functions, the so-called *compatible* functions. Their characterization is fairly elementary: a monotone endofunction  $f$  is compatible with  $b$  if  $f b \sqsubseteq b f$  (i.e.,  $\forall R, f(b(R)) \subseteq b(f(R))$ ). The class is of particular interest for two main properties. First, all compatible functions are sound up-to functions. Furthermore, the set of compatible functions for a given  $b$  forms a complete lattice. One may therefore consider the greatest compatible function, dubbed the *companion* and written  $t_b$ , and observe it is itself compatible. We refer the interested reader to Pous (2016) for more details, and to Pous (2024) for pedagogical example of the library in action.

From a user perspective, this is priceless: rather than craft and pick the right up-to principle for each proof, they can systematically work up to the companion, progressively

enhance their database of proven compatible principles,<sup>26</sup> and access them on the fly during the proofs. In the three examples above, access to these up-to principles would therefore be granted by proving respectively  $f_{\text{refl}} \sqsubseteq b_t$ ,  $f_{\text{bisim}} \sqsubseteq b_t$ , and  $f_{\text{br}} \sqsubseteq b_t$ , which in turn can be in particular proved by showing that  $f_{\text{refl}}$ ,  $f_{\text{bisim}}$ , and  $f_{\text{br}}$  are compatible.

**Tower induction.** Pous’s library relies on Schäfer and Smolka’s characterization of the companion via tower induction (Schäfer and Smolka, 2017). They associate to the endofunction  $b$  the so-called  $b$  – tower, or  $C_b$  the Chain of  $b$ , i.e., the inductive type closed under  $b$  and greatest lower bound. The greatest fixpoint of  $b$  is recovered as the infimum of its chain,  $\text{gfp } b = \inf C_b$ , and more generally the companion as  $t_b(x) = \inf\{y \in C_b \mid x \sqsubseteq y\}$ . Once again, we refer the interested reader to (Schäfer and Smolka, 2017) for technical details, and only highlight here the consequences for us, users of the library: since version 1.7, Pous’s library has been reimplemented following Schäfer and Smolka’s construction.

The usability improvement compared to the earlier companion-based implementation shows when proving the validity of additional proof principles: in particular, exploiting previously established sound principles in the proof of a new one required a clever but non-trivial notion of second-order companion before. In contrast, when working with tower induction, the statement of validity of an up-to principle defined by a function  $f$  is much more natural: it essentially amounts to proving that  $f$  preserves membership in the chain.

Let us make things concrete over our the three illustrative examples. Validity of  $f_{\text{refl}}$  is now expressed by stating that all elements of the chain are reflexive, i.e.,  $\forall(c : C_b) t, c \ t \ t$ . Validity of  $f_{\text{bisim}}$  captures that elements of the chain are stable by bisimilarity on both 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 in Rocq as `Proper (sbisim  $\implies$  sbisim  $\implies$  iff) c`. Finally, validity of  $f_{\text{br}}$  is written as  $\forall(c : C_b) p \ p' \ q \ q', c \ p \ p' \rightarrow c \ q \ q' \rightarrow c \ (\text{br } p \ \text{or } q) \ (\text{br } p' \ \text{or } q')$ , or again more concisely expressed in Rocq as `Proper (c  $\implies$  c  $\implies$  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 greatest lower bound 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.

<sup>26</sup> Or slightly more generally, of any function proven below the companion.