Choice Trees: Representing Nondeterministic, Recursive, and Impure Programs in Coq

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

Choice Trees

Representing Nondeterministic, Recursive, and Impure Programs in Coq

NICOLAS CHAPPE, Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP, F-69342, LYON Cedex 07, France
PAUL HE, University of Pennsylvania, USA
LUDOVIC HENRIO, Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP, F-69342, LYON Cedex 07, France
YANNICK ZAKOWSKI, Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP, F-69342, LYON Cedex 07, France
STEVE ZDANCEWIC, University of Pennsylvania, USA

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

We connect CTrees to the ITrees infrastructure by showing how a monad morphism embedding the former into the latter permits to use 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.

CCS Concepts: • **Theory of computation → Denotational semantics**; *Program verification*; **Concurrency**.

Additional Key Words and Phrases: Nondeterminism, Formal Semantics, Interaction Trees, Concurrency

ACM Reference Format:
Nicolas Chappe, Paul He, Ludovic Henrio, Yannick Zakowski, and Steve Zdancewic. 2023. Choice Trees: Representing Nondeterministic, Recursive, and Impure Programs in Coq. *Proc. ACM Program. Lang.* 7, POPL, Article 61 (January 2023), 31 pages. https://doi.org/10.1145/3571254

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

Authors' addresses: Nicolas Chappe, Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP, F-69342, LYON Cedex 07, France, nicolas.chappe@ens-lyon.fr; Paul He, University of Pennsylvania, Philadelphia, PA, USA, paulhe@cis.upenn.edu; Ludovic Henrio, Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP, F-69342, LYON Cedex 07, Lyon, France, ludovic.henrio@cnrs.fr; Yannick Zakowski, Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP, F-69342, LYON Cedex 07, France, yannick.zakowski@inria.fr; Steve Zdancewic, University of Pennsylvania, USA, stevez@cis.upenn.edu.

2475-1421/2023/1-ART61
https://doi.org/10.1145/3571254

which have been used to give semantics to nondeterministic programming language features such as concurrency [Harper 2016; Milner 1989; Sangiorgi and Walker 2001].

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

This paper introduces a new formalism designed specifically to facilitate the definition of and reasoning about nondeterministic computations in Coq's dependent type theory. The key idea is to update Xia, et al.'s *interaction trees* (ITrees) framework [Xia et al. 2020] with native support for nondeterminisic "choice nodes" that represent internal choices made during computation. The main technical contributions of this paper are to introduce the definition of these CTrees ("choice trees") and to develop the suitable metatheory and equational reasoning principles to accommodate that change.

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

At the same time, the notion of bisimulation for CTrees is still connected to familiar definitions like those from labeled transition systems (LTS), meaning that much of the well-developed theory from prior work can be imported whole-sale. Indeed, we define bisimilarity for CTrees by viewing them as LTSs and applying standard definitions. The fact that the definition of bisimulation ends up

being subtle and nontrivial is a sign that we gain something by working with the CTrees: like their ITree predecessors, CTrees have compositional reasoning principles, the type is a monad, and the useful combinators for working with ITrees, namely sequential composition, iteration and recursion, interpretation, etc., all carry over directly. CTrees, though, further allow us to conveniently, and flexibly, define nondeterminstic semantics, ranging from simple choice operators to various flavors of parallel composition. The benefit is that, rather than just working with a "raw" LTS directly, we can construct one using the CTrees combinators—this is a big benefit because, in practice, the LTS defining the intended semantics of a nondeterministic programming language cannot easily be built in a compositional way without using some kind of intermediate representation, which is exactly what CTrees provides (see the discussion about Figure 3). A key technical novelty of our CTrees definition is that it makes a distinction between *stepping* choices (which correspond to τ 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). This design allows for compositional construction of the LTS and generic reasoning rules that are usable in any context.

The net result of our contributions is a library, entirely formalized in Coq, that offers flexible building blocks for constructing nondeterministic, and hence concurrent, models of computation. To demonstrate the applicability of this library, we use it to implement the semantics from two different formalisms: `ccs` [Milner 1989] and a language with cooperative threads inspired from the literature [Abadi and Plotkin 2010]. Crucially, in both of these scenarios, we are able to define the appropriate parallel composition combinators such that the semantics of the programming language can be defined fully compositionally (i.e., by straightforward induction on the syntax). Moreover, we recover the classic definition of program equivalence for `ccs` directly from the equational theory induced by the encoding of the semantics using CTrees; for the language with cooperative threading, we prove some standard program equivalences.

To summarize, this paper makes the following contributions:

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

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

The remainder of the paper is organized as follows. The next section gives some background about interaction trees and monadic interpreters, along with a discussion of the challenges of modeling nondeterminism, laying the foundation for our results. We introduce the CTrees data structure and its main combinators in Section 3. Section 4 introduces several notions of equivalences over CTrees—(coinductive) equality, strong bisimilarity, weak bisimilarity, and trace equivalence—and describes its core equational theory. Section 5 describes how to interpret uninterpreted events in an ITree into "choice" branches in a CTree, as well as how to define the monadic interpretation of events from CTrees. Section 6 describes our first case study, a model for `ccs`. Section 7 describes

```
CoInductive itree (E: Type → Type) (R: Type) : Type :=
| Ret (r: R)                              (* computation terminating with value r *)
| Vis {A: Type} (e : E A) (k : A → itree E R) (* event e yielding an answer in A *)
| later (t: itree E R).                   (* "silent" tau transition with child t *)
```

Fig. 1. Interaction trees: definition

our second case study, a model for the imp language extended with cooperative multithreading. Finally, Section 8 discusses related work and concludes.

2 BACKGROUND

2.1 Interaction trees and monadic interpreters

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

Interaction trees are coinductive data structures for representing (potentially divergent) computations that interact with an external environment through *visible events*. A definition of the ITree datatype is shown in Figure 1.[1] The datatype takes as its first parameter a signature—described as a family of types `E : Type → Type`—that specifies the set of interactions the computation may have with the environment. The `Vis` constructor builds a node in the tree representing such an interaction, followed by a continuation indexed by the return type of the event. The second parameter, `R`, is the *result type*, the type of values that the entire computation may return, if it halts. The constructor `Ret` builds such a pure computation, represented as a leaf. Finally, the `later` constructor models an internal, non-observable step of computation, allowing the representation of silently diverging computations; it is also used for guarding corecursive definitions.[2]

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

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

[1] The signature of ITrees is presented with a positive coinductive datatype for expository purposes. The actual implementation is defined in the negative style.

[2] The ITree library uses Tau to represent `later` nodes. Tau and τ are overloaded in our context, so we rename it to `later` here to avoid ambiguity. CTrees will replace the `later` (i.e. Tau) constructor with a more general construct anyway.

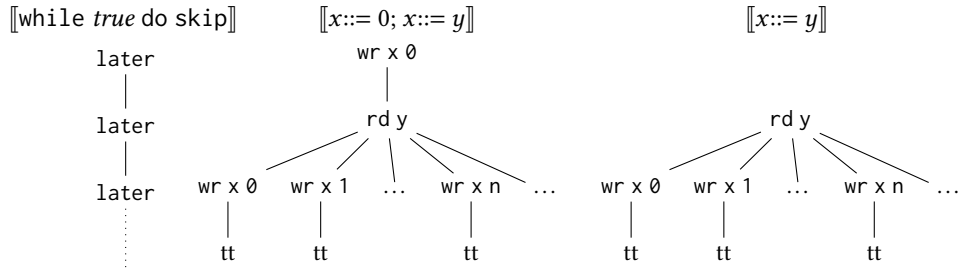


Fig. 2. Example ITrees denoting the imp programs p_1, p_2, and p_3.

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

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

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

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

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

$p_2 =$ `Vis (wr x 0) (fun _ ⇒ Vis (rd y) (fun ans ⇒ (Vis (wr x ans) (fun _ ⇒ Ret tt))))`.

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

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

$$\begin{aligned} \texttt{interp}\ h_{mem}\ [\![p_2]\!]\ \texttt{m} &= \texttt{later} - \texttt{later} - \texttt{later} - (\texttt{m}\{x \leftarrow 0\}\{x \leftarrow \texttt{m}(y)\}, tt) \\ \texttt{interp}\ h_{mem}\ [\![p_3]\!]\ \texttt{m} &= \texttt{later} - \texttt{later} - (\texttt{m}\{x \leftarrow \texttt{m}(y)\}, tt) \end{aligned}$$

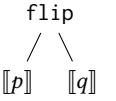
The `later` nodes are introduced by the interpretation of the memory events. More precisely, an `interp` combinator applies the handler h_{mem} to the `rd` and `wr` nodes of the trees, implementing their semantics in terms of the state monad. Assuming an appropriate implementation of the memory, one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

```
Variant Flip : Type → Type :=
| flip : Flip bool.
```

```
Vis flip (fun b ⇒ if b then p else q)
```



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

Zakowski et al. used this approach in the Vellvm project [Zakowski et al. 2021] for formalizing the nondeterministic features of the LLVM IR. Their model consists of a propositionally-specified set of computations: ignoring other effects, the monad they use is `itree E _ → 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. 2022b], this issue is not specific to ITrees but rather to an hypothetical "Prop Monad Transformer", i.e. to "fun M X => M X -> 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. 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. As has been well identified by the process calculi tradition, and while trace equivalence is sometimes the desired relation, such a model leads to equivalences of programs that are too coarse to be compositional in general. In a general purpose semantics library, we believe we should strive to provide as much compositional reasoning as possible and thus our tools support 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 — non-determinism 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.

This paper introduces CTrees, a suitable monad for modeling nondeterministic effects. As with ITrees, the structure is compatible with divergence, external interaction through uninterpreted events, extraction to executable reference interpreters, and monadic interpretation. The core intuition is based on the observation that the tree-like structure from ITrees is indeed the right one for modeling nondeterminism. The problem arises from how the ITrees definition of `eutt` observes which branch is taken, requiring that *all* branches be externally visible: while appropriate to model nondeterminism that results from a lack of information, it does not correspond to true *internal* choice. Put another way, thinking of the trees as labeled transition systems, ITrees are *deterministic*. With CTrees, we therefore additionally consider truly branching nodes, explicitly build the associated nondeterministic LTS, and define proper bisimulations on the structure.

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

$$\texttt{comm} \triangleq \texttt{skip} \mid x ::= e \mid c1; c2 \mid \texttt{while}\ b\ \texttt{do}\ c \mid \texttt{br}\ c1\ \texttt{or}\ c2 \mid \texttt{block} \mid \texttt{print}$$

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

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

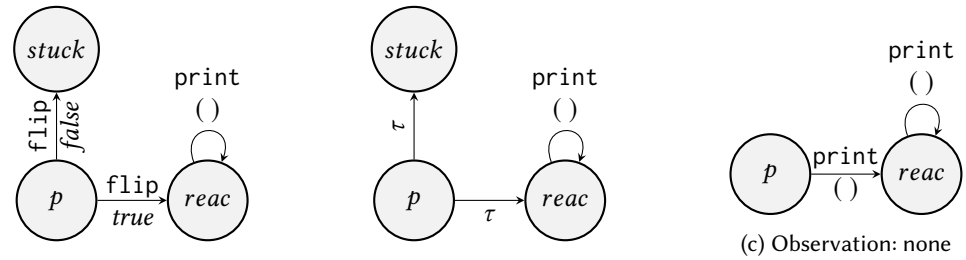
$$\frac{}{\texttt{br}\ c1\ \texttt{or}\ c2 \longrightarrow c1}\ \textsc{BrInternal} \qquad \frac{c_1 \longrightarrow c_1'}{\texttt{br}\ c_1\ \texttt{or}\ c_2 \longrightarrow c_1'}\ \textsc{BrDelayed}$$

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

In order to design the right monadic structure allowing for enough flexibility to model either behavior, it is useful to look ahead and anticipate how we will reason about program equivalence, as described in detail in Section 4. The intuition we follow is to interpret our computations as labeled transition systems and define bisimulations over those, as is generally done in the process algebra literature. From this perspective, the `imp` program p may correspond to three distinct LTSs depending on the intended semantics, as shown in Figure 3.

Figure 3a 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 $[\![\texttt{br}\ p\ \texttt{or}\ q]\!] \triangleq \mathit{Vis}\ \texttt{flip}\ \lambda b \cdot \text{if } b \text{ then } [\![p]\!] \text{ else } [\![q]\!]$.

Figure 3b corresponds to BrInternal: both the stuck and the reactive states are reachable, but we do not observe the label of the transition. This transition exactly corresponds to the internal τ step of process algebra. This situation is captured by introducing a new kind of node in our data



(a) Observation: branches (b) Observation: branching (c) Observation: none

Fig. 3. Three possible semantics for the program p, from an LTS perspective

structure, a Br_S branch, that maps in our bisimulations defined in Section 4 to a nondeterministic *internal step*. For this semantics, we thus have $[\![\texttt{br}\ p\ \texttt{or}\ q]\!] \triangleq Br_S^2\ \lambda b \cdot \text{if } b \text{ then } [\![p]\!] \text{ else } [\![q]\!]$.[3]

Figure 3c corresponds to BrDelayed but raises the question: how do we build such a behavior? A natural answer can be to assume that it is the responsibility of the model, i.e., the function mapping `imp`'s syntax to the semantic domain, CTrees, to explicitly build this LTS. Here, $[\![p]\!]$ would be an infinite sequence of `Vis print` nodes, containing no other node. While that would be convenient for developing the meta-theory of CTrees, this design choice would render them far less compositional (and hence less useful) than we want them to be. Indeed, we want our models to be defined as computable functions by recursion on the syntax, whenever possible. But to build this LTS directly, the model for `br p or q` needs to introspect the models for $[\![p]\!]$ and $[\![q]\!]$ to decide whether they can take a step, and hence whether it should introduce a branching node. But, in general, statically determining whether the next reachable instruction is `block` is intractable, so that introspection will be hard (or impossible) to implement. We thus extend CTrees with a third category of node, a Br_D node, which does not directly correspond to states of an LTS. Instead, Br_D nodes aggregate sub-trees such that the (inductively reachable) Br_D children of a Br_D node are "merged" in the LTS view of the CTree. This design choice means that, for the BrDelayed semantics, the model is again trivial to define: $[\![\texttt{br}\ p\ \texttt{or}\ q]\!] \triangleq Br_D^2\ \lambda b \cdot \text{if } b \text{ then } [\![p]\!] \text{ else } [\![q]\!]$, but the definition of bisimilarity for CTrees ensures that the behavior of $[\![p]\!]$ is exactly the LTS in Figure 3c.

Although BrDelayed and its corresponding LTS in Figure 3c is *one* example in which Br_D nodes are needed, we will see another concrete use in modeling `ccs` in Section 6. Similar situations arise frequently, for modeling mutexes, locks, and other synchronization mechanisms (e.g., the "await" construct in Encore [Brandauer et al. 2015]), dealing with crash failures in distributed systems, encoding relaxed-consistency shared memory models (e.g., the ThreadPromise rule of promising semantics [Kang et al. 2017]). The Br_D construct is needed whenever the operational semantics includes a rule whose possible transitions depend on the existence of other transitions, i.e. for any rule of the shape shown below, when it may sometimes be the case (usually due to nondeterminism) that $P \not\twoheadrightarrow$:

$$\frac{P \to P'}{C[P] \to C[P']}\ \textsc{ContingentStep}$$

The point is that to implement the LTS corresponding to ContingentStep without using a Br_D node would require introspection of P to determine whether it may step, which is potentially non-computable. The Br_D node bypasses the need for that introspection at *representation time*, instead pushing it to the characterization of the CTree as an LTS, which is used only for *reasoning* about the semantics. The presence of the Br_D nodes retains the constructive aspects of the model,

[3]The 2 indicates the arity of the branching.

```
(* Core datatype *)
CoInductive ctree (E : Type → Type) (R : Type) :=
| Ret (r : R)                                    (* a pure computation *)
| Vis {X : Type} (e : E X) (k : X → ctree)       (* an external event *)
| brS (n : nat) (k : fin n → ctree)              (* stepping branching *)
| brD (n : nat) (k : fin n → ctree)              (* delayed branching *)

(* Bind, sequencing computations *)
CoFixpoint bind {E T U} (t : ctree E T) (k : T → ctree E U) : ctree E U :=
match u with
| Ret r   ⇒ k r
| Vis e h ⇒ Vis e (fun x ⇒ bind (h x) k)
| brS n h ⇒ brS n (fun x ⇒ bind (h x) k)
| brD n h ⇒ brD n (fun x ⇒ bind (h x) k)
end

(* Unary guards *)
Definition Guard (t : ctree E R) : ctree E R := brD 1 (fun _ ⇒ t)
Definition Step  (t : ctree E R) : ctree E R := brS 1 (fun _ ⇒ t)

(* Main fixpoint combinator *)
CoFixpoint iter {I: Type} (body : I → ctree E (I + R)) : I → ctree E R :=
  bind (body i) (fun lr ⇒ match lr with
                          | inr r ⇒ Ret r
                          | inl i ⇒ Guard (iter body i)
                          end)

Notation "E ⇝ F" := (∀ X, E X → F X)
(* Atomic ctrees triggering a single event *)
Definition trigger : E ⇝ ctree E := fun R (e : E R) ⇒ Vis e (fun x ⇒ Ret x)
(* Atomic branching ctrees *)
Definition brS : ctree E (fin n) := fun n ⇒ brS n (fun x ⇒ Ret x)
Definition brD : ctree E (fin n) := fun n ⇒ brD n (fun x ⇒ Ret x)
```

Fig. 4. CTrees: definition and core combinators ( )

in particular the ability to *interpret* these nodes at later stages, for instance to obtain an executable version of the semantics.

3 CTREES: DEFINITION AND COMBINATORS

3.1 Core definitions

We are now ready to define our core datatype, displayed in the upper part of Figure 4. The definition remains close to a coinductive implementation of the free monad, but hardcodes support for an additional effect: unobservable, nondeterministic branching. The CTree datatype, much like an ITree, is parameterized by a signature of (external) events `E` encoded as a family of types, and a return type `R`. It is defined as a coinductive tree[4] with four kind of nodes: pure computations (`Ret`), external events (`Vis`), internal branching with implicitly associated τ step (`brS`), and delayed internal branching (`brD`). The continuation following external events is indexed by the return type specified by the emitted event. For the sake of simpler exposition, we restrict both internal branches to be

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

```
(* Stuck processes *)
Definition stuckE (e : E void) : ctree E void := trigger e
Definition stuckS : ctree E void := brS 0
Definition stuckD : ctree E void := brD 0
CoFixpoint spinD : ctree E R := Guard spinD
CoFixpoint spinD_nary n : ctree E R := brD n ;; spinD_nary n

(* Spinning processes *)
CoFixpoint spinS : ctree E R := Step spinS
CoFixpoint spinS_nary n : ctree E R := brS n ;; spinS_nary n
```

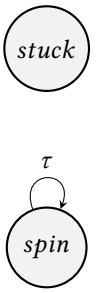


Fig. 5. Concrete representations of stuck and spinning LTSs

of finite width: their continuation are indexed by finite types `fin`, but this is not a fundamental limitation.[5] When using finite branching, we abuse notation and write, for instance, $Br_S^2\ t\ u$ for the computation branching with two branches, rather than explicitly spelling out the continuation which branches on the choice index: `fun i ⇒ match i with 0 ⇒ t | 1 ⇒ u end`.

The remainder of Figure 4 displays (superficially simplified) definitions of the core combinators. As expected, `ctree E` forms a monad for any interface `E`: the `bind` combinator simply lazily crawls the potentially infinite first tree, and passes the value stored in any reachable leaf to the continuation. The `iter` combinator is central to encoding looping and recursive features: it takes as argument a body, `body`, intended to be iterated, and is defined such that the computation returns either a new index over which to continue iterating, or a final value; `iter` ties the recursive knot. Its definition is analogous to the one for ITrees, except that we need to ask ourselves how to guard the `cofix`: if `body` is a constant, pure computation, unguarded corecursion would be ill-defined. ITrees use a `later` node for this purpose. Here, we instead use a unary delayed branch as a guard, written `Guard`. We additionally write `Step` for the unary stepping branch—we will discuss how they relate in Section 5.3. The minimal computations respectively triggering an event `e`, generating observable branching, or delaying a branch, are defined as `trigger`, Br_S^n, and Br_D^n.

Convenience in building models comes at a cost: many CTrees represent the same LTS. Figure 5 illustrates this by defining several CTrees implementing the stuck LTS and the silently spinning one. The `stuckS` and `stuckD` correspond to internal choices with no outcome, while `stuckE` is a question to the environment that cannot be answered, leaving the computation hanging; the `spinD*` trees are infinitely deep, but never find in their structure a transition to take. We define formally the necessary equivalences on computations to prove this informal statement in Section 4. The astute reader may wonder whether both kind of branching nodes really are necessary. While convenient, we show in Section 4.5 that Br_S can be expressed in terms of Br_D and `Step`.

3.2 A hint of introspection: heads of computations

Br_D nodes avoid the need for introspection on trees to model something as generic as a delayed branching construct such as the one specified by BrDelayed. However, introspection becomes necessary to build a tree that depends on the reachable external actions of the sub-trees. This is the case for example for `ccs`'s parallel operator that we model in Section 6. 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

[5] We actually also support an extended version of the structure that defines *enhanced CTrees*, with arbitrary branching over indexes specified in the type of the data structure by an interface akin to `E`.

```
Variant action E R :=
  | ARet (r : R)
  | ABr {n} (k : fin n → ctree E R)
  | AVis {X} (e : E X) (k : X → ctree E R).

CoFixpoint head {E X} (t : ctree E X) : ctree E (action E X) :=
  match t with
  | Ret x ⇒ Ret (ARet x) | Vis e k ⇒ Ret (AVis e k) | brS n k ⇒ Ret (ABr k)
  | BrD n k ⇒ BrD n (fun i ⇒ head (k i))
  end.
```

Fig. 6. Lazily computing the set of reachable observable nodes (🔥)

are reachable. We are, however, in luck, as we have at hand a semantic domain able to represent potentially divergent computations: CTrees themselves!

The `head` combinator, described on Figure 6, 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`[6] 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`.

4 EQUIVALENCES AND EQUATIONAL THEORY FOR CTREES

Section 3 introduced CTrees, the domain of computations we consider, as well as a selection of combinators upon it. We now turn to the question of comparing computations represented as CTrees for notions of equivalence and refinement. In particular, we formalize the LTS representation of a CTree that we have followed to justify our definitions. This section first introduces a (coinductive) syntactic equality of CTrees, then it recovers the traditional notions of strong and weak bisimilarity of processes, as well as trace equivalence, for an LTS derived from the tree structure. We equip these notions with a primitive equational theory for CTrees; this theory provides the building blocks necessary for deriving domain-specific equational theories, such as the ones established in Section 6 for `ccs`, and in Section 7 for cooperative scheduling.

4.1 Coinductive proofs and up-to principles in Coq

Working with CTrees requires the use of several coinductive predicates and relations to describe equivalences, refinements and invariants. Doing so at scale in `Coq` would be highly impractical if only using its native support for coinductive proofs, but it is nowadays more feasible thanks to library support [Hur et al. 2013; Pous 2016; Zakowski et al. 2020]. Our development relies on Pous's `coinduction` library [Pous 2022a] to define and reason about the various coinductive predicates and relations we manipulate. We briefly recall the essential facilities, based on the *companion* [Pous 2016], that the library provides. In particular, we define a subset of candidate "up-to" functions—we indicate in the subsequent subsections which ones constitute valid up-to principles for respectively the structural equality and the strong bisimulation we manipulate. Note: this section is aimed at the reader interested in understanding the internals of our library: it can be safely skipped at first read.

The core construction provided by the library is a greatest fixpoint operator (`gfp` b`:X`) for any complete lattice `X`, and monotone endofunction b : `X` → `X`. In particular, the sort of `Coq` propositions

[6]The `head t` computation could be typed at the empty interface. It is in practice simpler to directly type it at the same interface as the `t`.

$$\textsc{refl}^{up} R \triangleq \{(x,x)\} \qquad \textsc{sym}^{up} R \triangleq \{(y,x) \mid R\ x\ y\} \qquad \textsc{trans}^{up} R \triangleq \{(x,z) \mid \exists y,\ R\ x\ y \wedge R\ y\ z\}$$
$$\textsc{Vis}^{up} R \triangleq \{(Vis\ e\ k, Vis\ e\ k') \mid \forall v,\ R\ (k\ v)\ (k'\ v)\}$$
$$\textsc{bind}^{up}(equiv)\ R \triangleq \{(x \gg\!= k, y \gg\!= l) \mid equiv\ x\ y \wedge \forall v, R\ (k\ v)\ (l\ v)\}$$
$$\textsc{upto}^{up}(equiv)\ R \triangleq \{(x,y) \mid \exists x'\ y',\ equiv\ x\ x' \wedge R\ x'\ y' \wedge\ equiv\ y'\ y\}$$

Fig. 7. Main generic up-to principles used for relations of CTrees where `R : rel (ctree E X)`

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

The library provides tactic support for coinductive proofs based on Knaster-Tarski's theorem: any post fixpoint is below the greatest fixpoint. Specialized to relations on CTrees, the proof method 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 `(∀ t u, R t u → b R t u) → ∀ t u, R t u → gfp b t u`. The major benefit of the companion is, however, to further provide support for proofs by *enhanced coinduction*.

Given an endofunction b, a (sound) enhanced coinduction principle, also known as an *up-to principle*, relies on an additional function f: `X → X` allowing one to work with `bf` (the composition of b with f) instead of b: any post fixpoint of bf is below the greatest fixpoint of b. Concretely, the user has now access to a new proof principle. Rather than having to "fall back" exactly into their coinduction hypothesis after "stepping" through b, they may first apply f. In the case of a coinductive relation, simple examples of up-to principles include adding the diagonal or swapping the arguments, allowing one to conclude by reflexivity or invoke symmetry regardless of the coinduction candidate considered.

Significant effort has been invested in identifying classes of sound up-to principles, and developing ways to combine them [Pous 2007; Sangiorgi 1998; Sangiorgi and Rutten 2012]. The companion relies on one particular sub-class of sound principles that forms a complete lattice, the *compatible* functions: we hence can conduct proofs systematically up-to the greatest compatible function, dubbed the *companion* and written $\mathtt{t}_b$. In practice, this means that a coinductive proof can, on-the-fly, use any valid up-to principle drawn from the companion. We refer the interested reader to the literature [Pous 2016] and our formal development for further details, and describe below the specialization to CTrees of the up-to functions we use.

Figure 7 describes the main generic up-to principles we use for our relations on CTrees.[7] Note that since we are considering relations on CTrees, each of these principles, these candidates "f", are endofunctions of relations: for instance, $\textsc{refl}^{up}$ is the constant diagonal relation, $\textsc{sym}^{up}$ builds the symmetric relation, $\textsc{trans}^{up}$ is the composition of relations. The validity of $\textsc{refl}^{up}$, $\textsc{sym}^{up}$ and $\textsc{trans}^{up}$ for a given endofunction b entails respectively the reflexivity, symmetry and transitivity of the relations $(b\mathtt{t}_b\ R)$ and $(\mathtt{t}_b\ R)$. 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 $\textsc{Vis}^{up}$ and $\textsc{bind}^{up}(\_)$ up-to functions help when reasoning structurally, respectively allowing to cross through `Vis` nodes and `bind` constructs during proofs by coinduction. Finally, validity of the $\textsc{upto}^{up}(equiv)$ principle allows for rewriting via the *equiv* relation during coinductive proofs for b.

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

4.2 Coinductive equality for CTree

Coq's equality, `eq`, is not a good fit to express the structural equality of coinductive structures—even the eta-law for a coinductive data structure does not hold up-to `eq`. We therefore define, as is standard, a structural equality[8] by coinduction `equ: rel(ctree E A)` (written $\cong$ in infix). The endofunction simply matches head constructors and behaves extensionally on continuations.

Definition 4.1. Structural equality (🐢)

$$\texttt{equ} \triangleq \texttt{gfp}\ \lambda R \cdot \{(\texttt{Ret}\ v,\ \texttt{Ret}\ v)\} \cup \{(Vis\ e\ k,\ Vis\ e\ k') \mid \forall v,\ R\ (k\ v)\ (k'\ v)\} \cup \{(Br_D^n\ k,\ Br_D^n\ k') \mid \forall v,\ R\ (k\ v)\ (k'\ v)\} \cup \{(Br_S^n\ k,\ Br_S^n\ k') \mid \forall v,\ R\ (k\ v)\ (k'\ v)\}$$

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

Lemma 4.2. *Monadic laws (🐢)*

$$\texttt{Ret}\ v \gg\!= k \cong k\ v \qquad x \leftarrow t\ ;;\ \texttt{Ret}\ x \cong t \qquad (t \gg\!= k) \gg\!= l \cong t \gg\!= (\lambda x \Rightarrow k\ x \gg\!= l)$$

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

Finally, we establish some enhanced coinduction principles for equ.

Lemma 4.3. *Enhanced coinduction for* equ *(🐢)*
REFL^{up}, SYM^{up}, TRANS^{up}, BIND$^{up}(\cong)$ *and* UPTO$^{up}(\cong)$ *provide valid up-to principles for* equ.

While equ, being 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 differences in internal steps, that still imposes a tight correspondence over external events, but relaxes its requirement for nondeterministically branching nodes. We achieve this by drawing from standard approaches developed for process calculi.

4.3 Looking at CTrees under the lens of labeled transition systems

To build a notion of bisimilarity between CTree computations, we associate a labeled transition system to a CTree, as defined in Figure 8. This LTS exhibits three kinds of labels: a `tau`[9] witnesses a stepping branch, an `obs e x` observes the encountered event together with the answer from the environment considered, and a `val v` is emitted when returning a value. Interestingly, there is a significant mismatch between the structure of the tree and the induced LTS: the states of the LTS correspond to the nodes of the CTree *that are not immediately preceded by a delayed choice.* Accordingly, the definition of the transition relation between states inductively iterates over delayed branches. Stepping branches and visible nodes map immediately to a set of transitions, one for each outgoing edge; finally a return node generates a single `ret` transition, moving onto a stuck state, encoded as a nullary branching node and written $\emptyset$. These rules formalize the intuition we gave in Section 2.2 that allowed us to derive the LTSs of Figure 3 from the corresponding `ImpBr` terms.

Defining the property of a tree to be *stuck*, that is: $t \nrightarrow \triangleq \forall l\ u,\ \neg(t \xrightarrow{l} u)$, we can make the depictions from Figure 5 precise: nullary nodes are stuck by construction, since stepping would

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

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

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

$$\frac{k\ v \xrightarrow{l} t}{Br^n_D\ k \xrightarrow{l} t} \qquad \frac{}{Br^n_S\ k \xrightarrow{\texttt{tau}} k\ v} \qquad \frac{}{Vis\ e\ k \xrightarrow{\texttt{obs}\ e\ v} k\ v} \qquad \frac{}{\texttt{Ret}\ v \xrightarrow{\texttt{val}\ v} \emptyset}$$

Fig. 8. Inductive characterisation of the LTS induced by a CTree (🦀)

$$\frac{t \xrightarrow{l} u \quad l \neq \texttt{val}\ v}{t \gg\!= k \xrightarrow{l} u \gg\!= k} \qquad \frac{t \xrightarrow{\texttt{val}\ v} \emptyset \quad k\ v \xrightarrow{l} u}{t \gg\!= k \xrightarrow{l} u}$$

$$\frac{t \gg\!= k \xrightarrow{l} u}{(l \neq \texttt{val}\ v \wedge \exists t', t \xrightarrow{l} t' \wedge u \cong t' \gg\!= k) \vee (\exists v, t \xrightarrow{\texttt{val}\ v} \emptyset \wedge k\ v \xrightarrow{l} u)}$$

Fig. 9. Transitions under bind (🦀)

require a branch, while `spinD_nary` is proven to be stuck by induction, since it cannot reach any step.

$$Br^0_S \not\to \qquad Br^0_D \not\to \qquad \texttt{spinD\_nary}\ n \not\to$$

The stepping relation interacts slightly awkwardly with `bind`: indeed, although a unit for `bind`, the `Ret` construct is not inert from the perspective of the LTS. Non `val` transitions can therefore be propagated below the left-hand-side of a `bind`, while a `val` transition in the prefix does not entail the existence of a transition in the `bind`—Figure 9 describes the corresponding lemmas.

We additionally define the traditional *weak transition* $s \overset{l}{\Rightarrow} t$ on the LTS that can perform tau transitions before or after the l transition (and possibly be none 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 2022b], with the exception that we need to work in a Kleene Algebra with a model closed under `equ` rather than `eq`.

4.4 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 4.4 (Bisimulation for CTrees (🦀)). The progress function sb for bisimilarity maps a relation $\mathcal{R}$ over CTrees to the relation such that $sb\ R\ s\ t$ holds if and only if:

$$\forall l\ s',\ s \xrightarrow{l} s' \implies \exists t'.\ s'\ \mathcal{R}\ t' \wedge t \xrightarrow{l} t'$$
$$\text{and conversely}$$
$$\forall l\ t',\ t \xrightarrow{l} t' \implies \exists s'.\ s'\ \mathcal{R}\ t' \wedge s \xrightarrow{l} s'$$

i.e.

$$\begin{array}{ccc} s & sb\ \mathcal{R} & t \\ l\downarrow & & \downarrow l \\ s' & \mathcal{R} & t' \end{array}$$

Bisimilarity, written $s \sim t$, is defined as the greatest fixpoint of sb: $\texttt{sbisim} \triangleq \texttt{gfp}\ sb$.

All the traditional tools surrounding bisimilarity can be transferred to our setup. We omit the details for spacing concerns, but additionally provide:

- weak bisimilarity, written $s \approx t$, derived from the definition of the weak transition (🦀);

$$\frac{x = y}{\texttt{Ret}\ x \sim \texttt{Ret}\ y} \qquad \frac{\forall x,\ h\ x \sim k\ x}{Vis\ e\ h \sim Vis\ e\ k} \qquad \frac{(\forall x, \exists y,\ h\ x \sim k\ y) \wedge (\forall y, \exists x,\ h\ x \sim k\ y)}{Br_S^n\ h \sim Br_S^m\ k}$$

$$\frac{(\forall x, \exists y,\ h\ x \sim k\ y) \wedge (\forall y, \exists x,\ h\ x \sim k\ y)}{Br_D^n\ h \sim_R Br_D^m\ k} \qquad \frac{t \sim u \qquad (\forall x,\ g\ x \sim k\ x)}{t \gg\!= g \sim u \gg\!= k}$$

$$\texttt{Guard}\ t \sim t \qquad \frac{u \nrightarrow}{Br_D^2\ t\ u \sim t} \qquad Br_D^2\ t\ (Br_D^2\ u\ v) \sim Br_D^2\ (Br_D^2\ t\ u)\ v \qquad Br_D^2\ t\ u \sim Br_D^2\ u\ t$$

$$Br_D^2\ t\ t \sim t \qquad Br_D^2\ (Br_D^2\ t\ u)\ v \sim Br_D^3\ t\ u\ v \qquad Br_S^2\ t\ u \sim Br_S^2\ u\ t \qquad Br_S^2\ t\ t \sim \texttt{Step}\ t$$

$$\texttt{Step}\ t \approx t \qquad \texttt{spinD\_nary}\ n \sim \texttt{spinD\_nary}\ m \qquad \frac{(n > 0 \wedge m > 0) \vee (n = m = 0)}{\texttt{spinS\_nary}\ n \sim \texttt{spinS\_nary}\ m}$$

Fig. 10. Elementary equational theory for CTrees (🔥)

- a characterization of the traces (represented as colists) of a CTree, used to define trace-equivalence (written $\equiv_{\texttt{tr}}$) (🔥);
- strong simulations (written $\lesssim$), defined as the greatest fixpoint of the half game for strong bisimulation (🔥).

4.4.1 Core equational theory. Bisimilarity forms an equivalence relation satisfying a collection of primitive laws for CTrees summed up in Figure 10. We use simple inference rules to represent an implication from the premises to the conclusion, and double-lined rules to represent equivalences. Each rule is proved as a lemma with respect to the definitions above.

The first four rules recover some structural reasoning on the syntax of the trees from its semantic interpretation. These rules are much closer to what `eutt` provides by construction for ITrees: leaves are bisimilar if they are equal, and computations performing the same external interaction must remain point-wise bisimilar. Stepping branches, potentially of distinct arity, can be matched one against another if and only if both domains of indexes can be injected into the other to reestablish bisimilarity. In contrast, this condition is sufficient but not necessary for delayed branches, since the points of the continuation structurally immediately accessible do not correspond to accessible states in the LTS. Finally, bisimilarity is a congruence for `bind`.

Another illustration of this absence of equivalence for delayed branching nodes as head constructor is that such a computation may be strongly bisimilar to a computation with a different head constructor. The simplest example is that `sbisim` can ignore (finite numbers of) `Guard` nodes. Another example is that stuck processes behave as a unit for delayed branching nodes. We furthermore obtain the equational theory that we expect for nondeterministic effects. Delayed branching is associative, commutative, idempotent, and can be merged into delayed branching nodes of larger arity w.r.t. `sbisim`.[10] In contrast, stepping branches are only commutative, and almost idempotent, provided we introduce an additional `Step`. This `Step` can in turn be ignored by moving to weak bisimilarity, making stepping branches commutative and properly idempotent; however, it crucially remains *not* associative, a standard fact in process algebra.[11]

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

[10] 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.

[11] This choice would be referred as external in this community

$$\texttt{Ret}\ v \sim_R \texttt{Ret}\ v$$

$$\frac{\forall v,\ R\ (k\ v)\ (k'\ v)}{Vis\ e\ k \sim_R Vis\ e\ k'} \qquad \frac{(\forall x, \exists y,\ R\ (k\ x)\ (k'\ y)) \wedge (\forall y, \exists x,\ R\ (k\ x)\ (k'\ y))}{Br_S^n\ k \sim_R Br_S^m\ k'} \qquad \frac{\forall v,\ R\ (k\ v)\ (k'\ v)}{Br_S^n\ k \sim_R Br_S^n\ k'}$$

$$\frac{(\forall x, \exists y,\ (k\ x) \sim_R (k'\ y)) \wedge (\forall y, \exists x,\ (k\ x) \sim_R (k'\ y))}{Br_D^n\ k \sim_R Br_D^m\ k'} \qquad \frac{t \sim_R u}{\texttt{Guard}\ t \sim_R \texttt{Guard}\ u} \qquad \frac{t\ R\ u}{\texttt{Step}\ t \sim_R \texttt{Step}\ u}$$

Fig. 11. Proof rules for coinductive proofs of `sbisim` (🔥)

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

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

4.4.2 Proof system for bisimulation proofs. As is usual, the laws in Figure 10, 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.

Given a bisimulation candidate R, we write $t \sim_R u$ for $sb\ (\mathtt{t}_{sb}\ R)$: one needs to "play the bisimulation game" by crossing sb, and may rely on R to conclude by coinduction, while furthermore being able to exploit the companion $\mathtt{t}_{sb}$ to leverage sound up-to principles. We depict the main rules we use in Figure 11. These proof rules are notably convenient because they avoid an exponential explosion in the number of cases in our proofs, which otherwise would arise due to the systematic binary split entailed by the natural way to play the bisimulation game. These rules essentially match up counterpart CTree constructors at the level of bisimilarity, but additionally make a distinction as to whether applying the rule soundly acts as playing the game—i.e., the premises refer to R, allowing to conclude using the coinduction hypothesis—or whether they do not—, i.e., the premises still refer to $\sim_R$. The latter situation arises when using the proof rules that strip off delayed branches from the structure of our trees: on either side, they do not entail any step in the corresponding LTSs, but rather correspond to recursive calls to its inductive constructor.

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

Lemma 4.5. *Enhanced coinduction for* `sbisim` (🔥) *The functions* $\textsc{refl}^{up}$, $\textsc{sym}^{up}$, $\textsc{trans}^{up}$, $\textsc{Vis}^{up}$, $\textsc{bind}^{up}(\sim)$, $\textsc{upto}^{up}(\cong)$ *and* $\textsc{upto}^{up}(\sim)$ *provide valid up-to principles for* `sbisim`.

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

```
Definition interp (h : E ⇝ M) : ctree E ⇝ M := fun R ⇒
  iter (fun t ⇒ match t with
  | Ret r   ⇒ ret (inr r)
  | BrD n k ⇒ bind (mBrD n) (fun x ⇒ ret (inl (k x)))
  | BrS n k ⇒ bind (mBrS n) (fun x ⇒ ret (inl (k x)))
  | Vis e k ⇒ bind (h e)    (fun x ⇒ ret (inl (k x)))
  end).
```

Fig. 12. Interpreter for CTrees (class constraints omitted) ( )

4.5 Eliminating nondeterministic stepping choices

Out of programming convenience, CTrees offer both stepping and delayed branching of arbitrary arity. However, as hinted at in Section 3, this is superfluous: a stepping branch can always be simulated by delaying the same choice, and guarding each branch by a unary `Step`. We make this intuition formal by proving that the following combinator, precisely performing this transformation, preserves the behavior of the computation with respect to `sbisim`.

Definition 4.6 (Elimination of nondeterministic stepping choice ( )).

```
Definition BrSElim {E X} (t : ctree E X) : ctree E X :=
  iter (fun t ⇒ match t with
               | Ret r   ⇒ Ret (inr r)
               | brD k   ⇒ x ← Br_D^n ;; Ret (inl (k x))
               | brS k   ⇒ x ← Br_D^n ;; Step (Ret (inl (k x)))
               | Vis e k ⇒ x ← trigger e;; Ret (inl (k x))
               end) t.
```

Lemma 4.7. *Stepping nondeterministic branches can be eliminated ( )* $\quad \forall t,\ \texttt{BrSElim}\ t \sim t$

The proof of this statement is quite illustrative of bisimulation proofs over CTrees. We provide its sketch in the extended version [Chappe et al. 2023] as a means to illustrate the layers of facilities put in place in the formal development to allow for the translation of paper intuitions, most notably the enhanced coinduction principles that the companion allows us to build. It also illustrates the difficulties that the structure raises when bridging the distance between the syntax (the tree) and its implicit semantics (the LTS).

5 INTERPRETATION FROM AND TO CTREES

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

5.1 Interpretation

ITrees support interpretation: provided a *handler* `h:E ⇝ M` implementing its signature of events `E` into a suitable monad `M`, the `(interp h):itree E ⇝ M` combinator provides an implementation of any computation into `M`. The only restriction imposed on the target monad `M` is that it must support its own `iter` combinator, i.e., be iterative, so that the full coinductive nature of the tree can 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`.

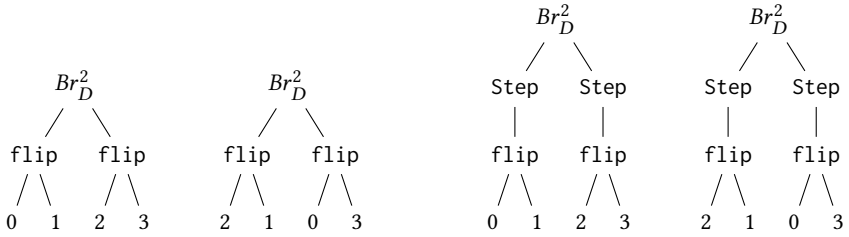


Fig. 13. Two strongly bisimilar trees before interpretation (left), but not after (right)

Unsurprisingly, given their structure, CTrees enjoy their own `interp` combinator. Its definition, provided in Figure 12, is very close to its ITree counterpart. The interpreter relies on the CTree version of `iter`, chaining the implementations of the external events in the process. The target monad must naturally still be iterative, but must also explain how it internalizes branching nodes through the `mBrD` and `mBrS` operations.

Perhaps more surprisingly, the requirement that `interp h` define 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 := ctree F`. The requirement becomes: $\forall t\ u, t \sim u \longrightarrow$ `interp h` $t \sim$ `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 13, where `flip` is the binary event introduced in Section 2.2, fleshes out this intuition. Indeed, both trees are strongly bisimilar: each of them can emit the label `obs flip` *false* by stepping to either the `Ret 0` or `Ret 2` node, or emit the label `obs flip` *true* by stepping to either the `Ret 1` or `Ret 3` node. However, they are strongly bisimilar because the induced LTS processes the question to the environment—`flip`— and its answer—*false*/*true*—in a single step, such that the computations never observe that they have had access to distinct continuations. However, if one were to introduce in the tree a `Step` node before the external events, for instance using the handler `h := fun e ⇒ Step (trigger e)`, a new state allowing for witnessing the distinct continuations would become available in the LTSs, leading to non-bisimilar interpreted trees.

We hence say a handler `h : E ⇝ ctree F` is *simple* if it implements each event `e` either as a pure leaf, i.e., `h e = Ret x`, or simply translates it, i.e., `h e = trigger f`. Similarly, we say that `h : E ⇝ stateT S (ctree F)` is simple if it is point-wise simple. We show that we recover the desired property for the subclass of simple handlers:

Lemma 5.1 (Simple handlers interpret into monad morphisms ( )).

- *If* `h : E ⇝ ctree F` *is simple, then* $\forall t\ u,\ t \sim u \longrightarrow$ `interp h` $t \sim$ `interp h` u.
- *If* `h : E ⇝ stateT (ctree F)` *is simple, then* $\forall t\ u,\ t \sim u \longrightarrow \forall s,$ `interp h` $t\ s \sim$ `interp h` $u\ s$.

The latter result 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 $p_3 \equiv$ `br` p_2 `or` p_3 mentioned in Section 2.2 ( ).

5.2 Refinement

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

```
Definition refine (h : bool → ∀(n:nat), M (fin n)) : ctree E ⇝ M := fun R ⇒
iter (fun t ⇒ match t with
| Ret r  ⇒ ret (inr r)
| BrD n k ⇒ bind (h false n) (fun x ⇒ ret (inl (k x)))
| BrS n k ⇒ bind (h true n) (fun x ⇒ ret (inl (k x)))
| Vis e k ⇒ bind (mtrigger e) (fun x ⇒ ret (inl (k x)))
end).
```

Fig. 14. Refiner for CTrees (class constraints omitted)

```
Definition refine_cst (h : bool → ∀(n:nat), fin (S n)) : ctree E ⇝ ctree E :=
refine (fun b n ⇒ match n with
| O ⇒ stuckD
| S n ⇒ if b then Step (Ret (h b n)) else Guard (Ret (h b n))
end).

Definition refine_state (h : St → ∀(b:bool) (n:nat), St * fin (S n))
: ctree E ⇝ stateT St (ctree E) :=
refine (fun b n s ⇒ match n with O ⇒ stuckD | S n ⇒ Ret (h s b n) end).
```

Fig. 15. Constant and stateful refinements

We provide, to this end, a new combinator, `refine`, defined in Figure 14. Similar to `interp`, it takes as an argument a handler specifying how to implement our object of interest into a monad `M`. Since here we are interested in the implementation of branching nodes, the handler describes how to monadically choose a branch—the boolean flag allows for different behaviors between delayed and stepping branches.[12] As usual, the implementation monad must be iterative, but this time it must also specify how to re-embed the external events using an operation called `mtrigger` (a concept already used in Yoon et al. [Yoon et al. 2022a]).

As hinted at by the combinator's name, the source program should be able to simulate the refined program. Fixing `M` to `ctree F`, this is expressed as $\forall t$, `refine h` $t \lesssim t$. However, one cannot hope to obtain such a result for an arbitrary `h`, because it could implement internal branches with an observable computation that can't be simulated by t. We hence provide two illustrative families of well-behaved refinements, depicted on Figure 15. Constant refinements, implemented into `ctree E`, lift a function systematically, returning the same branch given the nature and arity of the branching node. Stateful refinements carry a memory into a piece of state `St` and implement the refinement into `stateT St (ctree E)` by using the current state to pick a branch, and update the state. A particular example of such a stateful refinement is a round-robin scheduler.

Lemma 5.2. *The constant and stateful refinements are proper refinements* [13]

$$\forall h\ t,\ \texttt{refine\_cst}\ h\ t \lesssim t \qquad \text{and} \qquad \forall h\ t\ s,\ \texttt{refine\_state}\ h\ t\ s \lesssim t$$

Finally, the shallow nature of CTrees also offers testing opportunities. Xia et al. [Xia et al. 2020] describe how external events such as IO interactions can alternatively be implemented in `OCaml` and linked against at extraction. Similarly, we demonstrate on `ImpBr` how to execute a CTree

[12] This definition becomes even more uniform with `interp` once we allow for parameterized, arbitrary indexing of branching nodes, as in the enhanced CTrees branch of our development.

[13] The second theorem actually cannot be stated in the main development provided as artifact: the return types of the trees are not identical — the refined tree computes additionally a final state. We prove it in the enhanced CTrees branch with support for heterogeneous relations and arbitrary relations on labels.

```
Definition inject {E} : itree E ⤳ ctree E := interp (fun e ⇒ trigger e).
Definition internalize {E} : ctree (Choose +' E) ⤳ ctree E :=
  interp (fun e ⇒ match e with | inl1 (choose n) ⇒ Br^n_S | inr1 e ⇒ trigger e).
Definition embed {E} : itree (Choose +' E) ⤳ ctree E :=
  fun _ t ⇒ internalize (inject t).
```

Fig. 16. Implementing external branching events into the CTree monad (🔥)

by running an impure refinement implemented in `OCaml` by picking random branches along the execution.

5.3 ITree embedding

We have used CTrees directly as a domain to represent the syntax of `ImpBr`, as well as in our case studies (see Section 6 and 7). CTrees can, however, fulfill their promise sketched in Section 2, and be used as a domain to host the monadic implementation of external representations of nondeterministic events in an ITree.

To demonstrate this approach, we consider the family of events `choose (n: nat) : Choose (fin n)`, and aim to define an operator `embed` taking an ITree computation modeling nondeterministic branching using these events, and implementing them as stepping branches into a CTree. This operator, defined in Figure 16, 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 `Tau` 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 `Choose` event, using the isomorphic stepping branch. The resulting embedding forms a monad morphism transporting `eutt` ITrees into `sbisim` CTrees:

Lemma 5.3. `embed` *respects* `eutt` (🔥) $\quad \forall t\ u, \texttt{eutt}\ t\ u \implies embed(t) \sim embed(u)$

The proof of this theorem highlights how `Tau` nodes in ITrees 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 `Tau`—the corecursion is completely oblivious to these nodes in the structures. Corecursively, however, `Tau` nodes can be matched *symmetrically*—a construction that is useful in exactly one case, namely to relate the silently spinning computation, $\texttt{Tau}^\omega$, to itself. From the CTrees perspective, recursing in `euttF` corresponds to recursion in the definition of the LTS: `Tau` nodes are `Guard` nodes. But corecursing corresponds to a step in the LTS: $\texttt{Tau}^\omega$ corresponds to $\texttt{Step}^\omega$. ITrees' `Tau` thus corresponds to either a `Guard` or a `Step`. Nondeterminism forces us to separate both concepts, as whether a node in the tree constitutes an accessible state in the LTS becomes semantically relevant. We obtain two "dual" notions that have no equivalent in ITrees: we may have finite amounts of asymmetric corecursive choices, i.e., finitely many `Steps`, and structurally infinite stuck processes, i.e., $\texttt{Guard}^\omega$.

In the proof of Lemma 5.3, this materializes by the fact that an induction on `euttF` leaves us disappointed in the symmetric `Tau` 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 $\texttt{Tau}^\omega$, 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 non-deterministic 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 `Tau`) 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 would 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 4.4 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 implementing it to future work.

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

6 CASE STUDY: A MODEL FOR CCS

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 semantics for Milner's Calculus of Communicating Systems (`ccs`) [Milner 1989]. The results we obtain—the usual algebra, up-to principles, and precisely the same equivalence relation as the usual operational-based strong bisimulation—are standard, per se, but they are all established by exploiting the generic notion of bisimilarity of CTrees. The result is a shallowly embedded model for `ccs` in Coq that could be easily, and modularly, combined with other language features.

6.1 Syntax and operational semantics

The syntax and operational semantics of `ccs` are shown in Figure 17. 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 τ, which represents an internal action.

The standard operational semantics, shown in the figure, is expressed as a labeled transition system, where states are terms P and labels are actions l. The `ccs` operators are the following: 0 is the process with no behavior. A prefix process $l \cdot P$ emits an action l and then becomes the process P. The internal choice $P \oplus Q$ behaves either like the process P or like the process Q, in the same fashion as the BrInternal semantics for `br` in Section 2.2. The parallel composition of two processes $P \parallel Q$ interleaves the behavior of the two processes, while allowing the two processes

$$l ::= \tau \mid c \mid \bar{c} \qquad P ::= 0 \mid l \cdot P \mid P \oplus Q \mid P \parallel Q \mid \nu c \cdot P \mid !P$$

$$\frac{}{a \cdot P \xrightarrow{a}_{\text{ccs}} P} \qquad \frac{P \xrightarrow{l}_{\text{ccs}} P'}{P \oplus Q \xrightarrow{l}_{\text{ccs}} P'} \qquad \frac{Q \xrightarrow{l}_{\text{ccs}} Q'}{P \oplus Q \xrightarrow{l}_{\text{ccs}} Q'} \qquad \frac{P \xrightarrow{l}_{\text{ccs}} P'}{P \parallel Q \xrightarrow{l}_{\text{ccs}} P \parallel' Q}$$

$$\frac{Q \xrightarrow{l}_{\text{ccs}} Q'}{P \parallel Q \xrightarrow{l}_{\text{ccs}} P \parallel Q'} \qquad \frac{P \xrightarrow{c}_{\text{ccs}} P' \quad Q \xrightarrow{\bar{c}}_{\text{ccs}} Q'}{P \parallel Q \xrightarrow{\tau}_{\text{ccs}} P' \parallel Q'} \qquad \frac{P \xrightarrow{l}_{\text{ccs}} P' \quad l \notin \{c, \bar{c}\}}{\nu c \cdot P \xrightarrow{l}_{\text{ccs}} \nu c \cdot P'} \qquad \frac{P \parallel !P \xrightarrow{l}_{\text{ccs}} P'}{!P \xrightarrow{l}_{\text{ccs}} P'}$$

Fig. 17. Syntax for ccs (🔥) and its operational semantics (🔥)

$$\overline{0} \triangleq \emptyset \qquad a \bar{\cdot} p \triangleq \texttt{trigger}\ a\ ;;\ p \qquad p \overline{\oplus} q \triangleq Br^2_D\ p\ q \qquad \bar{!}p \triangleq p \overline{\parallel} !p$$

$$\nu c \bar{\cdot} P \triangleq \texttt{interp}\ \texttt{h\_new}\ c\ P \qquad \text{where } \texttt{h\_new}\ c\ e = \begin{cases} \emptyset & \text{if } e = act\ c \text{ or } e = act\ \bar{c} \\ \texttt{trigger}\ e & \text{otherwise} \end{cases}$$

$$p \overline{\parallel} q \triangleq \texttt{cofix}\ F\ p\ q \cdot Br^3_D\ \begin{array}{l} (p' \leftarrow \texttt{head}\ p\ ;;\ \texttt{actL}\ F\ q\ p') \\ (q' \leftarrow \texttt{head}\ q\ ;;\ \texttt{actR}\ F\ p\ q') \\ (p' \leftarrow \texttt{head}\ p\ ;;\ q' \leftarrow \texttt{head}\ q\ ;;\ actLR\ F\ p'\ q') \end{array}$$

$$\begin{array}{ll} \texttt{actL}\ F\ q\ (Br^n_S\ k) \triangleq Br^n_S\ (\lambda i \cdot F\ (k\ i)\ q) & \texttt{actR}\ F\ p\ (Br^n_S\ k) \triangleq Br^n_S\ (\lambda i \cdot F\ p\ (k\ i)) \\ \texttt{actL}\ F\ q\ (Vis\ e\ k) \triangleq Vis\ e\ (\lambda i \cdot F\ (k\ i)\ q) & \texttt{actR}\ F\ p\ (Vis\ e\ k) \triangleq Vis\ e\ (\lambda i \cdot F\ p\ (k\ i)) \end{array}$$

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

Fig. 18. Denotational model for ccs using $\text{ccs}^{\#}$ as a domain (🔥)

to communicate. If the process P emits a name c and the process Q emits its co-name $\bar{c}$, then the two processes can progress simultaneously and the parallel composition emits an internal action τ. Channel restriction $\nu c \cdot P$ prevents the process P from emitting an action c or $\bar{c}$: the operational rule states that any emission of another action is allowed. Finally the replicated process $!P$ behaves as an unbounded replication of the process P. Operationally, $!P$ has the behavior of $P \parallel !P$.

6.2 Model

We define a denotational model for ccs using `ctree actE 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 `unit` for answer: `Inductive actE ::= | act a : actE unit`. Figure 18 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 `h_new`, a handler that does the substitution.

Parallel composition is more intricate, as the operator requires significant introspection of the composed terms. The traditional operational semantics of ccs is not explicitly constructive: each of the three reduction rules depends on the existence of specific transitions in the sub-processes. We perform this necessary introspection, in a constructive way, defining the tree as an explicit cofixpoint, and using the head operator introduced in Section 3.2. While the operation head p precisely captures the desired set of actions that p may perform, computing this set could, in general, silently diverge. We therefore cannot bluntly initiate the computation by sequencing the heads of p and q, as divergence in the former may render inaccessible valid transitions in the latter.[14] 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 19 shows the CTree resulting from $p \overline{\|} q$.

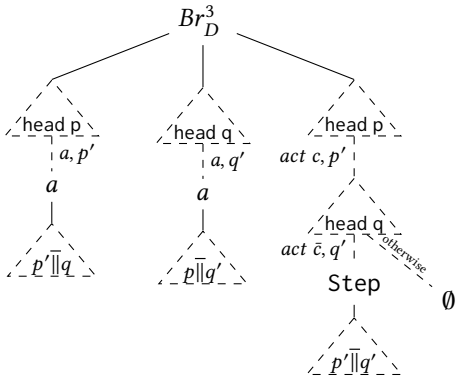


Fig. 19. Depiction of the tree resulting from $p \overline{\|} q$

The last operator to consider is the replication $\bar{!}$. In theory, it could be expressed in terms of parallel composition directly, as the cofix $\bar{!}p \triangleq p \overline{\|} \bar{!}p$. Unfortunately, although it is sound, defining the $\bar{!}$ operator in this way is too involved for Coq's syntactic criterion on cofixes to recognize that the corecursive call is guarded under $\overline{\|}$. To circumvent this difficulty, we use an auxiliary operator, $p \overline{\|} !q$, capturing the parallel composition of a process p with a replicated process $!q$. It 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 p', before being composed in parallel with p; the creation of a copy of q synchronizing with p; or the emission of two copies of q synchronizing one with another. We finally define the replication operator as $\bar{!}p \triangleq p \overline{\|} !p$. We omit the formal definition here, since it is similar to that for $\overline{\|}$, but point out the interested reader to its formal counterpart ( ).

With these tools at hand, the model $[\![\cdot]\!] : \texttt{ccs} \to \texttt{ccs}^{\#}$ is defined by recursion on the syntax.

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

[14] 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.

We also prove an illustrative collection of expected equations satisfied by our operators ( ):

$$p \overline{\oplus} q \sim q \overline{\oplus} p \qquad p \overline{\oplus} (q \overline{\oplus} r) \sim (p \overline{\oplus} q) \overline{\oplus} r \qquad p \overline{\oplus} \overline{0} \sim p \qquad p \overline{\oplus} p \sim p$$
$$p \overline{\|} \overline{0} \sim p \qquad p \overline{\|} q \sim q \overline{\|} p \qquad p \overline{\|} (q \overline{\|} r) \sim (p \overline{\|} q) \overline{\|} r$$

To facilitate these proofs, we first prove sound up-to principles at the level of ccs for each constructor: strong bisimulation up-to $c^{\overline{\cdot}}[\cdot]$, $[\cdot]\overline{\oplus}[\cdot]$, $[\cdot]\overline{\|}[\cdot]$, $\overline{!}[\cdot]$, and $\nu c^{\overline{\cdot}}[\cdot]$ are all valid principles, allowing us to rewrite sbisim under semantic contexts during bisimulation proofs. Additionally to these language-level up-to principles, we inherit the ones generically supported by sbisim (Lem. 4.5).

6.3 Equivalence with the operational strong bisimilarity

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

Definition 6.1 (Strong bisimulation between ccs *and* $\text{ccs}^{\#}$*.).* A relation $\mathcal{R} : rel(\text{ccs}, \text{ccs}^{\#})$ is a strong bisimulation if and only if, for any label l, ccs term P, and $\text{ccs}^{\#}$ tree q.

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

$$\begin{array}{ccc} P & \mathcal{R} & q \\ l \big\downarrow_{\text{ccs}} & & \big\downarrow \bar{l} \\ P' & \mathcal{R} & q' \end{array}$$

Lemma 6.2 ( ). *The relation* $R \triangleq \{(P, q) \mid [\![P]\!] \sim q\}$ *is a strong bisimulation.*

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

Lemma 6.3 ( ). $\forall P\ Q,\ [\![P]\!] \sim [\![Q]\!]$ *iff* $P \sim_{\text{ccs}} Q$

7 CASE STUDY: MODELING COOPERATIVE MULTITHREADING

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

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

The command fork forks two copies of the currently running program, the first of which first runs $c1$ and the second of which first runs $c2$, before they each proceed with the rest of the program. Importantly, $c2$ retains control, and $c1$ will only execute when $c2$ voluntarily yields control or ends. This yielding of control is achieved by the yield statement, which signals that the current thread stops and lets a new thread run—possibly the same one again.

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

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

forks two copies of the program, with the "main" thread immediately yielding, allowing for either thread to run next. Its semantics is to first spawn a thread for $x ::= 1$, then have the main thread reach the yield, giving $x ::= 1$ a chance to run. Assuming the spawned thread goes next, it runs in

sequence $x ::= 1$ and $y ::= x$, after which the main thread recovers control and finishes its execution. $y ::= x$ is part of both threads and is thus executed twice, after each assignment to x.

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

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

spawns two threads that set x to different values, and terminates. The two spawned threads can then be scheduled in either order, resulting in $x = 1$ or $x = 2$ in the final state. This constraint could be syntactically enforced in the language if relevant. Alternatively, one can use the command `while` *true* `do yield` to "terminate" a thread; for instance to prevent the first thread above from reaching $y ::= x$. With fancier encodings using reserved shared-variables, nested "joins" and other synchronization operations can be modeled. In this case study, we do not concern ourselves with such extensions, and restrict ourselves to the formalization of the syntax described above.

7.1 Model

As with the approach described in Section 2, the semantics of the language is defined in two stages. First, we represent statements as computations of type `ctree (YieldE + SpawnE + MemE) unit`, where `YieldE` and `SpawnE` are used to represent `yield` and `fork` respectively. The remaining statements of `imp` are defined in a standard way—we omit them. The new event families are defined as follows:

```
Variant YieldE : Type → Type :=          Variant SpawnE : Type → Type :=
| Yield : YieldE unit.                   | Spawn : SpawnE bool.
```

`Yield` carries no additional information and acts purely as a signal to yield control, and `Spawn` 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. Formally, these events are used to denote the corresponding statements:

$$[\![\texttt{yield}]\!] \triangleq \texttt{trigger}\ (\texttt{Yield}) \qquad [\![\texttt{fork}\ c1\ c2]\!] \triangleq \quad b \leftarrow \texttt{trigger}\ (\texttt{Spawn})\ ;;\ \text{if}\ b\ \text{then}\ [\![c1]\!]\ \text{else}\ [\![c2]\!]$$

The second stage is more complex than the simple stateful interpretation of standard `imp`: we need to interpret the concurrency-related events. In particular, since the semantics of these events involves scheduling other threads that work together, we cannot hope to use `interp`, as is done with memory events—this situation is similar to $\overline{\|}$ for `ccs`, in both cases the combinator cannot be implemented as a simple fold. Instead, we define a custom interpreter for `Spawn` and `Yield` events, namely a scheduler operating over a thread pool of threads, each of which is a CTree that may contain `Spawn` and `Yield` events, and producing a final CTree devoid of these concurrency events.

As is usual with cooperative scheduling, the thread pool, encoded as a vector, contains a designated *active* thread, i.e., the currently running thread. 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 scheduler, `schedule`, is defined formally in Figure 20. References to `schedule` in its body should be interpreted as corecursive calls—we abuse notations to lighten the presentation. It takes two arguments: the thread pool, and the index of the active thread in the pool. If no thread is active, the second argument is empty, written $\lfloor\text{-}\rfloor$ (we use an option datatype in `Coq`). If there is an active thread, $\lfloor i \rfloor$, the schedule makes progress in that thread. Otherwise, `schedule` nondeterministically chooses the next thread to run. If the thread pool is empty, then `schedule` emits a returning CTree. In more detail, there are several ways the active thread may progress. If the thread performs a branching or a memory operation, the schedule returns a CTree that also performs the same

$$
\begin{array}{rcl}
\text{schedule } v_0\ \lfloor\text{-}\rfloor & \triangleq & \texttt{ret}\ (\,) \\
\text{schedule } v_n\ \lfloor\text{-}\rfloor & \triangleq & Br_S^n\ (\lambda i \cdot \text{schedule } v_n\ \lfloor i \rfloor) \qquad \text{for } n > 0 \\
\text{schedule } v_n\ \lfloor i \rfloor & \triangleq & \\
\quad \text{if } v[i] = & & \text{then} \\
\quad\quad \texttt{ret}\ (\,) & & Br_D^1\ (\lambda\_ \cdot \text{schedule } v[-i]_{n-1}\ \lfloor\text{-}\rfloor) \\
\quad\quad Br_S^m\ k & & Br_S^m\ (\lambda i' \cdot \text{schedule } v[i \mapsto k\ i']_n\ \lfloor i \rfloor) \\
\quad\quad Br_D^m\ k & & Br_D^m\ (\lambda i' \cdot \text{schedule } v[i \mapsto k\ i']_n\ \lfloor i \rfloor) \\
\quad\quad Vis\ \texttt{Yield}\ k & & Br_D^1\ (\lambda\_ \cdot \text{schedule } v[i \mapsto k\ (\,)]_n\ \lfloor\text{-}\rfloor) \\
\quad\quad Vis\ \texttt{Spawn}\ k & & Br_S^1\ (\lambda\_ \cdot \text{schedule } (k\ \mathit{true} :: v[i \mapsto k\ \mathit{false}])_{n+1}\ \lfloor i+1 \rfloor) \\
\quad\quad Vis\ \texttt{e}\ k & & Vis\ e\ (\lambda x \cdot \text{schedule } (v[i \mapsto k\ x])_n\ \lfloor i \rfloor)
\end{array}
$$

Fig. 20. The definition of schedule (🔥)

operation, and in each branch replaces the active thread (index i) in the thread pool with its continuation. If the active thread returns, it is removed from the thread pool; there is no active thread after the removal. If the thread yields, then schedule removes the `Yield` and starts scheduling with no active thread (which will nondeterminstically pick a new active thread, potentially the same one again). If the thread spawns a new thread, then schedule adds the *true* branch of that thread to the thread pool, updates the active thread to the *false* branch, and continues running on the *false* branch (note that this shifts the active thread from index i to $i + 1$).

Using schedule, we can interpret the `Yield` and `Spawn` events in concert, allowing us to eliminate them from the CTrees. We denote this "scheduled" semantics by $\mathcal{S}[\![p]\!] \triangleq$ schedule $[[\![p]\!]]_1\ \lfloor\text{-}\rfloor$: `ctree MemE unit`: it denotes a CTree with only memory events. As described in Section 2.1, the final step of the second stage of modeling our `imp` programs can use a handler for the memory events to obtain a computation in the state monad of type `stateT mem (ctree voidE) unit`.

7.2 Equational theory

The model described in Section 7.1 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 (🔥):

$$\mathcal{S}[\![\texttt{fork}\ c\ \texttt{skip}]\!] \approx \mathcal{S}[\![\texttt{yield}; c]\!] \approx \mathcal{S}[\![c]\!]$$

We use weak rather than strong bisimilarity since schedule can introduce stepping branches when interpreting the two concurrency events. We emphasize that these equations are not compositional, they only hold in the absence of additional concurrent threads, hence why `yield` behaves as a no op: the monadic equivalence we obtain is a congruence for the source language after the first stage of interpretation, but not after the second stage.

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

Lemma 7.1 (schedule preserves $\sim$ (🔥)). *If the delayed branches of every element of vectors v_n and w_n have arity less than 2, and the elements of both vectors are strongly bisimilar up to a permutation ρ, then* schedule $v_n\ \lfloor i \rfloor \sim$ schedule $w_n\ \lfloor \rho\ i \rfloor$.

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

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

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

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

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 7.1 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 (🦀):

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

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 5.1, which allows us to transport equations made before interpreting state events into computations in the state monad after interpetation.

8 RELATED WORK

Since Milner's seminal work on ccs [Milner 1989] and the π-calculus [Milner et al. 1992], process algebras have been the topic of a vast literature [Bergstra et al. 2001]. We mention only a few parts of it that are most relevant to our work. In the Coq realm, Lenglet et al. [Lenglet and Schmitt 2018] have formalized HOπ, a minimal π-calculus, notably exhibiting the difficulty inherent to the formal treatment of name extrusion. Beyond its formalization, dealing with scope extrusion as part of a compositional semantics is known to be a challenging problem [Crafa et al. 2012; Cristescu et al. 2013]. By restricting to ccs in our case study, we have side-stepped this difficulty. Foster et al. [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 [Brookes 2002] by providing a model based on the compositional construction of infinite sets of traces: CTrees offer a complementary coinductive model to this more set-theoretic approach. Brookes tackles questions of fairness, an avenue that we have not yet explored in our setup.

Formal semantics for nondeterminism are especially relevant when dealing with low-level concurrent semantics. In shared-memory-based programming languages, rather than message passing ones, concurrency gives rise to the additional challenge of modeling their memory models, a topic that has received considerable attention. Understanding whether monadic approaches such as the one proposed in this paper are viable to tackle such models vastly remains to be investigated. Early suggestions that they may include Lesani et al. [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. 2021, 2014] framework appear less likely to transpose to our constructive setup.

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

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

9 CONCLUSION AND PERSPECTIVES

We have introduced CTrees, a model for nondeterministic, recursive, and impure programs in `Coq`. Inspired by ITrees, we have introduced two kinds of nondeterministic branching nodes, and designed a toolbox to reason about these new computations. In addition to the `Coq` library presented in this paper, we have developed an enhanced implementation (with heavier notations, but more expressive) supporting arbitrary indexing of branches and heterogeneous bisimulations between CTrees. We have illustrated the expressiveness of the framework through two significant case studies. Both nonetheless offer avenues for further work, notably through an extension of `ccs` to name passing *à la* π-calculus, and to further extend the equational theory for cooperative multithreading that we currently support.

The restriction imposed on handlers in Section 5.1 suggests another perspective: rather than requiring handlers to implement events in a non-observable way, we could prevent the programs in Figure 13 from being bisimilar in the first place. We conjecture that this could be achieved by tweaking the LTS so that external events induce two steps: the question to the environment first, and the answer second, resulting in a finer grained bisimilarity. Whether such a finer bisimilarity would still allow us to equate all the programs we want needs further investigation.

Data-Avaibility Statement

A snapshot of the development presented in this paper is archived on Zenodo [Chappe et al. 2022].

Acknowledgements

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

REFERENCES

Martín Abadi and Gordon D Plotkin. 2010. A model of cooperative threads. *Logical Methods in Computer Science* 6, 4 (2010), 1–39. https://doi.org/10.2168/LMCS-6(4:2)2010

Sampson Abramsky and Paul-Andreé Melliès. 1999. Concurrent Games and Full Completeness. In *Proceedings of the 14th Annual ACM/IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press. https://doi.org/10.1109/LICS.1999.782638

Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. 2021. Armed Cats: Formal Concurrency Modelling at Arm. *ACM Trans. Program. Lang. Syst.* 43, 2 (2021), 8:1–8:54. https://doi.org/10.1145/3458926

Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding cats: modelling, simulation, testing, and data-mining for weak memory. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 40. https://doi.org/10.1145/2594291.2594347

Thorsten Altenkirch, Nils Anders Danielsson, and Nicolai Kraus. 2017. Partiality, Revisited. In *Foundations of Software Science and Computation Structures*, Javier Esparza and Andrzej S. Murawski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 534–549. https://doi.org/10.1007/978-3-662-54458-7_31

J.A. Bergstra, A. Ponse, and S.A. Smolka (Eds.). 2001. *Handbook of Process Algebra*. Elsevier Science. https://doi.org/10.1016/B978-044482830-9/50038-2

B. Bloom, S. Istrail, and A. R. Meyer. 1988. Bisimulation Can't Be Traced. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 229–239. https://doi.org/10.1145/73560.73580

Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I Pun, Silvia Lizeth Tapia Tarifa, Tobias Wrigstad, and Albert Mingkun Yang. 2015. Parallel Objects for Multicores: A Glimpse at the Parallel Language Encore. In *Formal Methods for Multicore Programming - 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Bertinoro, Italy, June 15-19, 2015, Advanced Lectures (Lecture Notes in Computer Science, Vol. 9104)*, Marco Bernardo and Einar Broch Johnsen (Eds.). Springer, 1–56. https://doi.org/10.1007/978-3-319-18941-3_1

Stephen D. Brookes. 2002. Traces, Pomsets, Fairness and Full Abstraction for Communicating Processes. In *Proc. 13th Intl. Conf. on Concurrency Theory (CONCUR 2002) (LNCS, Vol. 2421)*, Lubos Brim, Petr Jancar, Mojmír Kretínský, and Antonín Kucera (Eds.). Springer, Berlin Heidelberg, 466–482. https://doi.org/10.1007/3-540-45694-5_31

Venanzio Capretta. 2005. General Recursion via Coinductive Types. *Logical Methods in Computer Science* 1, 2 (2005), 1–18. https://doi.org/10.2168/LMCS-1(2:1)2005

Simon Castellan, Pierre Clairambault, Silvain Rideau, and Glynn Winskel. 2017. Games and Strategies as Event Structures. *Log. Methods Comput. Sci.* 13, 3 (2017). https://doi.org/10.23638/LMCS-13(3:35)2017

Nicolas Chappe, Paul He, Ludovic Henrio, Yannick Zakowski, and Steve Zdancewic. 2022. *Choice Trees: Representing Nondeterministic, Recursive, and Impure Programs in Coq*. https://doi.org/10.5281/zenodo.7227966

Nicolas Chappe, Paul He, Ludovic Henrio, Yannick Zakowski, and Steve Zdancewic. 2023. Choice Trees: Representing Nondeterministic, Recursive, and Impure Programs in Coq — extended version. http://arxiv.org/abs/2211.06863

Silvia Crafa, Daniele Varacca, and Nobuko Yoshida. 2012. Event structure semantics of parallel extrusion in the pi-calculus. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 7213 LNCS. 225–239. https://doi.org/10.1007/978-3-642-28729-9_15

Ioana Cristescu, Jean Krivine, and Daniele Varacca. 2013. A Compositional Semantics for the Reversible π-Calculus. In *Proceedings - Symposium on Logic in Computer Science*. 388–397. https://doi.org/10.1109/LICS.2013.45

Crystal Chang Din, Reiner Hähnle, Einar Broch Johnsen, Ka I. Pun, and Silvia Lizeth Tapia Tarifa. 2017. Locally Abstract, Globally Concrete Semantics of Concurrent Programming Languages. In *Automated Reasoning with Analytic Tableaux and Related Methods*, Renate A. Schmidt and Cláudia Nalon (Eds.). Springer International Publishing, Cham, 22–43. https://doi.org/10.1007/978-3-319-66902-1_2

Crystal Chang Din, Reiner Hähnle, Ludovic Henrio, Einar Broch Johnsen, Violet Ka I Pun, and Silvia Lizeth Tapia Tarifa. 2022. LAGC Semantics of Concurrent Programming Languages. https://doi.org/10.48550/ARXIV.2202.12195

Simon Foster, Chung-Kil Hur, and Jim Woodcock. 2021. Formally Verified Simulations of State-Rich Processes Using Interaction Trees in Isabelle/HOL. In *32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference (LIPIcs, Vol. 203)*, Serge Haddad and Daniele Varacca (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 20:1–20:18. https://doi.org/10.4230/LIPIcs.CONCUR.2021.20

Robert Harper. 2016. *Practical Foundations for Programming Languages* (2 ed.). Cambridge University Press. https://doi.org/10.1017/CBO9781316576892

Ludovic Henrio, Eric Madelaine, and Min Zhang. 2016. A Theory for the Composition of Concurrent Processes. In *Formal Techniques for Distributed Objects, Components, and Systems*, Elvira Albert and Ivan Lanese (Eds.). Springer International Publishing, Cham, 175–194. https://doi.org/10.1007/978-3-319-39570-8_12

Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. 2013. The Power of Parameterization in Coinductive Proof. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. ACM, New York, NY, USA, 193–206. https://doi.org/10.1145/2429069.2429093

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 175–189. https://doi.org/10.1145/3009837.3009850

James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252

Oleg Kiselyov and Hiromi Ishii. 2015. Freer monads, more extensible effects. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*. 94–105. https://doi.org/10.1145/2804302.2804319

Jérémie Koenig and Zhong Shao. 2020. Refinement-Based Game Semantics for Certified Abstraction Layers. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science* (Saarbrücken, Germany) *(LICS '20)*. Association for Computing Machinery, New York, NY, USA, 633–647. https://doi.org/10.1145/3373718.3394799

Leonidas Lampropoulos and Benjamin C. Pierce. 2018. *QuickChick: Property-Based Testing in Coq*. Electronic textbook. https://softwarefoundations.cis.upenn.edu/qc-current/index.html

Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. 2020. Promising 2.0: global optimizations in relaxed memory concurrency. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 362–376. https://doi.org/10.1145/3385412.3386010

Sergueï Lenglet and Alan Schmitt. 2018. HOπ in Coq. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, June Andronick and Amy P. Felty (Eds.). ACM, 252–265. https://doi.org/10.1145/3167083

Mohsen Lesani, Li-yao Xia, Anders Kaseorg, Christian J. Bell, Adam Chlipala, Benjamin C. Pierce, and Steve Zdancewic. 2022. C4: verified transactional objects. *Proc. ACM Program. Lang.* 6, OOPSLA (2022), 1–31. https://doi.org/10.1145/3527324

Thomas Letan, Yann Régis-Gianas, Pierre Chifflier, and Guillaume Hiet. 2018. Modular Verification of Programs with Effects and Effect Handlers in Coq. In *Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings*. 338–354. https://doi.org/10.1007/978-3-319-95582-7_20

Kenji Maillard, Cătălin Hriţcu, Exequiel Rivas, and Antoine Van Muylder. 2020. The next 700 Relational Program Logics. *Proceedings of the ACM on Programming Languages* 4, POPL, Article 4 (2020), 33 pages. https://doi.org/10.1145/3371072

Paul-André Melliès and Samuel Mimram. 2007. Asynchronous Games: Innocence Without Alternation. In *CONCUR 2007 – Concurrency Theory*, Luís Caires and Vasco T. Vasconcelos (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 395–411. https://doi.org/10.1007/978-3-540-74407-8_27

Robin Milner. 1989. *Communication and Concurrency*. Prentice-Hall, Inc., USA.

Robin Milner, Joachim Parrow, and David Walker. 1992. A calculus of mobile processes, I. *Information and Computation* 100, 1 (1992), 1–40. https://doi.org/10.1016/0890-5401(92)90008-4

Arthur Oliveira Vale, Paul-André Melliès, Zhong Shao, Jérémie Koenig, and Léo Stefanesco. 2022. Layered and Object-Based Game Semantics. *Proc. ACM Program. Lang.* 6, POPL, Article 42 (jan 2022), 32 pages. https://doi.org/10.1145/3498703

Damien Pous. 2007. Complete Lattices and Up-To Techniques. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 351–366. https://doi.org/10.1007/978-3-540-76637-7_24

Damien Pous. 2016. Coinduction All the Way Up. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science* (New York, NY, USA) *(LICS '16)*. Association for Computing Machinery, New York, NY, USA, 307–316. https://doi.org/10.1145/2933575.2934564

Damien Pous. 2022a. *The coq-coinduction library*. https://github.com/damien-pous/coinduction

Damien Pous. 2022b. *The coq-coinduction library: examples*. https://github.com/damien-pous/coinduction-examples

Silvain Rideau and Glynn Winskel. 2011. Concurrent Strategies. In *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada*. IEEE Computer Society, 409–418. https://doi.org/10.1109/LICS.2011.13

Davide Sangiorgi. 1998. On the bisimulation proof method. *Mathematical Structures in Computer Science* 8, 5 (1998), 447–479. https://doi.org/10.1017/S0960129598002527

Davide Sangiorgi and Jan Rutten. 2012. *Advanced Topics in Bisimulation and Coinduction* (2nd ed.). Cambridge University Press, USA.

Davide Sangiorgi and David Walker. 2001. *The π-calculus* (1st ed.). Cambridge University Press, USA.

Jaroslav Sevcík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM* 60, 3 (2013), 22:1–22:50. https://doi.org/10.1145/2487241.2487248

M.B. Smyth. 1976. Powerdomains. In *Mathematical Foundations of Computer Science (Lecture Notes in Computer Science, Vol. 4)*. Springer. https://doi.org/10.1007/3-540-07854-1_226

The Coq Development Team. 2022. *The Coq Proof Assistant.* https://doi.org/10.5281/zenodo.5846982

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction Trees. *Proceedings of the ACM on Programming Languages* 4, POPL (2020). https://doi.org/10.1145/3371119

Irene Yoon, Yannick Zakowski, and Steve Zdancewic. 2022a. Formal Reasoning About Layered Monadic Interpreters. *Proceedings of the ACM on Programming Languages* 6, ICFP (2022). https://doi.org/10.1145/3547630

Irene Yoon, Yannick Zakowski, and Steve Zdancewic. 2022b. Formal reasoning about layered monadic interpreters. *Proc. ACM Program. Lang.* 6, ICFP (2022), 254–282. https://doi.org/10.1145/3547630

Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. 2021. Modular, Compositional, and Executable Formal Semantics for LLVM IR. *Proc. ACM Program. Lang.* 5, ICFP, Article 67 (aug 2021), 30 pages. https://doi.org/10.1145/3473572

Yannick Zakowski, Paul He, Chung-Kil Hur, and Steve Zdancewic. 2020. An Equational Theory for Weak Bisimulation via Generalized Parameterized Coinduction. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP).* https://doi.org/10.1145/3372885.3373813

Received 2022-07-07; accepted 2022-11-07