School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

From Kripke Structures to Interaction Trees in Isabelle/HOL

by

Seung Hoon Park

Thesis submitted as a requirement for the degree of

Bachelor of Computer Science (Hons)

Submitted:	August 2021
Supervisor:	Dr. Christine Rizkallah
Student ID:	z5146542

Abstract

Interaction Trees (ITrees) are expressive coinductive data structures that are capable of representing effectful, recursive, and potentially non-terminating programs. ITrees conform to many desired mathematical properties, enabling interaction with the environment. Furthermore, ITrees possess a rich equational theory of equivalence up to weak bisimulation.

In this thesis, we aim to provide a method of transforming Kripke structures to ITrees. Kripke structures have been extensively used for representing program behaviours in the context of model checking and have various means of defining program properties such as safety and liveness properties.

Acknowledgements

I would like to thank Christine and Johannes for the supervision and guidance thoughout the journey. I would also like to thank Ambroise for the assistance.

Finally, I would like to thank my family, who have been supportive all the time.

Contents

1 Introduction **1**

2 Background **3**

2.1 Coinduction 3

2.1.1 Induction 3

2.1.2 Coinduction 5

2.2 Interaction Trees 8

2.2.1 Structure 8

2.2.2 Properties 10

2.2.3 Bisimulation 11

2.3 Isabelle/HOL 13

2.3.1 Restrictive Types 13

2.3.2 Coinductive Library 14

2.4 Kripke Structures 17

2.4.1 Defintion 17

2.4.2 Representation 18

2.4.3 Isabelle/HOL Formalisation 20

2.5 Computational Tree Logic 20

2.5.1 Syntax 21

2.5.2 Semantics 22

2.5.3 Isabelle/HOL Formalisation 23

3 Formalising ITrees in Isabelle/HOL 25

3.1 Structural Representation 26

3.2 Monad Definition 26

3.2.1 Friends 27

3.2.2 Bind 27

3.2.3 Ret 30

3.2.4 Monad Laws 30

3.3 Weak Bisimulation 30

3.3.1 `euttF` 30

3.3.2 Equivalence Relation Proof 31

3.4 Comparing Works 32

4 From Kripke Structures to ITrees 34

4.1 Overview 35

4.2 Issues 36

4.2.1 Multiple Initial States 36

4.2.2 Event Output Nondeterminism 39

4.3 Relating Both Structures 40

5 Conclusion 43

5.1 Future Work 44

Bibliography 45

Chapter 1

Introduction

Formal verification is a process associated with ensuring programs adhere to some formal specification. Formally verifying a program involves creating a formal proof, where the proof is checked autonomously by a computer. By giving a formal proof that the program implemented in a certain programming language behaves as specified, users of the program can ultimately trust that the program will always work as expected.

Interaction Trees (ITrees) are *coinductive* data structures that are capable of representing effectful, recursive and potentially non-terminating programs (Xia et al., 2019). ITrees have a simple structure and have a number of neat mathematical properties useful for evaluation and verification. Such properties further allow showing equivalence of two programs through *bisimulation*.

As expressive and intriguing as ITrees may be, extensive study has yet been performed on ITrees given how recent it has been formalised in theorem provers Isabelle/HOL. On the other hand, Kripke structures, which are state-transition graphs, have been studied and used extensively to represent programs (Kripke, 1963). Furthermore, it is possible to interleave such structures as an infinitely branching computation tree and reason about the behaviour via Computational Tree Logic (CTL) (Clarke, 2008). Given that Kripke structures as a state-transition graph may be interleaved as a computation tree, it may be possible to provide a straightforward transformation into an ITree, using the

idea that ITrees can also express an infinitely branching computation tree thanks to its coinductive structure.

In this thesis, we investigate how to represent Kripke structures as ITrees. We aim to have the ITree representation to preserve the behaviour of the program expressed by a CTL formula in the Isabelle/HOL theorem prover. We describe some issues that prevent a straight-forward transformation and explore how such issues could be addressed. We finally discuss how the transformed ITree may relate to the original Kripke structure.

Chapter 2

Background

2.1 Coinduction

ITrees are capable of modelling potentially non-terminating programs thanks to their coinductive structure (Xia et al., 2019). To understand how coinductive structures help model potentially non-terminating programs, it is imperative to understand what coinduction is and how it works. While induction is familiar to most computer scientists, coinduction is less well understood and less widely used (Kozen and Silva, 2017). Fortunately, both induction and coinduction are heavily related to each other (Kozen and Silva, 2017), thus if one has a good understanding of how induction works, one can also become quickly familiar with coinduction. In this section, we briefly discuss about induction and its structure and definition. We then discuss about coinduction and its structure and definition by directly comparing coinduction to induction.

2.1.1 Induction

Intuitively, an inductive set A is defined by starting with initial base elements in A and any other elements in A that are not the initial base elements can be constructed by applying some constructive operator *finitely* many times (AbdelGawad, 2019).

The set of natural numbers $\mathbb{N}$ can be considered as an inductive set by setting the base element as 0 and having the constructor be the successor operator $Succ : \mathbb{N} \to \mathbb{N}$, where $Succ\ n = n + 1, n \in \mathbb{N}$. Using this, any natural number can be defined by applying the $Succ$ operator to 0 finitely many times.

To prove that an element is contained in an inductively defined set A, we use *rule induction* (Sangiorgi, 2011). At a basic level, we consider two inference rules: one that considers the base case and one that considers the inductive step. Then the set A must be *closed forward under the rules*, i.e. if the premise of a rule is satisfied, then so must the conclusion of the rule.

$$\frac{}{0 \in \mathbb{N}} \qquad \frac{n \in \mathbb{N}}{Succ(n) \in \mathbb{N}}$$

Figure 2.1: Inference rules for $\mathbb{N}$

Figure 2.1 shows the two inference rules for $\mathbb{N}$. For the left rule, which is the base case, no assumptions about 0 is necessary: 0 is in the set $\mathbb{N}$ by definition. The right rule, which is the inductive case, states that if n is a natural number, then it follows that $Succ(n)$ is also a natural number.

Note that the set that is consistent with the given inference rules is not necessarily unique. If one replaces $\mathbb{N}$ with the set of real numbers $\mathbb{R}$ in Figure 2.1, then one may observe that the set $\mathbb{R}$ is still consistent with the rules. It is most desirable to consider the smallest set that is consistent with the rules, as this is what is required to obtain an induction principle. Such a smallest set consistent with the rules is called the *least fixed point* of the $Succ$ function.

To show that two elements are equivalent to each other, we show that there is an *identity* relation between the two elements (Sangiorgi, 2011). An identity relation is the smallest equivalence relation that is closed forward under given equivalence rules. If we have elements $a \in A$ and $b \in B$, we show that if a and b are related to each other, then $a\ \mathcal{R}\ b$ holds, where $\mathcal{R} \subseteq A \times B$ is a subset that contains all possible pairs (a, b) in which a and b are related to each other, and $a\ \mathcal{R}\ b$ means $(a, b) \in \mathcal{R}$. For natural

numbers, $(=) \subseteq \mathbb{N} \times \mathbb{N}$ can be used as an identity relation.

In the case of natural numbers, the following rules govern the identity relation for natural numbers:

$$\frac{}{0 \;=\; 0} \qquad \frac{n \;=\; m}{Succ(n) \;=\; Succ(m)}$$

Using these two rules, it is possible to prove whether two natural numbers are equivalent, or rather, identical to each other.

2.1.2 Coinduction

A coinductive set A can be built upon from the initial base elements by applying the constructive operations a finite or infinite number of times (AbdelGawad, 2019).

The set of *extended natural numbers* can be defined using a coinductive set. We consider the set of extended natural numbers $\overline{\mathbb{N}}$, where $\overline{\mathbb{N}} = \mathbb{N} \cup \{\infty\}$. Like with the inductive definition of natural numbers, the base element is 0. The successor operator is now extended by setting $Succ\ \infty = \infty$.

To prove that an element is contained in a coinductively defined set A, we use *rule coinduction* (Sangiorgi, 2011). Given inference rules that define the coinductive set A, A must be *closed backward under the rules*, essentially stating that for a given rule, if the conclusion of a given rule is true, then so must the premise of the rule (Sangiorgi, 2011).

$$\frac{}{0 \in \overline{\mathbb{N}}} \qquad \frac{n \in \overline{\mathbb{N}}}{Succ(n) \in \overline{\mathbb{N}}}$$

Figure 2.2: Inference rules for $\overline{\mathbb{N}}$

Figure 2.2 shows the inference rules for $\overline{\mathbb{N}}$. Note that the inference rules for this coinductively defined set looks nearly identical to those of the set of natural numbers defined

Figure 2.1. To make sense of these coinductive rules, we interpret these rules backwards. The left rule is the base case that states 0 is in the set of extended natural numbers. The right rule states that if $Succ(n)$ is an extended natural number, then it follows that n is also an extended natural number.

While an inductively defined set deals with the smallest set that is consistent with the rules, a coinductively defined set deals with the largest set that is consistent with the rules, and such a largest set is called the *greatest fixed point.* The set $\overline{\mathbb{N}}$ is the largest set that is consistent with the inference rules defined in Figure 2.2.

To show equivalence between two elements, where the two elements belong to coinductively defined sets or structures that are not necessarily the same, we show that there is a *bisimilarity* between the two elements (Sangiorgi, 2011). A bisimilarity is the largest relation that is closed backwards under given equivalence rules.

For the set of extended natural numbers, we also use $(=) \subseteq \overline{\mathbb{N}} \times \overline{\mathbb{N}}$. The equivalence rules for the set of extended natural numbers are essentially the same as those of the set of natural numbers. Using those rules, it is possible to show that two numbers are equivalent, or bisimilar, to each other.

inductive definition	coinductive definition
induction proof principle	coinduction proof principle
'forward closure' in rules	'backward closure' in rules
identity	bisimilarity
least fixed point	greatest fixed point

Figure 2.3: The correspondence table for induction and coinduction (Sangiorgi, 2011)

$$\frac{}{\texttt{[] :: L}(\alpha)} \qquad \frac{\texttt{x :: }\alpha \quad \texttt{xs :: L}(\alpha)}{\texttt{x \# xs :: L}(\alpha)}$$

Figure 2.4: Inference rules for lists of type α

$$\frac{\texttt{0 :: } Z \quad \texttt{zs :: L(}Z\texttt{)}}{\texttt{0 \# zs :: L(}Z\texttt{)}} \qquad \frac{\texttt{0 :: } Z \quad \dfrac{\vdots}{\texttt{0 \# [0, 0, 0, ...] :: L(}Z\texttt{)}}}{\texttt{0 \# [0, 0, 0, ...] :: L(}Z\texttt{)}}$$

Figure 2.5: Inference rule for infinite list of 0's and a proof derivation tree

We now consider a more involved example. Figure 2.4 shows the inference rules for lists. Note that α is a type, `L(`α`)` is a type of list comprised of elements of type α, and `#` : $\alpha \rightarrow$ `L(`α`)` $\rightarrow$ `L(`α`)` is a binary infix operator that adds a specified element to the beginning of the specified list. Whether `L(`α`)` is a type of finite or infinite list depends on whether we define the lists inductively or coinductively. If lists are defined inductively, then the length is finite. If one were to derive a proof derivation tree using the inference rules, then the depth of the tree is finite. Dually, if lists are defined coinductively, then the list may be potentially infinite in length. In such a case, the depth of the proof derivation tree may be infinitely long, but we note that there are no contradictions in any part of the derivation itself. As inference rules are reasoned backwards for a coinductively defined set, we note that if `x # xs` is a potentially infinite list of type α, then it follows that `x` is of type α and `xs` is a potentially infinite list.

Using coinductively defined lists, it is possible to model a stream of infinite zeroes, and only one inference rule needs to be considered, which is the one in Figure 2.5. We instantiate α to be Z, where $Z = \{0\}$. As we are coinductively defining this stream, `L(`Z`)` is the set of lists of zeroes. Then given a particular list `zl = [0, 0, 0, ...]`, where `zl` is an infinite list of zeroes, we can interpret this list as `zl = 0 # [0, 0, 0, ...]`. Then, using the inference rules, it is possible to show that `0` $\in Z$ holds and `[0, 0, 0, ...]` is an infinite list of zeroes. The proof derivation tree in Figure 2.5 demonstrates how a proof derivation tree can be constructed using the inference rule. We note that the length of the proof derivation tree is not finite, as we are continuously applying the same rule in the premise. In this instance, we may observe that `zl` is the greatest fixed point with respect to the `#` operator, as it can be observed that `0 # zl = zl` holds.

```
CoInductive itree (E: Type → Type) (R: Type): Type:=
| Ret (r: R)                               (* terminates at r *)
| Tau (t: itree E R)                       (* silent transition *)
| Vis {A: Type} (e: E A) (k: A → itree E R). (* visible event *)
```

Figure 2.6: ITrees in Coq (Koh et al., 2019; Xia et al., 2019)

2.2 Interaction Trees

In the previous section, we gave a gentle introduction to coinduction. This knowledge is essential in order to understand how Interaction Trees work, as ITrees depend on the coinductive structure to ensure they can model potentially non-terminating programs and also to show the equivalence of two programs via bisimulation. In this section, we give an introduction to the ITree data structure: we give some examples of programs that the ITree can model. We then discuss some nice mathematical properties that ITrees possess.

2.2.1 Structure

The ITree data structure is formally described in Figure 2.6. Given that the structure is described in Coq syntax, we first discuss the syntactic structure. We are given a coinductive data structure with the name `itree`, which takes in two arguments. The first is `E`, which is a higher-order type that models external, visible interactions with the environment (Xia et al., 2019). Examples of such functions are memory read and store operations and IO operations. The second argument is `R`, which defines the return type, should the function ever return. Should the ITree ever return a value, `R` can be instantiated to types such as `nat`, `int`, etc.

There are three constructors that define the coinductive data structure as seen in Figure 2.6.

- `Ret r` corresponds to a computation that terminates while yielding the value `r`.
- `Tau t` models a silent step of computation that does not does not produce any

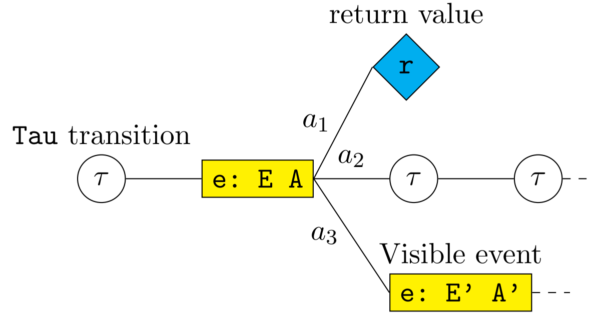


Figure 2.7: Visual representation of ITrees

visible artefacts to the environment and continues to the next computation as an ITree `t`.

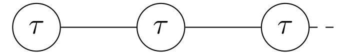
- `Vis e k` models a visible event that interacts with the environment. The visible event yields an answer of type `A` and passes this value to a continuation function `k`, which produces the rest of the ITree computation.

`Vis e k` is the only constructor that branches, where the branching is dependent on what the visible event `e` yields, as this could be observed in Figure 2.7.

Figure 2.8 give some examples of ITrees and their visual representations. The first example `spin` models a non-terminating program that does not produce any visible events. In Coq, such non-terminating programs can be modelled using the `CoFixpoint` construct, which could be understood as a corecursive definition. The corecursive definition `spin` also is the greatest fixed point of the corecursive function. The second example `echo` is a more interesting example. It is based off the UNIX command `echo`, which, given an input from `stdin`, immediately outputs the given value to `stdout`. Here, it is necessary to define an event type, which are the IO operations. Once again, it is a non-terminating program, and `echo` is the greatest fixed point for the program it represents.

```
CoFixpoint spin : itree _void :=
Tau spin.
```

(a) program `spin` denoted as an ITree

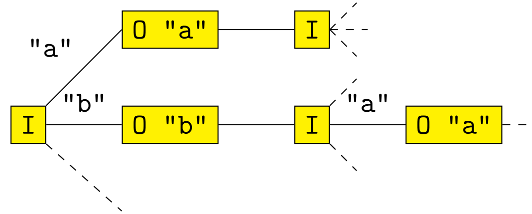


(b) ITree visual representation of `spin`

```
Inductive IO : Type → Type:=
| I : IO String
| O : String → IO ().

CoFixpoint echo:itree IO void :=
Vis I (λ x. Vis (O x) (λ _. echo))
```

(c) Program `echo` denoted as an ITree



(d) ITree visual representation of `echo`

Figure 2.8: Example programs denoted in ITrees (Xia et al., 2019)

2.2.2 Properties

ITrees adhere to many nice mathematical properties, as stated in Chapter 1. Perhaps the most interesting property ITrees possess is that the higher-order type `itree E` is a *monad* for any `(E: Type → Type)` (Xia et al., 2019). Monads provide a generic interface for program fragments (Hughes, 2000), and in the case of ITrees, it makes it convenient for the user to structure effectful computations, all through pure functional programming constructs (Xia et al., 2019). This also makes it easy to construct ITrees as if one were writing a program for an imperative language. Monadic operators such as `bind` and `ret` are defined for ITrees. After defining syntactic sugar to the `bind` operator like how it is done in Haskell, the `bind` operator may allow nicer representation of the ITree in a more human-readable form (Xia et al., 2019).

```
CoFixpoint echo:itree IO void :=
  Vis I (λ x.
    Vis (O x) (λ _.
      echo))
```

(a) Program `echo` denoted without using `bind`

```
CoFixpoint echo:itree IO void :=
  x ← trigger I;;
  trigger (O x);;
  echo;;
```

(b) Program `echo` denoted using `bind`

Figure 2.9: Equivalent programs of `echo` represented differently using `bind`

Figure 2.9 shows how the `echo` program may be represented using the `bind` operator. The `trigger` function simply takes in an event as argument, which returns a `Vis` node that takes in the event and returns the response produced by the event (Xia et al.,

2019). Formally, `trigger` is defined as follows:

```
Definition {E : Type → Type} {A: Type} (e : E A) : itree E A :=
  Vis e (λ x. Ret x)
```

Ultimately, note that we have a more visually pleasing ITree as a result of using `bind`, and this certainly holds for more complex ITrees.

2.2.3 Bisimulation

In Section 2.1.2, we briefly discussed how one can show equivalence between two elements that belong to coinductively defined set or structure that are not necessarily the same by showing there is a bisimilarity relation. As ITrees are coinductive structures, it is possible to show equivalence between two ITrees using the notion of *bisimulation*. There are two kinds of bisimulation that can be used for showing equivalence between two ITrees: *strong bisimulation* and *weak bisimulation*.

With strong bisimulation, the idea is that we check whether two ITrees essentially have the same shape. We denote `t1` ≅ `t2` if two ITrees are related by strong bisimulation. To show that two ITrees are related by strong bisimulation, it is essential to show that each nodes in the ITree are also related. In the same figure, we make the assumption that `Event` and `Event'` are related, and `r` and `r'` are related.

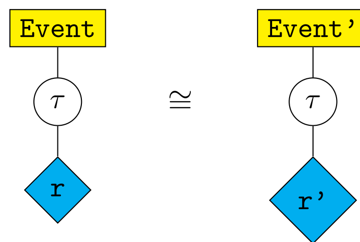


Figure 2.10: Visual depiction of two ITrees that are bisimilar

Figure 2.10 visually shows an instance where two ITrees are related under strong bisimulation. Loosely speaking, the two ITrees in the figure have the same shape, and we also note that for each node in one ITree, there is a node in the other ITree that we can relate.

Showing equivalence though strong bisimulation imposes a problem where strong bisimilarity may be too strict. Recall that `Tau` nodes are silent steps of computations that do not produce any visible artefacts to the environment; thus, if two programs ultimately yield the same visible behaviour and also returns equivalent values should the programs terminate, then both programs should be considered equivalent to each other, but it is not the case that both programs are strongly bisimilar to each other if both ITrees that denote their respective program differ in the number of finite `Tau` nodes. As an example, it is not possible to show `Tau t` $\cong$ `t` holds despite the fact that both ITrees are roughly equivalent. A coarser equivalence is needed, and the alternative method is to show that two ITrees are *equivalent up to* τ, a form of weak bisimulation defined (Xia et al., 2019).

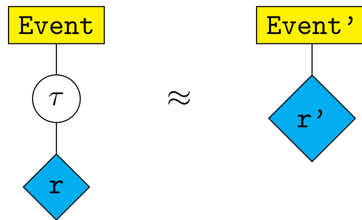


Figure 2.11: Visual representation of two ITrees related by *equivalence up to* τ

Figure 2.11 shows how two ITrees are related by weak bisimulation. Note that in this case, the two ITrees cannot be related by strong bisimulation due to the fact that the two structures, loosely speaking, have different shapes. Using equivalence up to τ, however, which is a coarser equivalence, it is possible to show that the two ITrees, which only differ in the number of `Tau` nodes by a finite amount, are equivalent.

When it comes to equivalence up to τ, on a high level, finitely chained `Tau` constructors are essentially ignored, or stripped off, when attempting to show equivalence between two ITrees. If two ITrees `t1` and `t2` are equivalent up to τ, we write `t1` $\approx$ `t2`. This makes it possible to show equivalence between programs that may silently diverge, like `spin`. It is also now possible to show that `Tau t` $\approx$ `t` holds using equivalence up to τ. This coarser definition will allow showing equivalence between two programs in the more user-intended way.

2.3 Isabelle/HOL

Isabelle/HOL is a well known interactive theorem prover capable expressing Higher-Order Logic (HOL) (Nipkow et al., 2002). It has extensive support for coinductive libraries, which makes it convenient for users to model coinductive structures in Isabelle/HOL (Biendarra et al., 2013). The Coq theorem prover has some features that are not present in Isabelle/HOL (Wiedijk, 2003, 2006) and used for the formalisation of ITrees. To port the data structure to Isabelle/HOL, it is important to understand the limitations of Isabelle/HOL. In this section we discuss how types can be defined in Isabelle/HOL and how the type system is a stumbling block for formalising ITrees. We also discuss some coinductive libraries that Isabelle/HOL supports and how they work in brief; they will be used as a solution to deal with the coinductive structure of ITrees.

2.3.1 Restrictive Types

The logic of Isabelle/HOL uses Higher-Order Logic (HOL) with rank-1 polymorphism (Kunčar and Popescu, 2017). Rank-1 polymorphism allows Isabelle/HOL to define datatypes parameterised by types. Lists in Isabelle/HOL is an example that uses rank-1 polymorphism to represent lists of type α. With this, users do not need to define a datatype for lists for every different types. This can also be achieved in the Coq theorem prover. The support of polymorphic inductive types allows a polymorphic list datatype to be defined. Figure 2.12 gives the implementation of lists in both the Isabelle/HOL and Coq theorem prover. In the case of Coq, we specify an arbitrary type `X`, which determines what type the list will hold. Both theorems provers utilise the rank-1 polymorphism feature to ensure that generic lists could be encoded.

In Coq, one may go even further, and this is apparent when one sees the ITree data structure defined in Figure 2.6. We notice `(E: Type → Type)` allows visible actions to be of a different type for each continuation of the ITree. `E`, in this case, is a higher-order type. Isabelle/HOL's restricted rank-1 polymorphism makes it very difficult to encode

```
datatype 'a list =
  Nil
| Cons 'a "'a list"
```

(a) List in Isabelle/HOL

```
Inductive list (X: Type) : Type:=
| nil : list X
| cons : X → list X → list X.
```

(b) List in Coq

Figure 2.12: Lists implemented in both Isabelle/HOL and Coq

codatatype $(\alpha_1, ..., \alpha_n)$ t $= C_1$ "$\tau_{1,1}$" ... "τ_{1,n_1}"
| ...
| C_k "$\tau_{k,1}$" ... "τ_{k,n_k}"

Figure 2.13: **codatatype** package (Blanchette et al., 2014; Nipkow and Klein, 2014)

higher-order types such as `E`. Restricting the type of visible event is an option; however, this would restrict the expressiveness of ITrees in Isabelle/HOL.

2.3.2 Coinductive Library

Coinductive data structures in Isabelle/HOL can be modelled using the **codatatype** package (Blanchette et al., 2014), and the syntax is defined in Figure 2.13. t is the name of the datatype, and $(\alpha_1, ..., \alpha_n)$ are the polymorphic type arguments presented in a tuple-like form. When defining a user type, the user must specify all of the types that will instantiate the polymorphic variables $(\alpha_1, ..., \alpha_n)$. The constructors are defined are defined as C_1, ..., C_k, and each constructor may be given a different number of postfix arguments.

Once a coinductive data structure has been defined using the **codatatype** package, Isabelle/HOL will automatically generate many nice theorems, including a coinduction principle. This makes it convenient to prove coinductive properties that the data structure possesses.

Various coinductive data structures were successfully formalised using the **codatatype** package in Isabelle/HOL (Lochbihler, 2010). Figure 2.15 gives an implementation of streams, which are practically lists that are infinite in length, using the **codatatype** package.

```
codatatype (α, o, ι) resumption =
  Pure (result: α)
| IO (output: o) (continuation: ι ⇒ (α, o, ι) resumption)
```

Figure 2.14: Resumption **codatatype** in Isabelle/HOL (Lochbihler and Züst, 2014)

```
codatatype 'a stream =
  SCons 'a "'a stream"

primcorec zeroes :: "nat stream" where
  "zeroes = SCons 0 zeroes"

lemma "zeroes = SCons 0 (SCons 0 zeroes)"
  using zeroes.code
  by (coinduction rule: stream.coinduct) auto
```

Figure 2.15: Application of **codatatype** and **primcorec** in Isabelle/HOL

There has been related work with modelling potentially non-terminating, effectful programs in Isabelle/HOL using **codatatype** done in the past. Based on Harrison's reactive resumption monad, the codatatype in Figure 2.14 defines a data structure in Isabelle/HOL that models interactive, potentially non-terminating programs. Specifically, the codatatype modelled the TLS networking protocol with the IO event[1]. We note that the structure is very similar to that of the ITree. The terminating `Ret` constructor corresponds to the `Pure` constructor in the resumption monad. Furthermore, the IO constructor, which models IO events, embedded in the resumption codatatype shares a very similar syntax to that of the `Vis` node in ITrees, especially with how continuations are passed. Despite the similarities, there are prominent differences between the two structures. In the resumption codatatype, the constructor responsible for silent transitions does not exist, whereas the `Tau` nodes in the Interaction Trees represent silent transitions. The bigger difference, however, would be how the event types are modelled. ITrees are capable of taking in generic event types, but in the resumption codatatype, it only specifically takes in an IO event with a specific behaviour.

Corecursive functions over codatatypes can be modelled using the **primcorec** package (Biendarra et al., 2013). In essence, much like how **primrec** defines primitive recursive functions over a defined datatype, **primcorec** defines primitive corecursive functions

[1]The IO event here differs the IO event that was based on ITrees in figure 2.8.

over a defined codatatype.

Figure 2.15 gives an example of a primitive corecursive definition defined using **primcorec**, which is based on the infinite list of zeroes in Figure 2.5. Using the theorems generated by **primcorec**, it is possible to further prove other lemmas. From this example, one may observe that `zeroes` is the greatest fixed point operator.

When it comes to ITrees, **primcorec** can model potentially non-terminating programs. Programs such as `spin` and `echo` could be modelled using **primcorec**, provided that a valid **codatatype** for ITrees is provided.

2.4 Kripke Structures

Developed by Saul Kripke, Kripke structures are state-transition graphs that, like ITrees, provide denotations to programs and represent the behaviour of such programs (Kripke, 1963). In practice, Kripke structures have been used extensively in the area of model checking and for quite a long time (Clarke, 2008). In this section, we formally define a Kripke structure and how programs may be represented in terms of a Kripke structure. We also provide a way to *unfold* such structures so that they may be interpreted as computation trees rather than a state-transition graph.

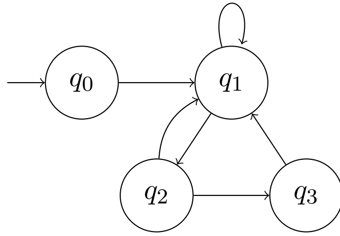
2.4.1 Defintion

First, we let AP be the set of atomic propositions. Then we formally define a Kripke structure as a 4-tuple $\mathcal{M} = (Q, I, \xrightarrow{\mathcal{R}}, L)$ (Clarke et al., 1999), where:

- Q is a finite set of states
- $I \subseteq Q$ is a set of initial states
- $\xrightarrow{\mathcal{R}} \subseteq Q \times Q$ is a (total) transition relation
- $L : Q \to \mathbb{P}(AP)$ is a labelling function

First, we note that $\xrightarrow{\mathcal{R}}$ is a total relation, i.e. for all $q \in Q$, there exists a $q' \in Q$ such that $q \xrightarrow{\mathcal{R}} q'$. Intuitively, all states in the Kripke structure must have an outgoing transition. Next, we note that for the labelling function L, $\mathbb{P}(AP)$ is the powerset of the set AP, the set of atomic propositions. Essentially, given a particular state in the Kripke structure, L returns which atomic propositions hold in that particular state.

Figure 2.16 depicts a visual example of the Kripke structure $\mathcal{M}$. For this specific structure we have four states $Q = \{q_0, q_1, q_2, q_3\}$. Of these four states, q_0 is the initial state for this structure, and this is depicted visually by an arrow that is not connected by two states. The transition relation for this structure is given as arrows between two



- $L(q_0) = \emptyset$
- $L(q_1) = \{x \geq 3\}$
- $L(q_2) = \{x = 3, y \neq 4\}$
- $L(q_3) = \{x = y\}$

Figure 2.16: Example of a Kripke structure $\mathcal{M}$

states. As an example, we observe that there is an arrow from state q_0 to q_1, i.e. there is a transition from state q_0 to state q_1. Formally, we state that $(q_0, q_1) \in \xrightarrow{\mathcal{R}}$. We may also treat $\xrightarrow{\mathcal{R}}$ as an infix operator where $q \xrightarrow{\mathcal{R}} q'$ is equivalent to $(q, q') \in \xrightarrow{\mathcal{R}}$. Thus, we have $q_0 \xrightarrow{\mathcal{R}} q_1$. For every other transitions between states q and q' in the structure, we have it that $q \xrightarrow{\mathcal{R}} q'$. Finally, the labels for each state has separately been given. In this instance, we have it that $AP = \{x \geq 3, x = 3, y \neq 4, x = y\}$, and we note that for any state $q \in Q$, $L(q)$ is a subset of AP. The labelling function states what specific properties hold at specific states in the structure. For instance, in state q_0, we observe that $L(q_0) = \emptyset$. This implies that none of the atomic propositions in AP holds in state q_0. On the other hand, for state q_2, we observe that $L(q_2) = \{x = 3, y \neq 4\}$. Specifically, this tells us that at state q_2, $x = 3$ and $y \neq 4$ are both true, whereas the other two atomic propositions in AP, which are $x \geq 3$ and $x = y$, are both false.

2.4.2 Representation

So far, we have given the impression that Kripke structures have a graph-like structures where *loops* are allowed by the definition of Kripke structures. However, it would be beneficial if we were able to interpret Kripke structures as a tree for two reasons:

- By interpreting Kripke structures as a tree instead of a graph, the transformation from Kripke structures to ITrees become much more straightforward and easy to understand.
- Properties of the program denotated by the Kripke structure may be formally expressed in terms of a modal logic formula. Specifically, by unfolding a Kripke

structure as a computation tree, we can express properties of the program in terms of a Computational Tree Logic (CTL) formula.

Given a Kripke structure, we are generally interested in how we interpret the structure when the program represented by this structure is executed. One way to obtain an execution model for the Kripke structure is to consider every possibility a program could take (Clarke et al., 1999). In this case, we consider every possible sequence of states, where any two contiguous states in the sequence are related by the transition relation. Because we interpret executions of programs in terms of a *linear* view, it is possible to express properties of all sequence of executions of programs in terms of Linear Temporal Logic (LTL) (Huth and Ryan, 2004). As stated above, however, we are more so interested in interpreting execution of programs in terms of a tree that may branch given each state at any point in the execution of the program. Instead of considering every possible sequences of states, it is possible to interleave the graph-like structure into a computation tree by unfolding the state graph from the initial states. Because the transition relation of Kripke structures are total, we obtain a branching computation tree with infinite depth, i.e. the resulting computation tree has no leaf nodes. Figure 2.17 depicts how the Kripke structure $\mathcal{M}$ in figure 2.16 can be *unfolded* into a computation tree.

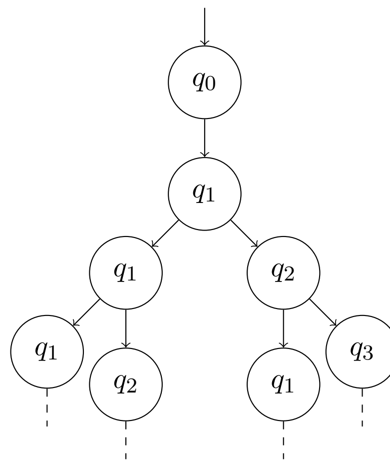


Figure 2.17: Kripke structure $\mathcal{M}$ in Figure 2.16 unfolded as a computation tree

2.4.3 Isabelle/HOL Formalisation

Kripke structures have been formalised under the more generalised Labelled Transition System (LTS) in Isabelle/HOL. It was created by Maximiliam Wuttke[2].

```
record ('states) TS =
  trans :: "'states rel"
  init :: "'states set"
record ('states, 'labels) LTS = "'states TS" +
  label :: "'states ⇒ 'labels"
```

Figure 2.18: LTS formalisation in Isabelle

Figure 2.18 shows how LTS is encoded in Isabelle/HOL. Maximilian first formalised a generic transition system that is comprised of a transition relation and a set of initial states via records. The record is then extended into an LTS by adding a labelling function. Here, we note that `'labels` is essentially the set of atomic propositions AP.

Using LTS, it is easy to obtain a Kripke structure by assuming that the number of states are finite. This can be done in a locale context where it is assumed that the number of states given for the LTS are finite.

2.5 Computational Tree Logic

In the previous section, we have formally defined what Kripke structures are and how such structures can be unfolded into an infinitely branching computation tree. In this section, we introduce Computational Tree Logic (CTL), a type of modal logic capable of specifying program properties (Baier and Katoen, 2008). Originally devised by Clarke and Emerson (1982), CTL is a powerful form of logic capable of expressing nontrivial properties such as *safety* properties, which loosely states that something bad will never occur, and *liveness* properties, which loosely states that something good will eventually occur (Lamport, 1977). We introduce the grammar that defines the general syntax for

[2]For unknown reasons, the github repository that contains this work tends to no longer be publicly accessible.

CTL. We then introduce the semantics for each of the operators in CTL and how each of these operators relate to Kripke structures in general.

2.5.1 Syntax

Formally, we define the grammar of CTL as follows (Baier and Katoen, 2008):

$$\phi ::= \text{true} \mid a \mid \phi \wedge \phi \mid \neg\phi \mid \mathbf{E}\varphi \mid \mathbf{A}\varphi$$
$$\varphi ::= \mathbf{X}\ \phi \mid \phi\ \mathbf{UNTIL}\ \phi$$

Given this grammar, we note the following:

- ϕ is a state formula
- φ is a path formula
- $a \in AP$ is an atomic proposition
- **E** and **A** are quantifiers for path fourmulae
- **X** and **UNTIL** are temporal operators
- A valid CTL formula is always derived starting from a state formula

Note that the syntax described above is the *smallest* grammar defined for CTL, that is, it is possible to derive different kind of operators using what is given by the grammar. For instance, the disjunction operator $x \vee y$ can be defined as $\neg x \wedge \neg y$, where we note both $\neg$ and $\wedge$ are operators that are part of the syntax for CTL. Similarly, other propositional logic operators such as $\longrightarrow$ (implication) can be defined using such operators.

With CTL, it is possible to express formulae where certain properties hold in the future, and this is made possible with the help of *path quantifiers* and *temporal operators.* Path quantifiers simply states whether a path formula holds for either only one possible execution, i.e. $\mathbf{E}\ \varphi$, or for all possible executions from the current state, i.e. $\mathbf{A}\ \varphi$. With the temporal operators, $\mathbf{X}\ \phi$ states whether the CTL formula ϕ holds in the next

state, whereas ϕ_1 **UNTIL** ϕ_2 states whether ϕ_1 holds for every state in the future until we encounter a state where ϕ_2 holds.

It is possible to express more sophisticated temporal operators using **X** and **UNTIL**. For instance, it is possible to define the temporal operator **F** ϕ to be *true* **UNTIL** ϕ, which loosely states that ϕ will eventually hold in the future. Another interesting temporal operator is **G** ϕ, which states that ϕ holds all the time. The **G** operator is dual to the **F** operator, thus, we define **EG** ϕ and **AG** ϕ to be $\neg$**AF** $\neg\phi$ and $\neg$**EF** $\neg\phi$ respectively.

Using operators such as **F** and **G**, it is possible to express safety and liveness properties. For instance, if ϕ represents a bad state, it is possible to represent the safety property where ϕ never occurs by expressing this in terms of the CTL formula **A G** $\neg\phi$. If ϕ represents a good state that should always be reached, it is possible to represent the liveness property where ϕ always eventually occurs by expressing this in terms of the CTL formula **A F** ϕ.

2.5.2 Semantics

We loosely defined the syntax of CTL and what each of the operators informally mean. We now formally define the semantics for each of these operators, how they relate to states in a Kripke structure, and finally how they relate to the Kripke structures themselves.

Before stating the semantics, we first discuss some preliminaries. Assume we are given a Kripke structure $\mathcal{M} = (Q, I \xrightarrow{\mathcal{R}}, L)$.

- Given a state $q \in Q$, $Paths(q)$ is a function that returns the set of all sequences of states that initially start from state q.
- Given $\sigma \in Paths(q)$ for some state $q \in Q$, $\sigma|_i$ denotes the $(i+1)$-th state in the sequence σ. By definition, we have it that $\sigma|_0 = q$.

Assume we are given a CTL formula ϕ and some Kripke structure $\mathcal{M} = (Q, I, \xrightarrow{\mathcal{R}}, L)$. Also, let $q \in Q$ and σ be a valid path for $\mathcal{M}$. The satisfiability relation $\models$ that provides the semantics for CTL is recursively given as follows (Baier and Katoen, 2008):

- $q \models a \quad \Leftrightarrow \quad a \in L(q)$
- $q \models \neg\phi \quad \Leftrightarrow \quad q \not\models \phi$
- $q \models \phi \wedge \psi \quad \Leftrightarrow \quad (q \models \phi) \wedge (q \models \psi)$
- $q \models \mathbf{E}\ \varphi \quad \Leftrightarrow \quad \exists \sigma \in Paths(q).\ \sigma \models \varphi$
- $q \models \mathbf{A}\ \varphi \quad \Leftrightarrow \quad \forall \sigma \in Paths(q).\ \sigma \models \varphi$
- $\sigma \models \mathbf{X}\ \phi \quad \Leftrightarrow \quad \sigma|_1 \models \phi$
- $\sigma \models \phi\ \mathbf{UNTIL}\ \psi \quad \Leftrightarrow \quad \exists j \geq 0.\ \sigma|_j \models \psi \wedge (\forall 0 \leq k < j.\ \sigma|_k \models \phi)$

We ultimately also have it that $\mathcal{M} \models \phi$ if and only if for all initial states $q_i \in I$, $q_i \models \phi$, i.e. we have: $\mathcal{M} \models \phi \Leftrightarrow \forall q_i \in I.\ q_i \models \phi$. Overall, this is how Kripke structures relate to a CTL formula.

2.5.3 Isabelle/HOL Formalisation

Like with LTS, Maximilian Wuttke formalised CTL alongside with LTS, although the work is also no longer publicly visible. The syntax of CTL is essentially formalised as a datatype in Isabelle/HOL, and the semantics is given in terms of a recursive function. Figure 2.19 notes the CTL syntax and semantics formalised in Isabelle/HOL. We note that the smallest grammar is not used in this case, and this can be checked by observing that the grammar allows additional propositional operators such as `False` and `Or`.

```
datatype 'a formula =
  True_form ("true𝔣")
| False_form ("false𝔣")
| Atom_form 'a ("atom𝔣'(_')")
| Neg_form "'a formula" ("not𝔣 _")
| And_form "'a formula" "'a formula" ("_ and𝔣 _")
| Or_form "'a formula" "'a formula" ("_ or𝔣 _")
| E_form "'a pformula" ("E𝔣 _")
| A_form "'a pformula" ("A𝔣 _")
and 'a pformula =
  X_form "'a formula" ("X𝔣 _")
| U_form "'a formula" "'a formula" ("_ U𝔣 _")
```

(a) CTL Syntax in Isabelle/HOL

```
fun ctl_sat :: "'states ⇒ 'ap formula ⇒ bool" and
  ctl_path_sat :: "'states word ⇒ 'ap pformula ⇒ bool" where
  ctl_sat_true: "s ⊨ctl true𝔣 = True"
| ctl_sat_false: "s ⊨ctl false𝔣 = False"
| ctl_sat_prop: "s ⊨ctl atom𝔣(a) = (a ∈ label 𝒯 s)"
| ctl_sat_and: "s ⊨ctl (ϕ1 and𝔣 ϕ2) = (s ⊨ctl ϕ1 ∧ s ⊨ctl ϕ2)"
| ctl_sat_or: "s ⊨ctl (ϕ1 or𝔣 ϕ2) = (s ⊨ctl ϕ1 ∨ s ⊨ctl ϕ2)"
| ctl_sat_not: "s ⊨ctl (not𝔣 ϕ) = (¬ s ⊨ctl ϕ)"
| ctl_sat_A: "s ⊨ctl (A𝔣 ϕ) = (∀ π. exec_frag 𝒯 s π ⟶ π ⊨ctlp φ)"
| ctl_sat_E: "s ⊨ctl (E𝔣 ϕ) = (∃ π. exec_frag 𝒯 s π ∧ π ⊨ctlp φ)"

| ctl_path_sat_X: "π ⊨ctlp (X𝔣 ϕ) = (π 1 ⊨ctl ϕ)"
| ctl_path_sat_U: "π ⊨ctlp (ϕ1 U𝔣 ϕ2) =
    (∃ j. (π j ⊨ctl ϕ2) ∧ (∀ k. 0 ≤ k ∧ k < j ⟶ π k ⊨ctl ϕ1))"
```

(b) CTL Semantics in Isabelle/HOL

Figure 2.19: CTL formalisation in Isabelle/HOL

Chapter 3

Formalising ITrees in Isabelle/HOL

Initially, the original aim of the thesis was to port ITrees to Isabelle/HOL. Around the beginning of this thesis, ITrees was formalised only in the Coq theorem prover by Xia et al. (2019). There were challenges associated to porting ITrees to Isabelle/HOL due to how the types worked differently between both theorem provers. Specifically, Isabelle/HOL's lack of *higher order types*, which were necessary for encoding event types, made the job of formalising the data structure much harder. Furthermore, Isabelle/HOL's restriction with corecursive functions made it somewhat more challenging to write certain programs that would otherwise be possible to do in the Coq formalisation.

Around the end of the first half of my thesis, Foster et al. (2021) were able to successfully formalise ITrees in Isabelle/HOL and published their work. Thereafter, we shifted the main focus of the thesis to a topic that was still related to ITrees.

In this section, we discuss our attempt at formalising the coinductive data structure in Isabelle/HOL and the progress that was made. We then introduce the formalisation Foster et al. (2021) completed and compare their work to ours.

3.1 Structural Representation

The original plan was to deal with the issue of higher order types later and first focus on defining an appropriate `codatatype` representation of ITrees in Isabelle/HOL. Figure 3.1 shows our attempt at formalising the data structure in Isabelle/HOL. One may note that there is a great similarity between the resumption codatatype defined in figure 2.14 and this ITree codatatype.

```
codatatype ('e, 'a, 'r) itree =
Ret (r: 'r)
| Tau (t: ('e, 'a, 'r) itree)
| Vis (e: 'e) (k: 'a ⇒ ('e, 'a, 'r) itree)
```

Figure 3.1: Attempted ITree formalisation in Isabelle/HOL

The idea behind the implementation of the codatatype in figure 3.1 was to parameterise the event response type as part of the ITree definition. While this restricts the event response type to `'a`, this allows us to define the continuation tree `k` that takes in a response of an event as input.

3.2 Monad Definition

To show that our implementation of the ITree data structure is a monad, there are primarily two tasks that need to be done:

- Define the `bind` and `ret` operators
- Prove the three laws of monad, which are associativity, left identity, and right identity

Defining the `ret` operator is trivial, as we can utilise the fact that we can return a `Ret` node as its definition. Defining the `bind` operator, however, turns out to be problematic in Isabelle/HOL. In particular, we encounter the issue of *friendly* corecursive functions.

3.2.1 Friends

In Isabelle/HOL, corecursive functions may be declared friendly, which states that such function preserves productivity of their arguments (Blanchette et al., 2017). If a corecursive function is friendly, it can be used on other corecursive call contexts; however, one needs to prove that a corecursive function is friendly, and this introduces restrictions as to how corecursive functions can be defined in practice.

In summary, here is a list of restrictions imposed for friendly functions (Blanchette et al., 2017):

- Friendly functions consume up to one constructor and returns a constructor
- All type variables used within the argument of the friendly function must also be present in the resulting codatatype.

3.2.2 Bind

Based on how `bind` was defined in the Coq formalisation, one may initially write the following corecursive definition for bind, given in figure 3.2

```
corec bind :: "('e, 'a, 'r) itree ⇒ ('r ⇒ ('e, 'a, 's) itree)
            ⇒ ('e, 'a, 's) itree"
  where
  "bind t1 f = (case t1 of
    Ret re ⇒ f re
    | Tau te ⇒ bind te f
    | Vis ex ke ⇒ Vis ex (λ x. (bind (ke x) f)))"
```

Figure 3.2: Initial attempt for defining monad operator `bind` for ITrees

Defining `bind` as a corecursive function like how it is given in figure 3.2 does not impose any problem in Isabelle/HOL, but one would not be able to use the `bind` operator in any ITrees defined corecursively unless the corecursive function is proven to be friendly; thus, it is essential to prove that the corecursive function is friendly. However, there are three problems with this definition when attempting to prove that this function is friendly:

- The type variable `'r` is missing in the return type for `bind`, which violates the second restriction.
- `t1` is being destructed, but in the case where `t1` is a `Ret` node, it does not return an ITree constructor, but rather whatever is produced by `f re`. This violates the first restriction.
- In the case where `t1` is a `Tau` node, we have the same problem as above where `bind te f` is not a constructor for an ITree. This again violates the first condition.

To solve the issue with the type variable, there are two solutions. One is to change the data structure such that it has an additional phantom type parameter (Blanchette et al., 2017). Figure 3.3 shows the modified data structure. Note that the type signature for bind can be changed such that the phantom type `'p` is of type `'r` everywhere. This ensures that the type variable `'r` is present in the return type. While this solves the type issue, it turns out that that *proving* the function to be friendly, assuming that the other issues are solved, still remains to be rather challenging. Blanchette et al. (2017) suggests an alternative approach where the type signature remains as is, but we provide a separate type signature when proving that the function is friendly. Contrary to the approach where we introduce phantom types, this approach ensures that proving friendliness is much easier. Ultimately, however, this means that the return type of ITrees must be restricted when using `bind`.

```
codatatype ('e, 'a, 'r, 'p) itree =
Ret (r: 'r)
| Tau (t: ('e, 'a, 'r, 'p) itree)
| Vis (e: 'e) (k: 'a ⇒ ('e, 'a, 'r, 'p) itree)
```

Figure 3.3: ITree with phantom types

To solve the other two issues, we may perform a case split on `f` and return constructors directly, or simply wrap them in a `Tau` node.

Figure 3.4 shows a corecursive definition of `bind` that is proven friendly. As mentioned above, however, this `bind` operator is limited in that return types between ITrees and continuations are restricted.

```
corec bind :: "('e, 'a, 'r) itree ⇒ ('r ⇒ ('e, 'a, 's) itree)
            ⇒ ('e, 'a, 's) itree"
  where
  "bind t1 f = (case t1 of
    Ret re ⇒ (case f re of
        Ret ree ⇒ Ret ree
        | Tau tee ⇒ Tau tee
        | Vis exx kee ⇒ Vis exx kee)
    | Tau te ⇒ Tau (bind te f)
    | Vis ex ke ⇒ Vis ex (λ x. (bind (ke x) f)))"

friend_of_corec bind :: "('e, 'a, 'r) itree
            ⇒ ('r ⇒ ('e, 'a, 'r) itree)
            ⇒ ('e, 'a, 'r) itree"
  where
  "bind t1 f = (case t1 of
    Ret re ⇒ (case f re of
        Ret ree ⇒ Ret ree
        | Tau tee ⇒ Tau tee
        | Vis exx kee ⇒ Vis exx kee)
    | Tau te ⇒ Tau (bind te f)
    | Vis ex ke ⇒ Vis ex (λ x. (bind (ke x) f)))"
  by (simp add: bind.code) transfer_prover
```

Figure 3.4: Monad operator `bind` defined for ITrees, with friendliness proof

3.2.3 Ret

As stated previously, `ret` can be defined rather easily. We use the same definition Xia et al. (2019) used.

```
defintion ret :: "'r ⇒ ('e, 'a, 'r)" itree
  where
  "ret x = Ret x"
```

Figure 3.5: Monad operator `ret` defined for ITrees

3.2.4 Monad Laws

Using the `bind` and `ret` operators, we were successfully able to prove that the three monadic laws hold. Specifically, we were able to prove associativity, left identity, and right identity. All three properties were proven using the coinduction principle for ITrees (i.e. strong bisimulation relation) that was automatically generated when defining the ITree codatatype.

3.3 Weak Bisimulation

As stated in Chapter 2, one of the nice features of defining a codatatype in Isabelle/HOL is that a coinduction principle is automatically generated, i.e. a strong bisimulation equivalence relation is produced for use. For us, this means that a strong bisimulation relation for ITrees has already been given to us, and the only work that remains is to define a weak bisimulation relation and prove that such a relation is an equivalence relation. Specifically, we formalise equivalence up to τ.

3.3.1 `euttF`

To formalise the equivalence up to τ relation, we first define the equivalence up to τ fixed point operator `euttF`. Our implementation of the operator is heavily based on the `euttF` definition provided by Xia et al. (2019).

```
inductive euttF :: "('r ⇒ 's ⇒ bool)
  ⇒ (('e, 'a, 'r) itree ⇒ ('e, 'a, 's) itree ⇒ bool)
  ⇒ ('e, 'a, 'r) itree
  ⇒ ('e, 'a, 's) itree
  ⇒ bool
  where
  EqRet "[| R a b |] ⟹ euttF R sim (Ret a) (Ret b)"
| EqVis "[| ∀v. sim (k1 v) (k2 v) |]
    ⟹ euttF R sim (Vis ev k1) (Vis ev k2)"
| EqTau "[| sim t1 t2 |] ⟹ euttF R sim (Tau t1) (Tau t2)"
| EqTauL "[| euttF R sim t1 ot2 |] ⟹ euttF R sim (Tau t1) ot2"
| EqTauR "[| euttF R sim ot1 t2 |] ⟹ euttF R sim ot1 (Tau t2)"
```

Figure 3.6: `euttF` operator for ITree

Figure 3.6 shows our initial implementation of the `euttF` operator in Isabelle/HOL. We note that `Ret` nodes are related by `R`, and both `Vis` and `Tau` nodes are related by `sim`. With `Tau` nodes, we consider extra cases where two nodes in an ITree are not the same type; two ITrees may still be related as long as one one of the ITrees have a finite chain of `Tau` nodes that could be 'peeled off'. Using this inductive definition, we were able to prove that `euttF R` for some relation `R` between return types is monotone.

3.3.2 Equivalence Relation Proof

When proving that a relation $\mathcal{R}$ is an equivalence relation, there are primarily three properties that must be proven:

- Reflexive property, where for any x, $x \mathcal{R} x$ is true
- Symmetric property, where assuming $x \mathcal{R} y$ is true, $y \mathcal{R} x$ is also true
- Transitive property, where assuming $x \mathcal{R} y$ and $y \mathcal{R} z$ are both true, $x \mathcal{R} z$ holds

We defined `eutt` to be the greatest fixed point over the `euttF` operator in Isabelle/HOL. Using this, we were able to show that `eutt` is reflexive and symmetric. Proving that `eutt` is transitive turned out to be much more challenging on the other hand. We attempted to formalise a reflexive-transitive closure for `eutt` to aid in proving the

transitive property, but this is around the time when Foster et al. (2021) finished formalising ITrees in Isabelle/HOL and published their results.

3.4 Comparing Works

```
codatatype ('e, 'r) itree =
Ret 'r
| Sil "('e, 'r) itree"
| Vis "'e ⇸ ('e, 'r) itree"
```

Figure 3.7: ITrees in Isabelle/HOL (Foster et al., 2021)

Figure 3.7 gives the encoding of ITrees in Isabelle/HOL done by Foster et al. (2021). There are some notable difference compared to the Coq ITree that we may observe. Trivially, the `Tau` node in this case is named `Sil`. More importantly, however, we observe the bigger deviation in how the `Vis` node is encoded and also how the continuation is defined. Instead of defining events inductively, they are defined in terms of `channels` in Isabelle/HOL, where channels are essentially a form of data. ITrees can be executed when given a finite set of channels, which essentially carries data of various types (Foster et al., 2021). Furthermore, the continuation is given as a partial function, hence ⇸ in the continuation function, rather than a total function like it is given in the Coq formalisation. The partial function used in their formalisation is based on a custom `Z_toolkit` library they have formalised. As as result, many of their formalisation and proofs rely heavily on their proprietary library that they have developed.

Foster et al. (2021) defined the monadic `bind` and `ret` operators incredibly similar to how we have defined them. They also had to deal with the problem where `bind` had to be proven friendly in order for it to be used in different corecursive call contexts. This implies that their implementation faces the same issue where the return types of ITrees have to be restricted. Foster et al. (2021) also formalised weak bisimulation with a proprietary library that they have developed; thus, they were able to show that their weak bisimulation relation is an equivalence relation. They define an inductive fixpoint operator for weak bisimulation like we and Xia et al. (2019) did, but there are some differences in the definition. The most important difference is the type of the

relation passed into the operator. In our implementation of the fixpoint operator given in 3.6, both the return relation and ITree relation passed into the fixpoint operator are *heterogeneous relations*. As opposed to *homogeneous relations* that generally has the form $\mathcal{R} \subseteq A \times A$ for some type A, heterogeneous relations are generalised such that we have $\mathcal{R} \subseteq A \times B$ for some types A and B. Note that in our definition of `euttF`, for both the return relation `R` and ITree relation `sim`, they take in different types. The idea behind this was to first define a generalised fixpoint operator like how it was done by Xia et al. (2019), and then pass in homogeneous relations for the fixpoint operator when proving equivalence. However, the generalisation of the structure resulted in harder proofs and defining other necessary lemmas relating to properties of `euttF` in Isabelle/HOL. Foster et al. (2021) formalises the fixpoint operator by simply fixing the type of the relation to be homogeneous. Furthermore, their definition does not require the return relation to be passed. This resulted in a much less-cluttered proof structure.

Finally, they proved other nontrivial properties of ITrees that we were planning on addressing afterwards.

At this point in the thesis, we shift our focus to a different plan that utilises the ITree formalisation given by Foster et al. (2021). In our modified plan, we talk about how Kripke structures could be represented as ITrees, which is discussed in the next chapter.

Chapter 4

From Kripke Structures to ITrees

In this section, we will talk about the modified plan. Note that in Chapter 2, we formally introduced ITrees and Kripke structures, but we have not stated how both structures necessarily relate to each other. Given the rich mathematical properties that ITrees have, it may be desirable if one is able to represent Kripke structures as ITrees. We provide a general idea of the transformation from Kripke structures to Itrees. We then discuss some nontrivial issues that may prevent a direct, faithful transformation and propose potential solutions to these issues. Finally we provide a (theoretical) relation operator that relates the Kripke structure and the ITree that was transformed from the Kripke structure.

4.1 Overview

In Chapter 2, we stated how Kripke structure as a state-transition graph could be interleaved, or unfolded, into an infinitely branching computation tree. This serves two purposes. One is to provide a way to relate Kripke structures to CTL formulae, since CTL reasons with computation trees. The other purpose is to show how the interleaved computation tree could be directly transformed into an Interaction Tree.

Before loosely defining the transformation process, we first discuss the event that will be used in the ITree, formalised in Coq style:

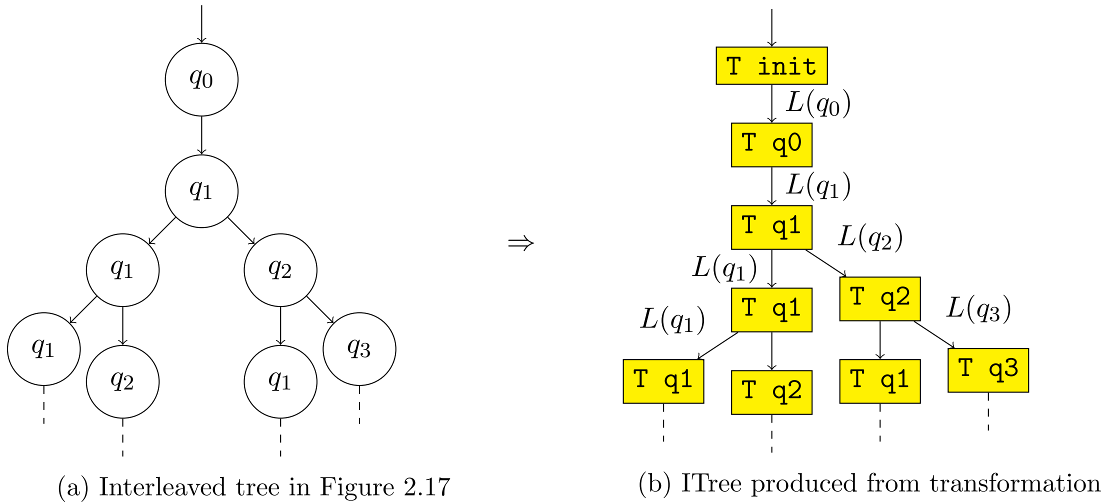
```
Inductive TransE : Type → Type :=
| T : Q → TransE ℙ(AP')
```

Intuitively, `T` is a visible event that takes in a state $q \in Q$ and outputs the labels of the *successors* of the state q. We also have it that $AP' \subseteq AP$ is the set of atomic propositions that were used in the labelling function for the given Kripke structure. For instance, if we had q' and q'' such that $q \xrightarrow{\mathcal{R}} q'$ and $q \xrightarrow{\mathcal{R}} q''$, then we have it that the response produced by the event `T q` are $L(q')$ and $L(q'')$.

Assume we are given an infinitely branching computation tree obtained from unfolding a Kripke structure $\mathcal{M} = (Q, I, \xrightarrow{\mathcal{R}}, L)$. We further assume that there is a certain CTL formula ϕ such that $\mathcal{M} \models \phi$. Here, ϕ is supposed to capture some behaviour of the Kripke structure, and the idea is that the same behaviour will be preserved after the Kripke structure is transformed into an ITree. For a subtree that starts with state $q \in Q$ that has transitions to states $q_1, ..., q_n \in Q$ we construct an ITree whose initial node is `Vis (T q)`. The input for the continuation tree k will be the labels of the successors of the state q, and assuming $L(q')$ is returned for some successor $q' \in Q$, `Vis (T q')` will be returned. This way, we observe that there is a one-to-one correspondence between the states in the computation tree and the nodes in the ITree.

However, this one-to-one correspondence between states and nodes is not enough to show that the Kripke structure and ITree are related. While the relation will be dis-

cussed in the last section of this chapter, we need to ensure that there is a transition that transitions into the initial state. Thus, the event `T` takes in an extra input we name `init`. The idea behind this is to ensure that there is a transition to the initial state. Figure 4.1 shows a visual example of how a computation tree interleaved from the Kripke structure in figure 2.16 is transformed into an ITree using the process we described above.



(a) Interleaved tree in Figure 2.17 (b) ITree produced from transformation

Figure 4.1: A visual representation of how the transformation between Kripke structures and ITrees work

4.2 Issues

Unfortunately, transforming a Kripke structure to an ITree is not a straightforward process, and there are a number of issues we need to consider.

4.2.1 Multiple Initial States

By definition, a Kripke structure is defined to have a set of initial states. This implies that it is possible for Kripke structures to have multiple initial states, which may be problematic when we attempt to transform this structure into an ITree. ITrees are trees that start off with a single node, so there is no notion of having multiple initial

nodes for an ITree.

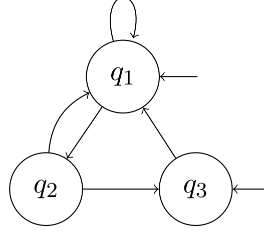


Figure 4.2: A Kripke structure $\mathcal{M}$ with multiple initial states

Figure 4.2 gives an example of a kripke structure that has multiple initial states. Here, both q_1 and q_3 are both initial states, for this particular Kripke structure. Because there are more than one initial states, it becomes unclear as to how the corresponding ITree would be constructed.

There are two solutions that solves this issue. For the first solution, Assume we are given a Kripke structure $\mathcal{M} = (Q, I, \xrightarrow{\mathcal{R}}, L)$, where $|I| \geq 2$. Also, assume we are given a CTL formula ϕ such that $\mathcal{M} \models \phi$ holds. The idea is to modify the Kripke structure such that we have a new state in the Kripke structure. This new state will be an initial state in the new Kripke structure and will transition to all the old initial states in I. Finally, all old initial states are no longer initial states for the new structure. Ultimately, we obtain a Kripke structure, say $\mathcal{M}' = (Q \cup \{q_i\}, \{q_i\}, \xrightarrow{\mathcal{R}}', L')$, that extends from $\mathcal{M}$ while having one unique initial state.

While this fixes the problem where we had multiple initial states, this now introduces a new problem where the modified model $\mathcal{M}'$ may no longer satisfy the CTL formula ϕ, i.e. we may have it that $\mathcal{M} \models \phi$ but $\mathcal{M}' \not\models \phi$, which is undesirable. To fix this issue, we also modify the CTL formula itself such that the behaviour of the modified Kripke structure can still be expressed in terms of a modified CTL formulae. The solution is as follows: we consider the CTL formula $\phi' = \mathbf{AX}\ \phi$. The claim is that $\mathcal{M} \models \phi \Leftrightarrow \mathcal{M}' \models \phi'$. For the proof idea, if $\mathcal{M} \models \phi$, then for all initial states $q_j \in I$, $q_j \models \phi$ holds. Note that when checking whether $\mathcal{M}' \models \phi'$ we are essentially checking whether $q_i \in \mathbf{AX}\ \phi$ holds. Informally, we check whether ϕ holds in all of the next states of q_i, but we note that all next states of q_i are exactly q_j, thus, by the assumption, we

have it that $\mathcal{M}' \models \phi'$ also holds. Using a similar argument, it is also possible to show that the converse of this statement holds.

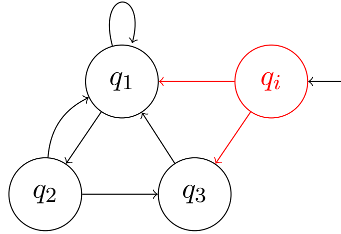


Figure 4.3: The modified Kripke structure $\mathcal{M}'$ from Figure 4.2

Figure 4.3 gives a visual representation of the modified Kripke structure. Note that the red states and transitions are the only parts that was modified from the original Kripke structure given in Figure 4.2.

The second solution involves no modification in the original Kripke structure. From the ITree, we let the event `T init` output labels of the initial states rather than simply fixing the event such that there is always one unique output.

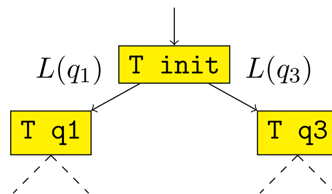


Figure 4.4: Output of event `T init` producing labels of all initial states

Figure 4.4 gives the first two levels of the ITree produced by the transformation process from the Kripke Structure given in Figure 4.2. Note that there are no modifications that need to be made to the original Kripke structure $\mathcal{M}$ along with the CTL formula ϕ where $\mathcal{M} \models \phi$. This ensures that there is no need to prove extra properties that would have otherwise been necessary. In the later section, we will show that the transformed ITree and the original Kripke structure still relate to each other.

4.2.2 Event Output Nondeterminism

Assume we are given a Kripke structure $\mathcal{M} = (Q, I, \xrightarrow{\mathcal{R}}, L)$. Also, assume we are given a state $q \in Q$ such that there exists two states $q', q'' \in Q$ such that $q' \neq q''$, $q \xrightarrow{\mathcal{R}} q'$, $q \xrightarrow{\mathcal{R}} q''$, and $L(q') = L(q'')$. Essentially, we have two distinct states in $\mathcal{M}$ that share some predecessor state and also share the same labels. While this is perfectly valid to have in a Kripke structure, problems arise when we attempt to create `Vis` nodes in the ITree.

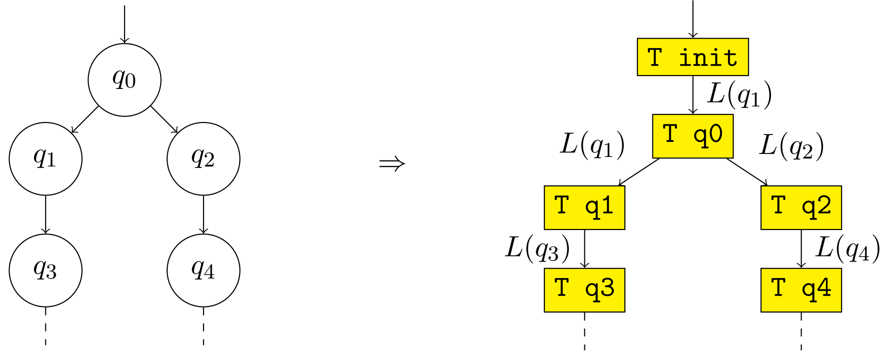


Figure 4.5: Case where issue of nondeterminism exists in ITree

Figure 4.5 gives an example where this issue is apparent. If we assume that $L(q_1) = L(q_2)$, then note that there is only one output the event `T q0` can produce. Thus, it is not possible to construct a continuation tree `L` that can make a deterministic transition to either `Vis (T q1)` or `Vis (T q2)`, which are distinct visible nodes in the ITree.

To ensure that we do not confront the issue of nondeterminism in the ITree, we make modifications to the Kripke structure. Specifically, we modify the labels of the states such that we would not have two states who share the same predecessor state and also have the same labels.

We introduce the idea of *fresh labels*. Given a Kripke structure $\mathcal{M}$ and a CTL formula ϕ such that $\mathcal{M} \models \phi$, a fresh label is a distinct atomic proposition $l \in AP$ such that $l \notin \{a \mid \forall q \in Q.\ a \in L(q)\}$ and $l \notin ap(\phi)$, where $ap(\phi)$ is the set of all atomic propositions used within the CTL formula ϕ. Essentially, l is an atomic proposition that is used in neither the labels of the Kripke structure nor the CTL formula.

The idea is that we extend the labelling function for states with the same labels by adding fresh labels. For instance, if we had $q \xrightarrow{\mathcal{R}} q'$, $q \xrightarrow{\mathcal{R}} q''$, and $L(q') = L(q'')$, then we modify the labelling function of the Kripke structure to L' such that L' is defined as follows:

$$L'(q) = \begin{cases} L(q) \cup \{l_1\} & \text{if } q = q' \\ L(q) \cup \{l_2\} & \text{if } q = q'' \\ L(q) & \text{else} \end{cases}$$

where $l_1, l_2 \in AP$ are distinct fresh labels, i.e. $l_1 \neq l_2$. We now observe that $L'(q') \neq L'(q'')$, which ensures that successors of q will no longer have the same labels.

Ultimately, the modified Kripke structure $\mathcal{M}' = (Q, I, \xrightarrow{\mathcal{R}}, L')$ will have no issues with nondeterminism when transforming the structure into an ITree.

Like with the issue with having multiple initial states, we need to ensure that the CTL formula satisfiability is preserved. In this case it should follow that modifying the structure preserves the satisfiability of the CTL formula, i.e. we have $\mathcal{M} \models \phi \Leftrightarrow \mathcal{M}' \models \phi$. For a high level idea behind proving this claim, we note that the only difference between the original and modified structures are the atomic propositions used within the structure. Because the labels added to the modified labelling function are fresh labels, the idea is that this does not affect CTL satisfiability.

4.3 Relating Both Structures

A computation tree interleaved from a Kripke structure and an Interaction Tree share many similar features. Both are, loosely speaking, transition systems that take the form of a tree, and both trees may have infinitely many transitions. However, they also are incredibly different. Computation trees interleaved from a Kripke structure are labelled transition systems, i.e. each states have labels that state what is true in each state. This is not necessarily the case with ITrees, there is no notion of a labelling function that takes in a node and outputs what is true at such nodes. For this reason, it is not easy to directly relate CTL with ITrees. As stated in Chapter 2, the CTL

semantics require that on the basic level, states have labels that define what is true at which states. This also imposes a challenge in relating the Kripke structure and the ITree directly.

It may be possible, however, to relate the two structures by considering the *traces* produced by the two structures. Traces for both Kripke structures and ITrees have a different definition. For Kripke structures, given a path $\sigma = q_i\ q_a\ q_b\ ... \in Paths(q_i)$, where $q_i \in I$ is an initial state in the Kripke structure $\mathcal{M} = (Q, I, \xrightarrow{\mathcal{R}}, L)$ and $q_a, q_b, ... \in Q$ with $q_i \xrightarrow{\mathcal{R}} q_a$, $q_a \xrightarrow{\mathcal{R}} q_b$, and so on, $Trace(\sigma)$ is the sequence of labels applied to each of the states in the path σ, i.e. $Trace(\sigma) = L(q_i)\ L(q_a)\ L(q_b)\ ...$ (Baier and Katoen, 2008).

For ITrees, a trace is inductively defined to be a *finite* sequence of events (Xia et al., 2019). Formally, traces are defined in Figure 4.6 in the Coq formalisation.

```
Inductive trace (E: Type → Type) (R: Type): Type:=
| TEnd : trace E R
| TRet : R → trace E R
| TEventEnd : ∀{X}, E X → trace E R
| TEventResponse : ∀{X}, E X → X → trace E R → trace E R.
```

Figure 4.6: Inductive definition of Traces for ITrees (Xia et al., 2019)

Note that in particular, `TEventResponse e x t` states whether a `Vis` node takes in an event `e` with response `x` and ultimately continues with trace `t` (Xia et al., 2019).

Using this, it is possible to define a function that checks whether Kripke structure trace relates to a ITree trace. The idea is that we compare traces of Kripke structures to the *responses* of the events in the trace of ITrees. There are two claims that we can make that ultimately relate both structures:

- For all traces defined for a Kripke structure, there exists a trace for an ITree where the outputs of the sequence of events is the same as the trace for the Kripke structure
- For all traces defined for ITrees, there exists a trace for the Kripke structure

where the outputs of the sequence of events matches some trace for the Kripke structure.

While the original goal was to formalise this in Isabelle/HOL, given how the events are encoded differently in Isabelle/HOL as opposed to how events were encoded in Coq, it may not be possible to define the function we want.

Chapter 5

Conclusion

The first half of the thesis was spent on formalising ITrees in Isabelle/HOL, where we implement the coinductive data structure and verify desired properties. Midway though the course of this thesis, Foster et al. (2021) have independently formalised ITrees in Isabelle/HOL with the results published. Thereafter, we decided to investigate how ITrees can be used in the context of model checking rather than proceeding with a duplicate formalisation. As an initial step, we have investigated how Kripke structures can be represented using ITrees.

In summary, we have complete the following tasks:

- We implemented ITrees in Isabelle/HOL with restricted event types
- We implemented monadic operators for ITrees and proved that ITrees are monads
- We attempted formalising weak bisimulation for ITrees and partially proved that the relation is an equivalence relation.

After the change in plans, we have completed the following tasks:

- We discovered how a Kripke structure unfolded into a computation tree may be represented as an ITree

- We discovered issues that prevent a straightforward transformation from Kripke structures to ITrees and provided potential solutions to these problem
- We proposed how a Kripke structure and the corresponding ITree may be related by examining the traces of both structures.

5.1 Future Work

While working on the new topic, there were some other ideas that seemed to be more realistic to attempt:

- As long as there is a formalisation of Kripke structures and CTL in Coq, it seems more reasonable to formalise the transformation in Coq, as the event types are given in a way more familiar to users.
- Rather than transforming Kripke structures to Interaction Trees, it may also be possible to do this the other way around, that is, given an Interaction Tree, transform this into a labelled computation tree and check whether some CTL formula is satisfied by the computation tree. This work seems to be much more practical, as this now implies that ITree behaviours can now be modelled using CTL formulae.

Bibliography

Moez A. AbdelGawad. Induction, Coinduction, and Fixed Points: Intuitions and Tutorial, 2019. URL `https://arxiv.org/abs/1903.05127`.

Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series).* The MIT Press, 2008. ISBN 026202649X.

Julian Biendarra, Jasmin Blanchette, Martin Desharnais, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. Defining (Co)datatypes and Primitively (Co)recursive Functions in Isabelle/HOL, 2013. URL `http://isabelle.in.tum.de/dist/doc/datatypes.pdf`.

Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. Truly Modular (Co)datatypes for Isabelle/HOL. In *Interactive Theorem Proving*, pages 93–110, Cham, 2014. Springer International Publishing. ISBN 978-3-319-08970-6.

Jasmin Christian Blanchette, Aymeric Bouzy, Andreas Lochbihler, Andrei Popescu, and Dmitriy Traytel. Friends with benefits. In *Proceedings of the 26th European Symposium on Programming Languages and Systems - Volume 10201*, page 111–140, Berlin, Heidelberg, 2017. Springer-Verlag. ISBN 9783662544334. doi: 10.1007/978-3-662-54434-1_5. URL `https://doi.org/10.1007/978-3-662-54434-1_5`.

Edmund M. Clarke. *The Birth of Model Checking*, pages 1–26. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-69850-0. doi: 10.1007/978-3-540-69850-0_1. URL `https://doi.org/10.1007/978-3-540-69850-0_1`.

Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, pages 52–71, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg. ISBN 978-3-540-39047-3.

E.M. Clarke, E.M.C.O.G.D. Peled, O. Grumberg, D. Peled, and EBSCO. *Model Checking.* The Cyber-Physical Systems Series. MIT Press, 1999. ISBN 9780262032704. URL `https://books.google.com.au/books?id=Nmc4wEaLXFEC`.

Simon Foster, Chung-Kil Hur, and Jim Woodcock. Formally verified simulations of state-rich processes using interaction trees in isabelle/hol. *CoRR*, abs/2105.05133, 2021. URL `https://arxiv.org/abs/2105.05133`.

John Hughes. Generalising Monads to Arrows. *Science of computer programming*, 37 (1-3):67–111, 2000.

Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems.* Cambridge University Press, USA, 2004. ISBN 052154310X.

Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. From C to Interaction Trees: Specifying, Verifying, and Testing a Networked Server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2019, page 234–248, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362221. doi: 10.1145/3293880.3294106. URL `https://doi.org/10.1145/3293880.3294106`.

Dexter Kozen and Alexandra Silva. Practical Coinduction. *Mathematical Structures in Computer Science*, 27(7):1132–1152, 2017. doi: 10.1017/S0960129515000493.

Saul A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94, 1963.

Ondřej Kunčar and Andrei Popescu. Comprehending Isabelle/HOL's Consistency. In *Programming Languages and Systems*, pages 724–749, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg. ISBN 978-3-662-54434-1.

Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.*, 3(2):125–143, 1977. ISSN 0098-5589. doi: 10.1109/TSE.1977.229904. URL `https://doi.org/10.1109/TSE.1977.229904`.

Andreas Lochbihler. Coinductive. *Archive of Formal Proofs*, February 2010. ISSN 2150-914x. `http://isa-afp.org/entries/Coinductive.html`, Formal proof development.

Andreas Lochbihler and Marc Züst. Programming TLS in Isabelle/HOL. In *Isabelle Workshop*, 2014. URL `http://www.andreas-lochbihler.de/pub/lochbihler14iw.pdf`.

Tobias Nipkow and Gerwin Klein. *Concrete Semantics: With Isabelle/HOL.* Springer Publishing Company, Incorporated, 2014. ISBN 3319105418.

Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic.* Springer-Verlag, Berlin, Heidelberg, 2002. ISBN 3540433767.

Davide Sangiorgi. *Introduction to Bisimulation and Coinduction.* Cambridge University Press, USA, 2011. ISBN 1107003636.

Freek Wiedijk. Comparing Mathematical Provers. In *Mathematical Knowledge Management*, pages 188–202, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-36469-6.

Freek Wiedijk. *The Seventeen Provers of the World.* Springer-Verlag, Berlin, Heidelberg, 2006. ISBN 3540307044.

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction Trees: Representing Recursive and Impure Programs in Coq. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi: 10.1145/3371119. URL `https://doi.org/10.1145/3371119`.