Calculating Compilers for Concurrency

PATRICK BAHR, IT University of Copenhagen, Denmark
GRAHAM HUTTON, University of Nottingham, UK

Choice trees [Chappe et al. 2023] have recently been introduced as a general structure for giving semantics to programming languages with a wide variety of features and effects. In this article we focus on concurrent languages, and show how a codensity version of choice trees allows the semantics for such languages to be systematically transformed into compilers using equational reasoning techniques. The codensity construction is the key ingredient that enables a high-level, algebraic approach. As a case study, we calculate a compiler for a concurrent lambda calculus with channel-based communication.

Additional Key Words and Phrases: program calculation, concurrency, choice trees, codensity monad

1 INTRODUCTION

Compilers are hard to write, and hard to get right. This is particularly so in the case of concurrent languages, where the addition of language primitives that introduce non-determinism make it significantly more challenging to develop and verify compilers.

One approach to compiler *verification* for concurrent languages is to define the semantics for both the source and target languages by translation into a lower-level concurrent language with suitable reasoning principles, such as bisimilarity and coinduction. This approach was pioneered by Wand [1995], who introduced the idea of translating into a process calculus, and has recently taken a step forward with the development of *choice trees* [Chappe et al. 2023], which provide a monadic language for expressing concurrency that supports modular, algebraic reasoning principles.

Such reasoning principles also make choice trees a suitable foundation for compiler *calculation*, a program synthesis technique that aims to derive correct-by-construction compilers from specifications of their correctness [Bahr and Hutton 2015, 2022]. The nature of the semantic reasoning principles is important for compiler calculation, because the aim is not only to produce correct compilers, but it also to *discover* compilation techniques. Simple, equational-style reasoning and a powerful (co-)induction principle are key features to enable this discovery.

In this article, we show how choice trees can be used as the semantic basis for compiler calculation for concurrent languages. In particular, the article makes the following contributions:

- We adapt the syntax and semantics of choice trees to enable the simple (co-)induction principle that powers the compiler calculation technique (Section 2).
- We identify a limitation of choice trees for defining the semantics of a simple concurrent language (Section 3), and show how the codensity construction can be applied to the choice tree monad to remove this limitation (Section 4).
- We present reasoning principles for the resulting *codensity choice trees* (Section 5), and show how they can be used to calculate a compiler for the simple language (Section 6).
- Finally, to demonstrate that our methodology scales to richly-featured concurrent languages, we show how to calculate a compiler for an untyped lambda calculus extended with concurrency and channel-based communication (Sections 7 and 8).

We use Haskell notation as our meta-language for accessibility, but assume that the language is total. Whereas in many articles calculations are often omitted or compressed for brevity, here they are the central focus so are typically presented in detail. All the calculations have been mechanically checked in Agda, and the proof scripts are available as online supplementary material.

Authors' addresses: Patrick Bahr, IT University of Copenhagen, Denmark; Graham Hutton, University of Nottingham, UK.

2 CHOICE TREES

In this section we introduce the basic concept of choice trees, show how they can be given a small-step operational semantics, and define the derived notion of parallel composition. To support the equational approach to reasoning that is used in compiler calculation, the syntax and semantics that we adopt for choice trees is different from the original article [Chappe et al. 2023]. As the full definitions for choice trees are developed in stages, we defer a discussion of these differences and their importance for compiler calculation until later on (Section 9).

2.1 Syntax

The type of choice trees *CTree e a* represents non-deterministic computations that return values of type a and that may use algebraic effects defined by the type function $e :: * \to *$:

```
data CTree e a where
  Now  :: a → CTree e a
  (⊕)  :: CTree e a → CTree e a → CTree e a
  Zero :: CTree e a
  Eff  :: e b → (b → CTree e a) → CTree e a
```

Informally, $Now\ v$ returns the value v without performing any effects, $p \oplus q$ makes a non-deterministic choice between two computations p and q, while $Zero$ is a computation that has terminated, and $Eff\ o\ c$ is a form of sequencing that feeds the result value produced by an effectful computation o into a continuation c. For example, if we wished to provide an effectful operation that prints an integer, this can be achieved by first defining a type function *PrintEff* that provides a single constructor called *PrintInt*, which is then used to define a *print* function:

```
data PrintEff a where
  PrintInt :: Int → PrintEff ()
print :: Int → CTree PrintEff ()
print n = Eff (PrintInt n) Now
```

Note that *print* does not actually print an integer, but rather builds a choice tree that represents the action of printing. Choice trees form a monad, with *return* and $\gg=$ defined as follows, which allows Haskell's **do** notation to be used to streamline the construction of choice trees:

```
return :: a → CTree e a
return = Now
(>>=) ::CTree e a → (a → CTree e b) → CTree e b
Now v  >>= f = f v
(p ⊕ q) >>= f = (p >>= f) ⊕ (q >>= f)
Zero   >>= f = Zero
Eff o c >>= f = Eff o (λi → c i >>= f)
```

We will also make use of the functorial map function, which is derived from $\gg=$ and *return*:

```
fmap :: (a → b) → CTree e a → CTree e a
fmap f p = p >>= (λx → return (f x))
```

Later on we will extend the notion of choice trees to support infinite (non-terminating) computations, but the above definitions will suffice for now.

$$\frac{}{Now\ v \xrightarrow{v} Zero} \qquad \frac{p \xrightarrow{l} p'}{p \oplus q \xrightarrow{l} p'} \qquad \frac{q \xrightarrow{l} q'}{p \oplus q \xrightarrow{l} q'}$$

$$\frac{o :: e\ b \quad c :: b \to CTree\ e\ a}{Eff\ o\ c \xrightarrow{\uparrow o} c} \qquad \frac{c :: b \to CTree\ e\ a \quad i :: b}{c \xrightarrow{\downarrow i} c\ i}$$

Fig. 1. Transition semantics for (codensity) choice trees.

2.2 Semantics

We define the semantics for choice trees by means of a labelled transition system. In our setting, a state for the transition system is either a choice tree $p :: CTree\ e\ a$, or a continuation $c :: b \to CTree\ e\ a$ that is waiting for an external input of type b in response to an effect of type $e\ b$. Transitions between states are labelled by one of four possible forms:

τ	a silent transition	$\uparrow o$	an effect $o :: e\ b$ for some type b
v	a return value $v :: a$	$\downarrow i$	an input $i :: b$ for some type b

Using these ideas, we define a labelled transition relation by the inference rules shown in Figure 1, which makes precise the informal meaning of choice trees from the previous section. Note that there is no rule for $Zero$ because it represents a terminated computation that can make no further transitions. The silent transition τ plays no role yet, but will be used later on.

By way of example, the expression $return\ 1 \oplus (print\ 2 \gg \lambda() \to return\ 3)$ expands to the choice tree $Now\ 1 \oplus Eff\ (PrintInt\ 2)\ (\lambda() \to Now\ 3)$, which has two possible transition sequences because of the use of the choice operator. In particular, it can either simply return the value 1,

$$Now\ 1 \oplus Eff\ (PrintInt\ 2)\ (\lambda() \to Now\ 3) \xrightarrow{1} Zero$$

or it can print 2, consume the resulting unit value (), and return the value 3:

$$Now\ 1 \oplus Eff\ (PrintInt\ 2)\ (\lambda() \to Now\ 3) \xrightarrow{\uparrow PrintInt\ 2} \lambda() \to Now\ 3 \xrightarrow{\downarrow()} Now\ 3 \xrightarrow{3} Zero$$

As illustrated in this latter transition sequence, every effectful transition is always immediately followed by an input transition that consumes the resulting value:

$$Eff\ o\ c \xrightarrow{\uparrow o} c \xrightarrow{\downarrow i} c\ i$$

Hence, we could simplify the semantics by combining the two transitions into one:

$$Eff\ o\ c \xrightarrow{\uparrow o \downarrow i} c\ i$$

While this approach has the benefit of avoiding the need for two kinds of states in the semantics, it results in a notion of bisimilarity that is too coarse. We return to this issue in Section 9 in our comparison to the work of Chappe et al. [2023], who do use this simplified semantics. Nonetheless, as such transitions always occur in pairs, it is useful to define a relation that combines them:

$$p \xRightarrow{\uparrow o \downarrow i} q \quad \text{iff} \quad \exists c.\ p \xrightarrow{\uparrow o} c \wedge c \xrightarrow{\downarrow i} q$$

2.3 Parallelism

Choice trees do not have a built-in notion of parallelism, as this can be derived from the other primitives. In particular, we can define parallel composition using three auxiliary operators:

$$
\begin{aligned}
&(\|) :: \mathit{CTree}\ e\ a \to \mathit{CTree}\ e\ b \to \mathit{CTree}\ e\ (a, b)\\
&p \parallel q = (p \lhd q) \oplus (p \rhd q) \oplus (p \bowtie q)
\end{aligned}
$$

The first operator $\lhd$ allows its left argument to perform an effectful computation:

$$
\begin{aligned}
&(\lhd) :: \mathit{CTree}\ e\ a \to \mathit{CTree}\ e\ b \to \mathit{CTree}\ e\ (a, b)\\
&\mathit{Now}\ v \quad \lhd q = \mathit{Zero}\\
&(p_1 \oplus p_2) \lhd q = (p_1 \lhd q) \oplus (p_2 \lhd q)\\
&\mathit{Zero} \quad \lhd q = \mathit{Zero}\\
&\mathit{Eff}\ o\ c \quad \lhd q = \mathit{Eff}\ o\ (\lambda i \to c\ i \parallel q)
\end{aligned}
$$

The second operator $\rhd$ does the same for the right argument, and is defined symmetrically to $\lhd$. The final operator $\bowtie$ allows both argument choice trees to perform computations simultaneously, with the resulting return values from each side being combined as a pair:

$$
\begin{aligned}
&(\bowtie) :: \mathit{CTree}\ e\ a \to \mathit{CTree}\ e\ b \to \mathit{CTree}\ e\ (a, b)\\
&(p_1 \oplus p_2) \bowtie q \quad\quad = (p_1 \bowtie q) \oplus (p_2 \bowtie q)\\
&p \quad\quad\quad \bowtie (q_1 \oplus q_2) = (p \bowtie q_1) \oplus (p \bowtie q_2)\\
&\mathit{Now}\ v \quad \bowtie \mathit{Now}\ w \quad = \mathit{Now}\ (v, w)\\
&\_ \quad\quad\quad \bowtie \_ \quad\quad = \mathit{Zero}
\end{aligned}
$$

The behaviour of $\|$ can be concisely characterised by the following inference rules, which together express that parallel composition has the expected behaviour:

$$
\frac{p \overset{\uparrow o \downarrow i}{\Longrightarrow} p'}{p \parallel q \overset{\uparrow o \downarrow i}{\Longrightarrow} p' \parallel q}
\qquad
\frac{q \overset{\uparrow o \downarrow i}{\Longrightarrow} q'}{p \parallel q \overset{\uparrow o \downarrow i}{\Longrightarrow} p \parallel q'}
\qquad
\frac{p \overset{v}{\Longrightarrow} \mathit{Zero} \qquad q \overset{w}{\Longrightarrow} \mathit{Zero}}{p \parallel q \overset{(v,w)}{\Longrightarrow} \mathit{Zero}}
$$

Moreover, these rules are complete, in the sense that any transition from $p \parallel q$ can be derived using them. Later we will consider more general situations in which both arguments may perform an effectful computation simultaneously, such as when one sends a message to the other.

It is also useful to define a variant of parallel composition that discards any result values produced by the left argument, hence this argument is only executed for its effects:

$$
\begin{aligned}
&(\vec{\shortparallel}) :: \mathit{CTree}\ e\ a \to \mathit{CTree}\ e\ b \to \mathit{CTree}\ e\ b\\
&p \vec{\shortparallel} q = \mathit{fmap}\ \mathit{snd}\ (p \parallel q)
\end{aligned}
$$

Right-biased parallel composition $\vec{\shortparallel}$ can be characterised in a similar way to $\|$, except that the inference rule for values only propogates the right value w rather than the pair (v, w).

3 EXAMPLE LANGUAGE

In this section, we introduce a simple concurrent language that we will use as an initial example for presenting our compiler calculation technique, and show how a semantics for the language can be defined in terms of choice trees. As we shall see, some care is required to ensure that the semantics correctly captures the intended concurrent behaviour.

3.1 Syntax

We consider a minimal expression language that comprises arithmetic expressions built up from integers values using an addition operator, extended with a primitive that prints the value of an expression, and a primitive that forks the evaluation of an expression:

data *Expr* = *Val Int* | *Add Expr Expr* | *Print Expr* | *Fork Expr*

Informally, *Fork e* starts evaluation of the expression *e* using a new concurrent process and immediately returns the result value 0, in a manner reminiscent of Haskell's *forkIO* primitive [Jones et al. 1996]. While the above language is not suitable for actual programming, it provides just what we need to explain our compiler calculation technique. In particular, the integers provide a simple notion of value, addition provides a simple notion of (sequential) computation, print provides a simple form of observable effect, and fork provides a simple form of concurrency.

3.2 Semantics

Using the choice tree machinery that was introduced in Section 2, a semantics for our simple expression language can be defined in terms of choice trees as follows:

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Expr} \to \mathit{CTree}\ \mathit{PrintEff}\ \mathit{Int}\\
&\mathit{eval}\ (\mathit{Val}\ n) \quad\ = \mathit{return}\ n\\
&\mathit{eval}\ (\mathit{Add}\ x\ y) = \mathbf{do}\ n \leftarrow \mathit{eval}\ x\,;\ m \leftarrow \mathit{eval}\ y\,;\ \mathit{return}\ (n+m)\\
&\mathit{eval}\ (\mathit{Print}\ x) \ = \mathbf{do}\ n \leftarrow \mathit{eval}\ x\,;\ \mathit{print}\ n\,;\ \mathit{return}\ n\\
&\mathit{eval}\ (\mathit{Fork}\ x) \ = \mathit{eval}\ x \mathbin{\vec{\shortparallel}} \mathit{return}\ 0
\end{aligned}
$$

The first three cases are as we would expect, while the case for fork formalises the idea that the argument expression is evaluated in parallel with returning the result 0. While this semantics for fork is simple, unfortunately it does not capture the desired behaviour. The problem is that the $\vec{\shortparallel}$ operator is *synchronous*, in the sense that it waits for both sides to complete.

To illustrate the problem, consider the expression $\mathit{Add}\ (\mathit{Fork}\ x)\ y$ with semantics:

$$(\mathit{eval}\ x \mathbin{\vec{\shortparallel}} \mathit{return}\ 0) \gg\!= \lambda n \to \mathit{eval}\ y \gg\!= \lambda m \to \mathit{return}\ (n+m)$$

We would expect that x and y are evaluated in parallel — that is the point of using fork. However, in the above choice tree y is only evaluated after $\mathit{eval}\ x \mathbin{\vec{\shortparallel}} \mathit{return}\ 0$ has completed, and hence only after x is evaluated. Instead, we would expect the semantics of $\mathit{Add}\ (\mathit{Fork}\ x)\ y$ to be

$$\mathit{eval}\ x \mathbin{\vec{\shortparallel}} (\mathit{return}\ 0 \gg\!= \lambda n \to \mathit{eval}\ y \gg\!= \lambda m \to \mathit{return}\ (n+m))$$

which then simplifies to $\mathit{eval}\ x \mathbin{\vec{\shortparallel}} \mathit{eval}\ y$. Here, evaluation of x has been floated to the top-level, which ensures that it takes place *asynchonously*. In Haskell [Jones et al. 1996], this behaviour is realised by defining the semantics using an evaluation context that captures where the next step of evaluation takes place. In our setting, this gives the following semantics for fork:

$$\mathit{eval}\ (C[\,\mathit{Fork}\ x\,]) \ = \ \mathit{eval}\ x \mathbin{\vec{\shortparallel}} \mathit{eval}\ (C[\,\mathit{Val}\ 0\,])$$

That is, if we are evaluating in a context C for which the next step is to fork an expression x, then we simply float evaluation of x to the top level, and continue with this expression replaced by the value zero. However, the above semantics is no longer compositional, because the semantics of $\mathit{Fork}\ x$ is no longer defined purely in terms of the semantics of x, but also involves the semantics for the residual expression $C[\,\mathit{Val}\ 0\,]$. Our compiler calculation methodology depends on the semantics being compositional, so we can't use the contextual approach here.

Fortunately, we can achieve the same effect as the contextual semantics while retaining compositionality by rewriting the semantics in continuating-passing style. In particular, we take an

additional argument c — the continuation — that is used to process of result of evaluating an expression, and hence captures the idea of what to do after the current evaluation:

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Expr} \to (\mathit{Int} \to \mathit{CTree}\ \mathit{PrintEff}\ \mathit{Int}) \to \mathit{CTree}\ \mathit{PrintEff}\ \mathit{Int}\\
&\mathit{eval}\ (\mathit{Val}\ n) \quad\ \ c = c\ n\\
&\mathit{eval}\ (\mathit{Add}\ x\ y)\ c = \mathit{eval}\ x\ (\lambda n \to \mathit{eval}\ y\ (\lambda m \to c\ (n + m)))\\
&\mathit{eval}\ (\mathit{Print}\ x)\ \ c = \mathit{eval}\ x\ (\lambda n \to \mathit{print}\ n \gg\!= \lambda() \to c\ n)\\
&\mathit{eval}\ (\mathit{Fork}\ x)\ \ \ c = \mathit{eval}\ x\ \mathit{return} \mathbin{\vec{\shortparallel}} c\ 0
\end{aligned}
$$

This definition ensures that any continuation c that follows on from $\mathit{Fork}\ x$ is in evaluated in parallel with $\mathit{eval}\ x\ \mathit{return}$. For example, the expression $\mathit{Add}\ (\mathit{Fork}\ x)\ y$ has the semantics $\mathit{eval}\ x\ \mathit{return} \mathbin{\vec{\shortparallel}} \mathit{eval}\ y\ c$, which ensures that x and y are evaluated in parallel as expected.

4 CODENSITY CHOICE TREES

While the continuation semantics in the previous section captures the intended behaviour, it is not so appealing as the simple, but incorrect, monadic semantics that we originally presented. More importantly, the explicit use of continuations would complicate the reasoning process, as we discuss later on (Section 6). However, we can regain both the simplicity of the monadic semantics and its reasoning principles by first generalising the return type of *eval*,

$$(\mathit{Int} \to \mathit{CTree}\ \mathit{PrintEff}\ \mathit{Int}) \to \mathit{CTree}\ \mathit{PrintEff}\ \mathit{Int}$$

from the specific case of integers to an arbitrary input type a and result type r,

$$(a \to \mathit{CTree}\ \mathit{PrintEff}\ r) \to \mathit{CTree}\ \mathit{PrintEff}\ r$$

and then observing that this is in fact the codensity monad for $\mathit{CTree}\ \mathit{PrintEff}$. The *codensity monad* [Voigtländer 2008] is similar to the familiar continuation monad, except that rather than having a fixed result type r, it has a variable (polymorphic) result type. Based on the generalised return type for *eval*, we can define a type of codensity choice trees $\mathit{CTree}_\mathrm{c}\ e\ a$ as follows:

$$\textbf{type}\ \mathit{CTree}_\mathrm{c}\ e\ a = \mathit{forall}\ r\ .\ (a \to \mathit{CTree}\ e\ r) \to \mathit{CTree}\ e\ r$$

Note that the return type r is universally quantified on the right-hand side of the declaration, in contrast to the continuation monad where r is a parameter on the left-hand side. Codensity choice trees form a monad, with the *return* and $\gg\!=$ primitives defined as follows:

$$
\begin{aligned}
&\mathit{return} :: a \to \mathit{CTree}_\mathrm{c}\ e\ a\\
&\mathit{return}\ v = \lambda c \to c\ v
\end{aligned}
$$

$$
\begin{aligned}
&\gg\!= :: \mathit{CTree}_\mathrm{c}\ e\ a \to (a \to \mathit{CTree}_\mathrm{c}\ e\ b) \to \mathit{CTree}_\mathrm{c}\ e\ b\\
&p \gg\!= f = \lambda c \to p\ (\lambda v \to f\ v\ c)
\end{aligned}
$$

In turn, we can also define $\mathit{CTree}_\mathrm{c}$ versions of the non-deterministic choice primitives, effect sequencing and printing, and the two versions of parallel composition:

$$
\begin{aligned}
&(\oplus_\mathrm{c}) :: \mathit{CTree}_\mathrm{c}\ e\ a \to \mathit{CTree}_\mathrm{c}\ e\ a \to \mathit{CTree}_\mathrm{c}\ e\ a\\
&p \oplus_\mathrm{c} q = \lambda c \to p\ c \oplus q\ c
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Zero}_\mathrm{c} :: \mathit{CTree}_\mathrm{c}\ e\ a\\
&\mathit{Zero}_\mathrm{c} = \lambda c \to \mathit{Zero}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Eff}_\mathrm{c} :: e\ b \to (b \to \mathit{CTree}_\mathrm{c}\ e\ a) \to \mathit{CTree}_\mathrm{c}\ e\ a\\
&\mathit{Eff}_\mathrm{c}\ o\ k = \lambda c \to \mathit{Eff}\ o\ (\lambda v \to k\ v\ c)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{print}_\mathrm{c} :: \mathit{Int} \to \mathit{CTree}_\mathrm{c}\ \mathit{PrintEff}\ ()\\
&\mathit{print}_\mathrm{c}\ n = \mathit{Eff}_\mathrm{c}\ (\mathit{PrintInt}\ n)\ \mathit{return}
\end{aligned}
$$

$$(\|_c) :: CTree_c\ e\ a \to CTree_c\ e\ b \to CTree_c\ e\ (a, b)$$
$$p \|_c q = \lambda c \to (p\ return \parallel q\ return) \gg\!= c$$
$$(\vec{\amalg}_c) :: CTree_c\ e\ a \to CTree_c\ e\ b \to CTree_c\ e\ b$$
$$p\ \vec{\amalg}_c\ q = \lambda c \to p\ return\ \vec{\amalg}\ q\ c$$

Using these operations, it is now straightforward to redefine the continuation semantics from the previous section using the notion of codensity choice trees:

$$\begin{aligned}
&eval :: Expr \to CTree_c\ PrintEff\ Int\\
&eval\ (Val\ n) &&= return\ n\\
&eval\ (Add\ x\ y) &&= \mathbf{do}\ n \leftarrow eval\ x;\ m \leftarrow eval\ y;\ return\ (n + m)\\
&eval\ (Print\ x) &&= \mathbf{do}\ n \leftarrow eval\ x;\ print_c\ n;\ return\ n\\
&eval\ (Fork\ x) &&= eval\ x\ \vec{\amalg}_c\ return\ 0
\end{aligned}$$

This definition regains the simplicity of our original monadic definition of *eval* from Section 3.2, but now correctly captures the intended semantics. Each $CTree_c$ represents a *CTree*, which can be obtained simply by passing *return* for choice trees as the continuation:

$$ctree :: CTree_c\ e\ a \to CTree\ e\ a$$
$$ctree\ p = p\ return$$

Using this translation function, the labelled transition system that defines the semantics for choice trees *CTree* can be lifted to codensity choice trees $CTree_c$, and satisfies the same rules as Figure 1 with the operations replaced by the corresponding codensity versions.

We conclude by noting that codensity choice trees are represented in a slighly different manner in our Agda formalisation, namely as a datatype with the operations *return*, $\gg\!=$, $\oplus_c$, etc as constructors, together with an interpretation function that converts each constructor to the corresponding operation above. This approach ensures that the termination checker accepts the definition for *eval*, and allows us to prove laws we will need later on, such as the congruence law for $\gg\!=$.

5 BISIMILARITY AND LAWS

In this section we introduce the notion of bisimilarity for (codensity) choice trees, together with a number of laws for the operations on such trees that we will need for compiler calculation. We begin by considering choice trees, then extend to codensity choice trees.

Because the semantics of choice trees *CTree e a* is defined using a labelled transition relation, the notion of bisimilarity $\cong$ can be defined in the standard way. In particular, $\cong$ is the largest relation on choice trees (and continuations) that satisfies the following two properties:

$$\text{If } p \cong q \text{ and } p \xrightarrow{l} p', \text{ then there is some } q' \text{ with } q \xrightarrow{l} q' \text{ and } p' \cong q'$$
$$\text{If } p \cong q \text{ and } q \xrightarrow{l} q', \text{ then there is some } p' \text{ with } p \xrightarrow{l} p' \text{ and } p' \cong q'$$

Bisimilarlity is an equivalance relation, i.e. reflexive, symmetric and transitive. Moreover, all operations on choice trees that we have defined satisfy congruence laws with the respect to bisimilarity. For example, for the monadic bind operator this means that:

$$\frac{p \cong q \qquad f\ x \cong g\ x\ \text{ for all } x}{p \gg\!= f\ \ \cong\ \ q \gg\!= g}$$

Other relevant laws for choice trees are given in Figure 2. In particular, choice trees form a monad, and an idempotent, commutative monoid under $\oplus$ with *Zero* as the unit. As expected, $\vec{\amalg}$ satisfies associativity and commutativity laws. However, because return values matter, commutativity only

$$
\begin{array}{rcll}
\mathit{return}\ x \gg\!= f & \cong & f\ x & \\
p \gg\!= \mathit{return} & \cong & p & \text{(monad laws)} \\
(p \gg\!= f) \gg\!= g & \cong & p \gg\!= \lambda x \to (f\ x \gg\!= g) &
\end{array}
$$

$$
\begin{array}{rcll}
\mathit{Zero} \oplus p & \cong & p & (\oplus\text{-}\mathit{Zero}) \\
p \oplus p & \cong & p & (\oplus\text{-idem}) \\
p \oplus q & \cong & q \oplus p & (\oplus\text{-comm}) \\
(p \oplus q) \oplus r & \cong & p \oplus (q \oplus r) & (\oplus\text{-assoc}) \\
(p \oplus q) \gg\!= f & \cong & (p \gg\!= f) \oplus (q \gg\!= f) & (\oplus\text{-bind}) \\
\mathit{Zero} \gg\!= f & \cong & \mathit{Zero} & (\mathit{Zero}\text{-bind})
\end{array}
\qquad
\begin{array}{rcll}
\mathit{return}\ v \mathbin{\vec{\shortparallel}} p & \cong & p & (\vec{\shortparallel}\text{-}\mathit{return}) \\
\mathit{fmap}\ f\ (p \mathbin{\vec{\shortparallel}} q) & \cong & (\mathit{fmap}\ g\ p) \mathbin{\vec{\shortparallel}} (\mathit{fmap}\ f\ q) & (\vec{\shortparallel}\text{-}\mathit{fmap}) \\
(p \mathbin{\vec{\shortparallel}} q) \mathbin{\vec{\shortparallel}} r & \cong & p \mathbin{\vec{\shortparallel}} (q \mathbin{\vec{\shortparallel}} r) & (\vec{\shortparallel}\text{-assoc}) \\
(p \mathbin{\vec{\shortparallel}} q) \mathbin{\vec{\shortparallel}} r & \cong & (q \mathbin{\vec{\shortparallel}} p) \mathbin{\vec{\shortparallel}} r & (\vec{\shortparallel}\text{-comm})
\end{array}
$$

Fig. 2. Laws for (codensity) choice trees.

applies to the left argument of $\vec{\shortparallel}$, for which return values are discarded. An important observation is that $\gg\!=$ does not distribute over $\vec{\shortparallel}$, due to the synchronous nature of parallel composition for choice trees. In particular, with the expression $(p \mathbin{\vec{\shortparallel}} q) \gg\!= f$ the computation f can only start once both p and q have completed, whereas with $p \mathbin{\vec{\shortparallel}} (q \gg\!= f)$ the computation f can start as soon as q is complete and hence can run in parallel to p. Instead, distributivity is restricted to *fmap*.

We now turn our attention to codensity choice trees. Using the translation function $\mathit{ctree} :: \mathit{CTree}_c\ e\ a \to \mathit{CTree}\ e\ a$ from the previous section, codensity choice trees inherit the notion of bisimilarity from choice trees. However, to ensure that bisimilarity is a congruence for $\gg\!=$, we have to strengthen the notion of bisimilarity for codensity choice trees by defining it as follows:

$$p \cong q \quad \text{iff} \quad p\ c \cong q\ c \ \text{ for all } c$$

That is, two codensity choice trees are bisimilar precisely when they are bisimilar as choice trees for every continuation. Using this definition, it follows that if $p \cong q$ for two codensity choice trees, then their translations are bisimilar as choice trees, i.e. $\mathit{ctree}\ p \cong \mathit{ctree}\ q$.

All laws in Figure 2 as well as the congruence laws carry over to codensity choice trees. In addition, we obtain the following distributivity law for parallel composition:

$$(p \mathbin{\vec{\shortparallel}_c} q) \gg\!= f \ \cong\ p \mathbin{\vec{\shortparallel}_c} (q \gg\!= f) \qquad (\vec{\shortparallel}_c\text{-bind})$$

As observed above, this law does not hold for $p \mathbin{\vec{\shortparallel}} q$ because the parallel computation is synchronous, whereas $p \mathbin{\vec{\shortparallel}_c} q$ is asynchronous in that it produces a result as soon as q has completed, rather than also waiting for p. The $\vec{\shortparallel}_c$-bind law captures this intuition formally. For example, this law can be used to show that $\mathit{Add}\ (\mathit{Fork}\ x)\ y$ has the intended semantics using codensity choice trees:

$$
\begin{array}{ll}
 & \mathit{eval}\ (\mathit{Add}\ (\mathit{Fork}\ x)\ y) \\
\cong & \quad \{\, \text{applying } \mathit{eval} \,\} \\
 & \mathit{eval}\ (\mathit{Fork}\ x) \gg\!= (\lambda n \to \mathit{eval}\ y \gg\!= \lambda m \to \mathit{return}\ (n + m)) \\
\cong & \quad \{\, \text{applying } \mathit{eval} \,\} \\
 & (\mathit{eval}\ x \mathbin{\vec{\shortparallel}_c} \mathit{return}\ 0) \gg\!= (\lambda n \to \mathit{eval}\ y \gg\!= \lambda m \to \mathit{return}\ (n + m)) \\
\cong & \quad \{\, \vec{\shortparallel}_c\text{-bind law} \,\} \\
 & \mathit{eval}\ x \mathbin{\vec{\shortparallel}_c} (\mathit{return}\ 0 \gg\!= \lambda n \to \mathit{eval}\ y \gg\!= \lambda m \to \mathit{return}\ (n + m)) \\
\cong & \quad \{\, \text{monad laws} \,\} \\
 & \mathit{eval}\ x \mathbin{\vec{\shortparallel}_c} \mathit{eval}\ y
\end{array}
$$

That is, x is evaluated in parallel with y, and the result produced by y is returned.

The crucial semantic difference between $\vec{\amalg}$ and $\vec{\amalg}_c$ discussed above also raises the question of whether we could define a version of $\vec{\amalg}$ on choice trees that *does* satisfy a distributivity law. Unfortunately, this is not possible. In general, with a parallel computation $p \,\vec{\amalg}\, q$ we expect that the effects of p may interact with the effects of q. Indeed, we extend parallel composition in Section 7 to allow such interaction by message passing. However, if the $\vec{\amalg}$-bind law holds, then for an expression $(p \,\vec{\amalg}\, q) \gg\!= f$ we should also expect that the effects of p may interact with the effects of $q \gg\!= f$. Clearly, this cannot be the case for an operator $\vec{\amalg}$ that can only inspect p and q.

6 COMPILER CALCULATION

We have now defined a datatype *Expr* that represents the syntax of a simple concurrent language, an evaluation function *eval* that gives a semantics for the language in terms of codensity choice trees, and a notion of bisimilarity for such trees. In this section, we show how to specify the desired behaviour of a compiler for the simple language, and how such a specification can be used as the basis for systematically calculating an implementation of the compiler.

6.1 Specification

Our goal is to define a compilation function $comp :: Expr \to Code$ that translates an expression into code for an (as yet unspecified) target language. We assume the compiler targets a stack-based machine, whose semantics is given by a function $exec :: Code \to Stack \to Stack$, where **type** $Stack = [Int]$ is the stack type for the machine. However, because our evaluation function defines the semantics of expressions in terms of codensity choice trees,

$$eval :: Expr \to CTree_c\ PrintEff\ Int$$

we also generalise the type of the execution function function to operate in the same monadic setting, i.e. within the codensity monad $CTree_c\ PrintEff$:

$$exec :: Code \to Stack \to CTree_c\ PrintEff\ Stack$$

The definitions for the *Code* datatype and *exec* function are not given up front, but will rather fall out naturally as part of the process of calculating the compiler itself.

Prior to specifying the desired behaviour of the compiler, we generalise the function *comp* to take additional code to be executed after the compiled code. The addition of such a *code continuation* is a key aspect of the methodology [Bahr and Hutton 2015], and significantly simplifies the resulting calculations. Using this idea, our goal now is to establish the following compiler correctness property for the generalised compilation function $comp : Expr \to Code \to Code$:

$$\mathbf{do}\ v \leftarrow eval\ e;\ exec\ c\ (v : s) \quad \cong \quad exec\ (comp\ e\ c)\ s$$

That is, compiling an expression and then executing the resulting code together with the supplied additional code should give the same result (up to bisimilarity) as executing the additional code with the value of the expression on top of the stack.

6.2 Calculation

The proof of the compiler correctness property proceeds by structural induction on the expression e. For each case, we start with the left-hand-side of the property, and seek to transform it by equational reasoning using the bisimilarity laws from Section 5 into the form $exec\ c'\ s$ for some code c'. We then define $comp\ e\ c = c'$, which gives us a clause for the compiler in this case that is guaranteed by construction to satisfy the correctness property. During such calculations we will also discover new constructors for the *Code* datatype and new clauses for the *exec* function, driven by the desire to transform the term that is being manipulated into the required form.

The cases for values and addition proceed in the same manner as [Bahr and Hutton 2022], except that because our language doesn't yet include non-termination, simple bisimilarity suffices rather than the more refined notion of step-indexed bisimilarity:

Case: $e = \mathit{Val}\ n$

$$
\begin{aligned}
&\mathbf{do}\ v \leftarrow \mathit{eval}\ (\mathit{Val}\ n)\,;\ \mathit{exec}\ c\ (v : s)\\
\cong\ &\quad \{\ \text{definition of } \mathit{eval}\ \}\\
&\mathbf{do}\ v \leftarrow \mathit{return}\ n\,;\ \mathit{exec}\ c\ (v : s)\\
\cong\ &\quad \{\ \text{monad laws}\ \}\\
&\mathit{exec}\ c\ (n : s)\\
\cong\ &\quad \{\ \text{define: } \mathit{exec}\ (\mathit{PUSH}\ n\ c)\ s = \mathit{exec}\ c\ (n : s)\ \}\\
&\mathit{exec}\ (\mathit{PUSH}\ n\ c)\ s
\end{aligned}
$$

In the final step above, we introduce a code constructor *PUSH* and a corresponding clause for *exec*. These definitions arise naturally from the fact that we are aiming for a term of the form $\mathit{exec}\ c'\ s$, and hence need to solve the equation $\mathit{exec}\ c'\ s \cong \mathit{exec}\ c\ (n : s)$. We solve this equation by strengthening bisimilarity to equality, i.e. $\mathit{exec}\ c'\ s = \mathit{exec}\ c\ (n : s)$, and instantiating c' to $\mathit{PUSH}\ n\ c$ so that the equation then becomes a defining clause for the function *exec*.

Case: $e = \mathit{Add}\ x\ y$

$$
\begin{aligned}
&\mathbf{do}\ v \leftarrow \mathit{eval}\ (\mathit{Add}\ x\ y)\,;\ \mathit{exec}\ c\ (v : s)\\
\cong\ &\quad \{\ \text{definition of } \mathit{eval}\ \}\\
&\mathbf{do}\ v \leftarrow \mathbf{do}\ \{m \leftarrow \mathit{eval}\ x\,;\ n \leftarrow \mathit{eval}\ y\,;\ \mathit{return}\ (m + n)\}\,;\ \mathit{exec}\ c\ (v : s)\\
\cong\ &\quad \{\ \text{monad laws}\ \}\\
&\mathbf{do}\ m \leftarrow \mathit{eval}\ x\,;\ n \leftarrow \mathit{eval}\ y\,;\ \mathit{exec}\ c\ ((m + n) : s)\\
\cong\ &\quad \{\ \text{define: } \mathit{exec}\ (\mathit{ADD}\ c)\ (n : m : s) = \mathit{exec}\ c\ ((m + n) : s)\ \}\\
&\mathbf{do}\ m \leftarrow \mathit{eval}\ x\,;\ n \leftarrow \mathit{eval}\ y\,;\ \mathit{exec}\ (\mathit{ADD}\ c)\ (n : m : s)\\
\cong\ &\quad \{\ \text{induction hypothesis for } y\ \}\\
&\mathbf{do}\ m \leftarrow \mathit{eval}\ x\,;\ \mathit{exec}\ (\mathit{comp}\ y\ (\mathit{ADD}\ c))\ (m : s)\\
\cong\ &\quad \{\ \text{induction hypothesis for } x\ \}\\
&\mathit{exec}\ (\mathit{comp}\ x\ (\mathit{comp}\ y\ (\mathit{ADD}\ c)))\ s
\end{aligned}
$$

In the third step above, we introduce another code constructor *ADD* and clause for *exec*. In particular, in order to apply the induction hypothesis for y, we need to transform the term $\mathit{exec}\ c\ ((m + n) : s)$ into the form $\mathit{exec}\ c'\ (n : s')$ for some code c' and stack s'. We achieve this instantiating $c' = \mathit{ADD}\ c$ and $s' = m : s$, and defining a corresponding new clause for *exec*. Applying the two induction hypotheses then completes the transformation into the required form.

The case for print is straightforward, once again introducing a new clause for *exec* that brings the term into the form that is required to apply the induction hypothesis:

Case: $e = \mathit{Print}\ x$

$$
\begin{aligned}
&\mathbf{do}\ v \leftarrow \mathit{eval}\ (\mathit{Print}\ x)\,;\ \mathit{exec}\ c\ (v : s)\\
\cong\ &\quad \{\ \text{definition of } \mathit{eval}\ \}\\
&\mathbf{do}\ v \leftarrow \mathbf{do}\ \{n \leftarrow \mathit{eval}\ x\,;\ \mathit{print}\ n\,;\ \mathit{return}\ n\}\,;\ \mathit{exec}\ c\ (v : s)\\
\cong\ &\quad \{\ \text{monad laws}\ \}\\
&\mathbf{do}\ n \leftarrow \mathit{eval}\ x\,;\ \mathit{print}\ n\,;\ \mathit{exec}\ c\ (n : s)\\
\cong\ &\quad \{\ \text{define: } \mathit{exec}\ (\mathit{PRINT}\ c)\ (n : s) = \mathbf{do}\ \mathit{print}\ n\,;\ \mathit{exec}\ c\ (n : s)\ \}\\
&\mathbf{do}\ n \leftarrow \mathit{eval}\ x\,;\ \mathit{exec}\ (\mathit{PRINT}\ c)\ (n : s)
\end{aligned}
$$

Target language

data *Code* = *PUSH Int Code* | *ADD Code* | *PRINT Code* | *FORK Code Code* | *HALT*

Compiler

$$
\begin{aligned}
&\mathit{compile} :: \mathit{Expr} \to \mathit{Code}\\
&\mathit{compile}\ e = \mathit{comp}\ e\ \mathit{HALT}\\
&\mathit{comp} :: \mathit{Expr} \to \mathit{Code} \to \mathit{Code}\\
&\mathit{comp}\ (\mathit{Val}\ n)\ \ c = \mathit{PUSH}\ n\ c\\
&\mathit{comp}\ (\mathit{Add}\ x\ y)\ c = \mathit{comp}\ x\ (\mathit{comp}\ y\ (\mathit{ADD}\ c))\\
&\mathit{comp}\ (\mathit{Print}\ x)\ \ c = \mathit{comp}\ x\ (\mathit{PRINT}\ c)\\
&\mathit{comp}\ (\mathit{Fork}\ x)\ \ c = \mathit{FORK}\ (\mathit{comp}\ x\ \mathit{HALT})\ c
\end{aligned}
$$

Virtual machine

$$
\begin{aligned}
&\mathit{exec} :: \mathit{Code} \to \mathit{Stack} \to \mathit{CTree}_{\mathrm{c}}\ \mathit{PrintEff}\ \mathit{Stack}\\
&\mathit{exec}\ (\mathit{PUSH}\ n\ c)\ s\ \ \ \ \ \ \ \ = \mathit{exec}\ c\ (n : s)\\
&\mathit{exec}\ (\mathit{ADD}\ c)\ (n : m : s) = \mathit{exec}\ c\ ((m + n) : s)\\
&\mathit{exec}\ (\mathit{PRINT}\ c)\ (n : s)\ \ = \mathbf{do}\ \mathit{print}\ n\,;\ \mathit{exec}\ c\ (n : s)\\
&\mathit{exec}\ (\mathit{FORK}\ c'\ c)\ s\ \ \ \ \ = \mathit{exec}\ c'\ [\,]\ \vec{\shortparallel}_{\mathrm{c}}\ \mathit{exec}\ c\ (0 : s)\\
&\mathit{exec}\ \mathit{HALT}\ s\ \ \ \ \ \ \ \ \ \ \ \ = \mathit{return}\ s
\end{aligned}
$$

Fig. 3. Compiler and virtual machine for the simple expression language.

$$
\begin{aligned}
&\cong \quad \{\ \text{induction hypothesis for } x\ \}\\
&\quad \mathit{exec}\ (\mathit{comp}\ x\ (\mathit{PRINT}\ c))\ s
\end{aligned}
$$

Finally, the case for fork first exploits the laws for right-biased parallel composition to introduce a new clause for *exec* that allows the induction hypothesis to be applied, and then introduces a further new clause to bring the resulting term into the required form:

Case: $e = \mathit{Fork}\ x$

$$
\begin{aligned}
&\quad \mathbf{do}\ v \leftarrow \mathit{eval}\ (\mathit{Fork}\ x)\,;\ \mathit{exec}\ c\ (v : s)\\
&\cong \quad \{\ \text{definition of } \mathit{eval}\ \}\\
&\quad \mathbf{do}\ v \leftarrow (\mathit{eval}\ x\ \vec{\shortparallel}_{\mathrm{c}}\ \mathit{return}\ 0)\,;\ \mathit{exec}\ c\ (v : s)\\
&\cong \quad \{\ \mathbf{do}\ \text{notation}\ \}\\
&\quad (\mathit{eval}\ x\ \vec{\shortparallel}_{\mathrm{c}}\ \mathit{return}\ 0) \gg\!= \lambda v \to \mathit{exec}\ c\ (v : s)\\
&\cong \quad \{\ \vec{\shortparallel}_{\mathrm{c}}\text{-bind law}\ \}\\
&\quad \mathit{eval}\ x\ \vec{\shortparallel}_{\mathrm{c}}\ (\mathit{return}\ 0 \gg\!= \lambda v \to \mathit{exec}\ c\ (v : s))\\
&\cong \quad \{\ \text{monad laws}\ \}\\
&\quad \mathit{eval}\ x\ \vec{\shortparallel}_{\mathrm{c}}\ \mathit{exec}\ c\ (0 : s)\\
&\cong \quad \{\ \vec{\shortparallel}_{\mathrm{c}}\text{-}\mathit{fmap}\ \text{law}\ \}\\
&\quad \mathit{fmap}\ (\lambda v \to [\,v\,])\ (\mathit{eval}\ x)\ \vec{\shortparallel}_{\mathrm{c}}\ \mathit{exec}\ c\ (0 : s)\\
&\cong \quad \{\ \text{definition of } \mathit{fmap}\ \}\\
&\quad (\mathbf{do}\ v \leftarrow \mathit{eval}\ x\,;\ \mathit{return}\ [\,v\,])\ \vec{\shortparallel}_{\mathrm{c}}\ \mathit{exec}\ c\ (0 : s)\\
&\cong \quad \{\ \text{define: } \mathit{exec}\ \mathit{HALT}\ s = \mathit{return}\ s\ \}\\
&\quad (\mathbf{do}\ v \leftarrow \mathit{eval}\ x\,;\ \mathit{exec}\ \mathit{HALT}\ [\,v\,])\ \vec{\shortparallel}_{\mathrm{c}}\ \mathit{exec}\ c\ (0 : s)\\
&\cong \quad \{\ \text{induction hypothesis for } x\ \}\\
&\quad (\mathit{exec}\ (\mathit{comp}\ x\ \mathit{HALT})\ [\,])\ \vec{\shortparallel}_{\mathrm{c}}\ \mathit{exec}\ c\ (0 : s)\\
&\cong \quad \{\ \text{define: } \mathit{exec}\ (\mathit{FORK}\ c'\ c)\ s = \mathit{exec}\ c'\ [\,]\ \vec{\shortparallel}_{\mathrm{c}}\ \mathit{exec}\ c\ (0 : s)\ \}\\
&\quad \mathit{exec}\ (\mathit{FORK}\ (\mathit{comp}\ x\ \mathit{HALT})\ c)\ s
\end{aligned}
$$

In summary, we have calculated the definitions in Figure 3, to which we add a top-level compilation function $\mathit{compile} :: \mathit{Expr} \to \mathit{Code}$ that simply applies *comp* and then halts.

6.3 Reflection

We conclude this section with some reflective remarks. First of all, note that the *exec* function is not total because addition and printing require stacks of specific forms. To make it total, we can add the catch-all case $exec\ \_\ \_ = Zero_c$, but the choice of semantics here is not important as compiler correctness does not depend on it. Secondly, the *exec* function returns a collection of parallel computations in which recursive calls are always tail calls, i.e. the final operation performed, which justifies referring to *exec* as a (parallel) virtual machine. More precisely, using the transition semantics we can observe that all transition sequences starting from $exec\ c\ s$ are of the form

$$exec\ c\ s \overset{\uparrow o_1 \downarrow i_1}{\Longrightarrow} p_1 \overset{\uparrow o_2 \downarrow i_2}{\Longrightarrow} p_2 \overset{\uparrow o_3 \downarrow i_3}{\Longrightarrow} \cdots \overset{s}{\Longrightarrow} Zero_c$$

where each p_i is bisimilar to an expression of the form

$$exec\ c_1\ s_1\ \vec{\parallel}_c\ exec\ c_2\ s_2\ \vec{\parallel}_c\ \cdots\ \vec{\parallel}_c\ exec\ c_{n+1}\ s_{n+1}$$

That is, the state of the virtual machine consists of $n + 1$ parallel threads of execution, each with its own code and stack. The result stack s from the whole execution is produced by the rightmost thread, while the remaining n threads are only executed for their effects, and the order of these threads does not matter according to the $\vec{\parallel}_c$-comm law. Using the $\vec{\parallel}_c$-*return* law, *HALT* kills the current thread, and using the $\vec{\parallel}_c$-assoc law, *FORK* spawns a new thread with an empty stack.

And finally, as shown in Section 3, we can capture the semantics of the source language using choice trees alone if we use a continuation-passing style, and indeed we can calculate a compiler based on this semantics. However, this approach comes with the drawback of having to prove ad-hoc lemmas about each semantics, e.g. congruence and distributivity properties such as:

$$\frac{c\ x \cong c'\ x \quad \text{for all } x}{eval\ e\ c \cong eval\ e\ c'} \qquad \frac{}{fmap\ f\ (eval\ e\ c) \cong eval\ e\ (\lambda x \to fmap\ f\ (c\ x))}$$

These properties suggest that the continuation-passing style semantics $eval\ e\ c$ behaves similarly to $eval\ e \gg\!= c$ for a suitably defined monad. Codensity choice trees make this idea precise, and provide an abstraction that satisfies these and other crucial structural properties *by construction*. In contrast, the continuation-passing style semantics makes essentially no use of the monadic structure of choice trees, because its only use of $\gg\!=$ can be replaced by continuation passing.

7 NON-TERMINATION AND EFFECT HANDLERS

For expository purposes, we have used a simplified version of (codensity) choice trees so far. We now give the full definition, which will allow us to consider non-terminating computations (Section 7.1). In addition, we introduce concurrent effect handlers on choice trees to allow us to consider concurrent computations that can interact, e.g. by sending and receiving messages (Section 7.2 and Section 7.3). The resulting semantic structure forms the basis for our compiler calculation for a concurrent lambda calculus with channel-based communication in Section 8.

7.1 Non-termination

To support non-termination, we extend *CTree* with an additional constructor *Step*, whose argument is a value of a coinductive type *CTreeInf* with a single constructor *Delay*:

```
data CTree e a where
  ...
  Step :: CTreeInf e a → CTree e a
codata CTreeInf e a = Delay (CTree e a)
```

That is, *CTree* is still an inductive type, but is now defined mutually recursively with a coinductive type *CTreeInf*, which we indicate by writing **codata** instead of **data**. In this manner, a value of type *CTree e a* is a potentially infinite tree with nodes labelled by the constructors *Now*, *Step* and so on, such that every infinite path from the root must contain infinitely many nodes labelled *Step*. Subsequently, we write *Later p* as a shorthand for *Step* (*Delay p*) and don't use *Step* or *Delay* directly. For example, a computation that never terminates can be defined as follows:

$$never :: CTree\ e\ a$$
$$never = Later\ never$$

Despite being non-terminating, this definition is total because the recursive call is guarded by *Later*, and system such as Agda will accept it. The transition semantics for choice trees, and hence the notion of bisimilarity, is extended to account for non-termination by adding the following transition rule, which expresses that the effect of *Later* is a silent transition τ:

$$\frac{}{Later\ p \xrightarrow{\tau} p}$$

For example, *never* gives the infinite transition sequence $never \xrightarrow{\tau} never \xrightarrow{\tau} never \xrightarrow{\tau} \cdots$. In essence, *Later* is a more restrictive variant of *Eff* that can be used to express non-terminating behaviour. With this intuition in mind, we can extend monadic bind and parallel composition to take account of non-termination by adding the following clauses:

$$Later\ p \gg\!= f = Later\ (p \gg\!= f)$$
$$Later\ p \lhd q\ = Later\ (p \parallel q)$$
$$p \rhd Later\ q\ = Later\ (p \parallel q)$$

To prove bisimilarity properties for choice trees that are defined co-recursively, such as *never*, we need a (co)-induction principle that is powerful enough to account for this. To this end, we follow the approach of Bahr and Hutton [2022] and use a step-indexed version of bisimilarity, denoted by $\cong_i$, indexed by a natural number i that counts the number of steps. While $\cong$ is defined coinductively, $\cong_i$ is defined inductively as the smallest relation such that $p \cong_0 q$, and $p \cong_{i+1} q$ if the following two conditions hold (where $j = i$ if $l = \tau$, and $j = i + 1$ otherwise):

$$\text{If } p \xrightarrow{l} p' \text{ then there is some } q \xrightarrow{l} q' \text{ with } p' \cong_j q'$$
$$\text{If } q \xrightarrow{l} q' \text{ then there is some } p \xrightarrow{l} p' \text{ with } p' \cong_j q'$$

We can show (classically, using the law of excluded middle) that $p \cong q$ iff $p \cong_i q$ for all i. In particular, this means that we can prove bisimilarity $p \cong q$ by proving $p \cong_i q$ by induction on i. To do this, we make use the following congruence law for *Later*:

$$\frac{p \cong_j q \quad \text{for all } j < i}{Later\ p\ \cong_i\ Later\ q}$$

That is, whenever we 'go under' a *Later* we may use our induction hypothesis because we decrease the step index i. For example, suppose we define a variant of *never* that invokes *Later* twice before making a recursive call by $never' = Later\ (Later\ never')$. Then we can prove that *never′* and *never* are bisimilar by proving that $never' \cong_i never$ by induction on i, which proceeds as follows:

$$never'\ \cong_i\ Later\ (Later\ never')\ \cong_i\ Later\ (Later\ never)\ \cong_i\ Later\ never\ \cong_i\ never$$

That is, we first apply the definition of *never′*, then apply the induction hypothesis nested under two occurrences of *Later*, and finally apply the definition of *never* twice.

The extended definition of choice trees carries over to the codensity construction. In particular, we have a $Later_c$ operation on codensity choice trees $CTree_c$,

$Later_c :: CTree_c\ e\ a \to CTree_c\ e\ a$
$Later_c\ p = \lambda c \to Later\ (p\ c)$

as well a step-indexed bisimilarity relation, defined by:

$$p \cong_i q \quad \text{iff} \quad p\ c \cong_i q\ c \ \text{ for all } c$$

All our previous laws for (codensity) choice trees, e.g. the congruence laws and those in Figure 2, also hold for step-indexed bisimilarity. In addition, we have the following laws for $Later_c$:

$$\frac{p \cong_j q \quad \text{for all } j < i}{Later_c\ p \ \cong_i \ Later_c\ q}\ (Later_c\text{-CONG}) \qquad \frac{}{Later_c\ p \gg\!= f \ \cong_i \ Later_c\ (p \gg\!= f)}\ (Later_c\text{-BIND})$$

Similarly to choice trees, $Later_c$-CONG provides us with a powerful induction principle for codensity choice trees as we can prove $p \cong q$ by proving $p \cong_i q$ for all i.

7.2 Concurrent effect handlers

While (codensity) choice trees have a parallel composition operator, there is no non-trivial interaction between parallel computations. In particular, there is no way for such computations to communicate with each other. To support this, we need to extend the definition of the $\bowtie$ operator from Section 2.3, which describes how two parallel computations interact. With the current definition, only very simple interaction is possible, namely that if both computations finish with a result value then their parallel composition finishes with the combined result value:

$$Now\ v \bowtie Now\ w \ = \ Now\ (v, w)$$

To allow effects of the two computations to interact, we parameterise the parallel composition operator with a type class that provides a concurrent effect handler:

class *Concurrent e* **where**
  $(\rightleftarrows) :: e\ a \to e\ b \to CTree\ e\ (a, b)$

The definition of $\bowtie$ is then generalised so that it uses this concurrent effect handler, which is achieved by adding the following clause to its definition:

$$Eff\ e_1\ c_1 \bowtie Eff\ e_2\ c_2 = (e_1 \rightleftarrows e_2) \gg\!= \lambda(x, y) \to c_1\ x \parallel c_2\ y$$

To ensure associativity of parallel composition $\parallel$, as well as the right-biased version $\vec{\shortparallel}$, we require that choice trees $e_1 \rightleftarrows e_2$ can only have transitions of the form

$$e_1 \rightleftarrows e_2 \stackrel{\tau}{\Longrightarrow} return\ v$$

which means that $e_1 \rightleftarrows e_2$ is a possibly empty sum of terms of the form $Later\ (return\ (v, w))$, i.e. two concurrent effects e_1 and e_2 are handled by a silent transition τ that provides return values v and w for the two effects. If the sum is empty, the two effects do not interact concurrently, whereas if there is more than one summand, their interaction is non-deterministic. To ensure commutativity, we also require that $\rightleftarrows$ is commutative in the following way:

$$e_1 \rightleftarrows e_2 \stackrel{\tau}{\Longrightarrow} return\ (v, w) \quad \text{implies} \quad e_2 \rightleftarrows e_1 \stackrel{\tau}{\Longrightarrow} return\ (w, v)$$

The resulting generalised $\parallel$ operator specialises to the previous version if $\rightleftarrows$ always returns $Zero$. For example, the instance declaration for the printing effect is simply:

```
instance Concurrent PrintEff where
  _ ⇄ _ = Zero
```

For a more interesting example, let us consider an effect type *CommEff* that supports communication between computations in the form of sending and receiving integer values:

```
data CommEff a where
  Send    :: Int → CommEff ()
  Receive :: CommEff Int

send :: Int → CTree CommEff ()
send n = Eff (Send n) return

receive :: CTree CommEff Int
receive = Eff Receive return
```

For the two operations to interact in the desired way, we define $\rightleftarrows$ as follows:

```
instance Concurrent CommEff where
  Send n ⇄ Receive = Later (return ((), n))
  Receive ⇄ Send n = Later (return (n, ()))
  _       ⇄ _      = Zero
```

For example, we have the following transitions:

$$send\ 1 \parallel receive \xRightarrow{\tau} return\ () \parallel return\ 1 \xRightarrow{1} Zero$$

All previously presented laws for $\parallel$ and $\parallel_c$ (see Figure 2) carry over to this extension with concurrent effect handlers. Moreover, the rules characterising $\parallel$ and $\parallel_c$ (see Section 2.3) carry over to this generalisation. However, to make the rules complete we add the following rules:

$$\frac{p \xRightarrow{\tau} p'}{p \parallel q \xRightarrow{\tau} p \parallel q} \qquad \frac{q \xRightarrow{\tau} q'}{p \parallel q \xRightarrow{\tau} p \parallel q'} \qquad \frac{p \xRightarrow{\uparrow e} c \qquad p' \xRightarrow{\uparrow e'} c' \qquad e \rightleftarrows e' \xRightarrow{\tau} return\ (v, v')}{p \parallel p' \xRightarrow{\tau} c\ v \parallel c'\ v'}$$

These account for both the addition of the *Later* constructor and concurrent effect handlers.

7.3 Effect handlers

While the generalised parallel composition *allows* communication, it does not prevent communication effects to trigger independently. For example, we also have the transition

$$send\ 1 \parallel receive \xRightarrow{\uparrow()\downarrow 2} send\ 1 \parallel return\ 2$$

in which the value 2 is received from the outside context independently of the parallel computation *send* 1. To restrict such communication to a local context, we follow Chappe et al. [2023] and use an effect handling function *interp*, defined as follows:

```
interp :: (forall b . e b → CTree f b) → CTree e a → CTree f a
interp han (Now v)   = Now v
interp han (Later p) = Later (interp han p)
interp han (p ⊕ q)   = interp han p ⊕ interp han q
interp han Zero      = Zero
interp han (Eff e c) = han e >>= λi → interp han (c i)
```

The argument *han* handles each effect from the effect type e by interpreting it as a choice tree with a potentially different effect type f. We restrict effects to a local context by simply removing all effects, which is achieved by interpreting them by the choice tree $Zero$:

$$\begin{aligned}&\mathit{restrict} :: \mathit{CTree}\ \mathit{CommEff}\ a \to \mathit{CTree}\ e\ a\\&\mathit{restrict} = \mathit{interp}\ (\lambda\_ \to \mathit{Zero})\end{aligned}$$

Note that the result type of *restrict* is polymorphic in the effect type e. In particular, we could choose the empty effect type, which witnesses the fact that there are indeed no observable effects other than τ transitions. Using restriction, we now only have a single transition from the choice tree $\mathit{restrict}\ (\mathit{send}\ 1 \mathbin{\vec{\shortparallel}} \mathit{receive})$, namely the τ transition to $\mathit{restrict}\ (\mathit{return}\ () \mathbin{\vec{\shortparallel}} \mathit{return}\ 1)$. That is, values are now prevented from being sent to and received from the outside context.

We define the corresponding variant of *interp* for codensity choice trees as follows:

$$\begin{aligned}&\mathit{interp}_{\mathrm{c}} :: (\mathit{forall}\ b\,.\,e\ b \to \mathit{CTree}\ f\ b) \to \mathit{CTree}_{\mathrm{c}}\ e\ a \to \mathit{CTree}_{\mathrm{c}}\ f\ a\\&\mathit{interp}_{\mathrm{c}}\ f\ p\ c = \mathit{interp}\ f\ (p\ \mathit{return}) \gg\!= c\end{aligned}$$

In this definition, the bind operator for choice trees is used to apply the continuation. Following the extension of other choice tree operators to codensity choice trees, we may have expected the following definition, in which the continuation is passed directly to p:

$$\mathit{interp}'_{\mathrm{c}}\ f\ p\ c = \mathit{interp}\ f\ (p\ c)$$

However, this definition suffers from two drawbacks. First of all, its type would be restricted to the case where $f = e$, which means that it would not be applicable for effect handlers that change the set of effects. For example, the type of the *restrict* function would no longer reflect the absence of observable effects. And secondly, $\mathit{interp}'_{\mathrm{c}}$ satisfies the following law:

$$\mathit{interp}'_{\mathrm{c}}\ h\ p \gg\!= f \ \cong_i\ \mathit{interp}'_{\mathrm{c}}\ h\ (p \gg\!= f)$$

This law is similar to the $\vec{\shortparallel}_{\mathrm{c}}$-bind law and could be useful for calculation. However, it also reveals that the scope of $\mathit{interp}'_{\mathrm{c}}$ extends arbitrarily to the right of a bind operator, which would make it unsuitable for defining a restriction function with a delimited scope. Instead of the above undesirable law, both *interp* and $\mathit{interp}_{\mathrm{c}}$ satisfy the following restricted version:

$$\mathit{fmap}\ f\ (\mathit{interp}\ h\ p) \ \cong_i\ \mathit{interp}\ h\ (\mathit{fmap}\ f\ p) \qquad (\textit{interp-fmap})$$

In addition, both *interp* and $\mathit{interp}_{\mathrm{c}}$ also satisfy congruence laws.

Finally, we note that effect handlers can also be generalised so that they can use an internal state, by means of the following interpretation functions:

$$\begin{aligned}&\mathit{interpSt} :: s \to (\mathit{forall}\ b\,.\,s \to e\ b \to \mathit{CTree}\ f\ (b, s)) \to \mathit{CTree}\ e\ a \to \mathit{CTree}\ f\ a\\&\mathit{interpSt}\ s\ f\ (\mathit{Now}\ v) = \mathit{Now}\ v\\&\mathit{interpSt}\ s\ f\ (\mathit{Later}\ p) = \mathit{Later}\ (\mathit{interpSt}\ s\ f\ p)\\&\mathit{interpSt}\ s\ f\ (p \oplus q) = \mathit{interpSt}\ s\ f\ p \oplus \mathit{interpSt}\ s\ f\ q\\&\mathit{interpSt}\ s\ f\ \mathit{Zero} = \mathit{Zero}\\&\mathit{interpSt}\ s\ f\ (\mathit{Eff}\ e\ c) = f\ s\ e \gg\!= (\lambda(x, s') \to \mathit{interpSt}\ s'\ f\ (c\ x))\end{aligned}$$

$$\begin{aligned}&\mathit{interpSt}_{\mathrm{c}} :: s \to (\mathit{forall}\ b\,.\,s \to e\ b \to \mathit{CTree}\ f\ (b, s)) \to \mathit{CTree}_{\mathrm{c}}\ e\ a \to \mathit{CTree}_{\mathrm{c}}\ f\ a\\&\mathit{interpSt}_{\mathrm{c}}\ s\ f\ p\ c = \mathit{interpSt}\ s\ f\ (p\ \mathit{return}) \gg\!= c\end{aligned}$$

For example, instead of using *restrict* to prevent communication with the outside context, we could simulate a context that stores sent values and returns them back when asked:

$$\begin{aligned}&\mathit{parrot} :: \mathit{Int} \to \mathit{CTree}\ \mathit{CommEff}\ a \to \mathit{CTree}\ e\ a\\&\mathit{parrot}\ s = \mathit{interpSt}\ s\ \mathit{han}\ \textbf{where}\end{aligned}$$

$$
\begin{aligned}
&\mathit{han} :: \mathit{Int} \to \mathit{CommEff}\ b \to \mathit{CTree}\ e\ (b, \mathit{Int}) \\
&\mathit{han}\ s\ (\mathit{Send}\ n) = \mathit{Later}\ (\mathit{return}\ ((), n)) \\
&\mathit{han}\ s\ \mathit{Receive}\ \ = \mathit{Later}\ (\mathit{return}\ (s, s))
\end{aligned}
$$

For example, we have the following transitions:

$$\mathit{parrot}\ 0\ (\mathit{send}\ 1 \gg \mathit{receive}) \overset{\tau}{\Longrightarrow} \mathit{parrot}\ 1\ \mathit{receive} \overset{\tau}{\Longrightarrow} \mathit{return}\ 1$$

8 CONCURRENT LAMBDA CALCULUS

To demonstrate that codensity choice trees and their associated reasoning principles scale to more sophisticated concurrent languages, we show to to calculate a compiler for an untyped lambda calculus extended with concurrency and channel-based communication.

8.1 Syntax

The syntax for our language is defined as follows, in which bound variables are represented using de Bruijn indices, and channel names are represented as integers:

$$
\begin{aligned}
\textbf{data}\ \mathit{Expr} = {} & \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr} \mid \\
& \mathit{Var}\ \mathit{Int} \mid \mathit{Abs}\ \mathit{Expr} \mid \mathit{App}\ \mathit{Expr}\ \mathit{Expr} \mid \\
& \mathit{Send}\ \mathit{Expr}\ \mathit{Expr} \mid \mathit{Receive}\ \mathit{Expr} \mid \mathit{Fork}\ \mathit{Expr}
\end{aligned}
$$

Informally, $\mathit{Var}\ i$ is the variable with de Bruijn index $i \geqslant 0$, $\mathit{Abs}\ x$ constructs an abstraction over the expression x, $\mathit{App}\ x\ y$ applies the abstraction that results from evaluation of x to the value of y, $\mathit{Send}\ c\ x$ sends the integer that results from evaluation of x on the channel c, and $\mathit{Receive}\ c$ receives an integer on the channel c. Finally, $\mathit{Fork}\ x$ spawns a new concurrent process to evaluate x, creates a new channel c that is passed to this process, and returns c as the result value.

In this lambda calculus, we can express complex concurrent programs that fork processes, create communication channels, and pass integers on those channels. For example, using a standard syntax that can readily be translated into the *Expr* type, the following program forks a new process that increments an integer n received on the newly created channel c:

$$\mathit{fork}\ (\lambda c\,.\,\textbf{let}\ n = \mathit{receive}\ c\ \textbf{in}\ \mathit{send}\ c\ (n + 1))$$

8.2 Semantics

We begin by defining an effect type *ChanEff* for channels that support sending and receiving integers and creating new channels, where channels themselves are simply integers:

$$
\begin{aligned}
&\textbf{type}\ \mathit{Chan} = \mathit{Int} \\
&\textbf{data}\ \mathit{ChanEff}\ a\ \textbf{where} \\
&\quad \mathit{SendInt}\ \ \ :: \mathit{Chan} \to \mathit{Int} \to \mathit{ChanEff}\ () \\
&\quad \mathit{ReceiveInt} :: \mathit{Chan} \to \mathit{ChanEff}\ \mathit{Int} \\
&\quad \mathit{NewChan}\ \ :: \mathit{ChanEff}\ \mathit{Chan} \\
&\mathit{send} :: \mathit{Chan} \to \mathit{Int} \to \mathit{CTree}_{\mathrm{c}}\ \mathit{ChanEff}\ () \\
&\mathit{send}\ c\ n = \mathit{Eff}_{\mathrm{c}}\ (\mathit{SendInt}\ c\ n)\ \mathit{return} \\
&\mathit{receive} :: \mathit{Chan} \to \mathit{CTree}_{\mathrm{c}}\ \mathit{ChanEff}\ \mathit{Int} \\
&\mathit{receive}\ c = \mathit{Eff}_{\mathrm{c}}\ (\mathit{ReceiveInt}\ c)\ \mathit{return} \\
&\mathit{newChan} :: \mathit{CTree}_{\mathrm{c}}\ \mathit{ChanEff}\ \mathit{Chan} \\
&\mathit{newChan} = \mathit{Eff}_{\mathrm{c}}\ \mathit{NewChan}\ \mathit{return}
\end{aligned}
$$

To use the parallel composition operator, we also provide a concurrent effect handler $\rightleftarrows$ that expresses the interaction of send and receive effects. In particular, $SendInt\ c\ n$ causes any concurrent effect $ReceiveInt\ c$ on the same channel c to evaluate to the integer n:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Concurrent}\ \mathit{ChanEff}\ \textbf{where} \\
\quad \mathit{SendInt}\ c\ n \rightleftarrows \mathit{ReceiveInt}\ c' \mid c \equiv c' = \mathit{Later}\ (\mathit{return}\ ((), n)) \\
\quad \mathit{ReceiveInt}\ c' \rightleftarrows \mathit{SendInt}\ c\ n \mid c \equiv c' = \mathit{Later}\ (\mathit{return}\ (n, ())) \\
\quad \_ \rightleftarrows \_ = \mathit{Zero}
\end{array}
$$

Using the effect type for channels, our aim now is to define the semantics of the language as a function that evaluates an open expression to a value in a given environment:

$$\mathit{eval} : \mathit{Expr} \to \mathit{Env} \to \mathit{CTree}_{\mathrm{c}}\ \mathit{ChanEff}\ \mathit{Value}$$

Because the language now has first-class functions, it no longer suffices to use integers as the value domain for the semantics, so we define a value type that also includes closures, which comprise an unevaluated expression and an environment that captures its free variables:

$$\textbf{data}\ \mathit{Value} = \mathit{Num}\ \mathit{Int} \mid \mathit{Clo}\ \mathit{Expr}\ \mathit{Env}$$

In turn, an environment can be represented simply as a list of values,

$$\textbf{type}\ \mathit{Env} = [\,\mathit{Value}\,]$$

where the value of the variable with de Bruijn index i is given by indexing into the list at position i using a lookup function that prevents further transitions if the variable is not found:

$$
\begin{array}{l}
\mathit{lookup} :: \mathit{Int} \to [\,a\,] \to \mathit{CTree}_{\mathrm{c}}\ e\ a \\
\mathit{lookup}\ \_\ [\,] \quad = \mathit{Zero}_{\mathrm{c}} \\
\mathit{lookup}\ 0\ (v : vs) = \mathit{return}\ v \\
\mathit{lookup}\ i\ (v : vs) = \mathit{lookup}\ (i - 1)\ vs
\end{array}
$$

Using these ideas, the semantics for open expressions can now be defined as follows:

$$
\begin{array}{l}
\mathit{eval} :: \mathit{Expr} \to \mathit{Env} \to \mathit{CTree}_{\mathrm{c}}\ \mathit{ChanEff}\ \mathit{Value} \\
\mathit{eval}\ (\mathit{Val}\ n) \quad e = \mathit{return}\ (\mathit{Num}\ n) \\
\mathit{eval}\ (\mathit{Add}\ x\ y) \quad e = \textbf{do}\ \mathit{Num}\ n \leftarrow \mathit{eval}\ x\ e\,;\ \mathit{Num}\ m \leftarrow \mathit{eval}\ y\ e\,;\ \mathit{return}\ (\mathit{Num}\ (n + m)) \\
\mathit{eval}\ (\mathit{Var}\ n) \quad e = \mathit{lookup}\ n\ e \\
\mathit{eval}\ (\mathit{Abs}\ x) \quad e = \mathit{return}\ (\mathit{Clo}\ x\ e) \\
\mathit{eval}\ (\mathit{App}\ x\ y) \quad e = \textbf{do}\ \mathit{Clo}\ x'\ e' \leftarrow \mathit{eval}\ x\ e\,;\ v \leftarrow \mathit{eval}\ y\ e\,;\ \mathit{Later}_{\mathrm{c}}\ (\mathit{eval}\ x'\ (v : e')) \\
\mathit{eval}\ (\mathit{Send}\ x\ y) \quad e = \textbf{do}\ \mathit{Num}\ c \leftarrow \mathit{eval}\ x\ e\,;\ \mathit{Num}\ n \leftarrow \mathit{eval}\ y\ e\,;\ \mathit{send}\ c\ n\,;\ \mathit{return}\ (\mathit{Num}\ n) \\
\mathit{eval}\ (\mathit{Receive}\ x) \quad e = \textbf{do}\ \mathit{Num}\ c \leftarrow \mathit{eval}\ x\ e\,;\ n \leftarrow \mathit{receive}\ c\,;\ \mathit{return}\ (\mathit{Num}\ n) \\
\mathit{eval}\ (\mathit{Fork}\ x) \quad e = \textbf{do}\ c \leftarrow \mathit{newChan}\,;\ \mathit{eval}\ x\ (\mathit{Num}\ c : e)\ \vec{\amalg}_{\mathrm{c}}\ \mathit{return}\ (\mathit{Num}\ c)
\end{array}
$$

There are a few points to note about the above semantics. First of all, to make the definition well-founded, we need to guard the final recursive call in the *App* case with a $Later_{\mathrm{c}}$. All other recursive calls are on structurally smaller expressions. Secondly, some cases make use of non-exhaustive pattern matching. For example, the results of the recursive calls in the *Add* case must be of the form $Num\ n$ and $Num\ m$. Here we assume that if pattern matching fails, e.g. by attempting to add closures, then the whole expression evaluates to $Zero_{\mathrm{c}}$. In Haskell, this can be achieved by implementing the *fail* method of the *MonadFail* type class for $CTree_{\mathrm{c}}$:

$$
\begin{array}{l}
\mathit{fail} :: \mathit{String} \to \mathit{CTree}_{\mathrm{c}}\ e\ a \\
\mathit{fail}\ \_ = \mathit{Zero}_{\mathrm{c}}
\end{array}
$$

And finally, the *Send*, *Receive* and *Fork* cases are defined using the operations from the effect type *ChanEff*. For example, *Fork x* creates a new channel using *newChan*, and then spawns a new concurrent process to evaluate *x*, with the with the new channel being passed to this process by adding it to the environment, and to the current process by returning it as the result.

We conclude this section by defining the top-level semantics of closed expressions. This semantics must cover three aspects: i) providing the initial environment; ii) disallowing *SendInt* effects that have not been handled by a concurrent *ReceiveInt* effect and vice versa; and iii) giving the semantics for the *NewChan* effect. The first is achieved by simply providing the empty environment, while the latter two are taken care of by a suitable stateful effect handler:

```
data NoEff a

eval' :: Expr → CTree_c NoEff Value
eval' x = interpSt_c 0 hanChan (eval x [])

hanChan :: Chan → ChanEff a → CTree None (a, Chan)
hanChan c (SendInt _ _) = Zero
hanChan c (ReceiveInt _) = Zero
hanChan c NewChan       = return (c, c + 1)
```

The effect handler *hanChan* handles the effect from *ChanEff* without using any uninterpreted effects, which is indicated by the empty effect type *NoEff* that provides no operations. *SendInt* and *ReceiveInt* effects are simply handled by *Zero*, whereas the *NewChan* effect is handled by using a state of type *Chan*, which is initialised to 0 and incremented every time the effect is used.

8.3 Compiler specification

Following the approach of Bahr and Hutton [2022], our goal to define a compiler *comp* :: *Expr* → *Code* → *Code* that produces code for a stack machine *exec*:: *Code* → *Conf* → *CTree*$_c$ *ChanEff Conf*, where *Conf* is the type of configurations for the machine. Because the source language semantics now requires an environment, the configuration type also includes an environment:

```
type Conf = (Stack, Env')
```

However, the machine may require a different form of environment to the source language, so we use a new type *Env'* for this purpose, defined as a list of machine values of type *Value'*:

```
type Env' = [Value']
```

To convert between source language and machine values, we assume a conversion function *conv* :: *Value* → *Value'*, which is lifted to environments by simply mapping over the list of values:

```
conv_E :: Env → Env'
conv_E = map conv
```

Similarly to *comp*, *Code* and *exec*, the definitions for *Value'* and *conv* are not given in advance, but will be derived during the compiler calculation. Finally, a stack is initially defined as a list of machine values, with the element type being extended as required during the calculation:

```
type Stack = [Elem]
data Elem  = VAL Value'
```

Using these definitions, compiler correctness for *comp* can be specified as follows:

$$\begin{array}{l}\textbf{do } v \leftarrow eval\ x\ e \\ \quad exec\ c\ (VAL\ (conv\ v) : s, conv_E\ e)\end{array} \quad \cong \quad exec\ (comp\ x\ c)\ (s, conv_E\ e)$$

This property has the same form as for the simple language in Section 6, except that the machine now operates on configurations comprising a stack and environment, and we need to take account of the different value and environment types used by the souce and target languages. In turn, for the top-level semantics $eval'$ of closed expressions, our aim is to define a top-level compilation function $compile :: Expr \to Code$ and a top-level execution function $execute :: Code \to Conf \to CTree_{\mathrm{c}}\ NoEff\ Conf$ that satisfy the following correctness property:

$$\begin{array}{l}\textbf{do}\ v \leftarrow eval'\ x \\ \quad return\ ([\,VAL\ (conv\ v)\,],[\,])\end{array} \quad \cong \quad execute\ (compile\ x)\ ([\,],[\,])$$

That is, compiling an expression and then executing the resulting code using an empty stack and environment should result in a stack that contains the value of the expression.

8.4 Compiler calculation

Using the fact that $p \cong q$ iff $p \cong_i q$ for all step counts i, to prove the correctness property for *comp* it suffices to prove the following by induction on the step count i and expression x:

$$\begin{array}{l}\textbf{do}\ v \leftarrow eval\ x\ e \\ \quad exec\ c\ (VAL\ (conv\ v) : s, conv_E\ e)\end{array} \quad \cong_i \quad exec\ (comp\ x\ c)\ (s, conv_E\ e)$$

For each case of x, we start on the left-side of the property and seek to transform it into the form $exec\ c'\ (s, conv_E\ e)$ for some code c', which then gives a clause $comp\ x\ c = c'$ for the compiler in this case. As previously, the calculation is driven by the desire to transform the term being manipulated into the require form using the induction hypotheses.

The calculation proceeds in a similar manner to the lambda calculus [Bahr and Hutton 2022], with the cases for the extra concurrency primitives *Send*, *Receive* and *Fork* being similar to the cases for *Print* and *Fork* in Section 6. The supplementary material for the articles includes full details of the calculations. One notable difference, however, is that the source language now has both externally observable effects and the possibility of getting stuck because of an error, such as trying to add non-numeric values or looking up the value of an unbound variable. Due to the presence of effects, it may be observable precisely when a computation gets stuck. To illustrate the consequences, consider the calculation for the *Add* case, which proceeds as follows:

$$\begin{array}{ll}
& \textbf{do}\ v \leftarrow eval\ (Add\ x\ y)\ e \\
& \quad exec\ c\ (VAL\ (conv\ v) : s, conv_E\ e)) \\
\cong_i & \quad \{\,\text{definition of } eval\,\} \\
& \textbf{do}\ v \leftarrow \textbf{do}\ \{Num\ n \leftarrow eval\ x\ e\,;\ Num\ m \leftarrow eval\ y\ e\,;\ return\ (Num\ (n+m))\,\} \\
& \quad exec\ c\ (VAL\ (conv\ v) : s, conv_E\ e) \\
\cong_i & \quad \{\,\text{monad laws}\,\} \\
& \textbf{do}\ Num\ n \leftarrow eval\ x\ e\,;\ Num\ m \leftarrow eval\ y\ e \\
& \quad exec\ c\ (VAL\ (conv\ (Num\ (n+m))) : s, conv_E\ e) \\
\cong_i & \quad \{\,\text{define: } conv\ (Num\ n) = Num'\ n\,\} \\
& \textbf{do}\ Num\ n \leftarrow eval\ x\ e\,;\ Num\ m \leftarrow eval\ y\ e \\
& \quad exec\ c\ (VAL\ (Num'\ (n+m)) : s, conv_E\ e) \\
\cong_i & \left\{\begin{array}{l}\text{define: } exec\ (ADD\ c)\ (VAL\ (Num'\ m) : VAL\ (Num'\ n) : s, e) = \\ \quad exec\ c\ (VAL\ (Num'\ (n+m)) : s, e)\end{array}\right\} \\
& \textbf{do}\ Num\ n \leftarrow eval\ x\ e\,;\ Num\ m \leftarrow eval\ y\ e \\
& \quad exec\ (ADD\ c)\ (VAL\ (Num'\ m) : VAL\ (Num'\ n) : s, conv_E\ e) \\
\cong_i & \quad \{\,\text{define: } exec\ (ADD\ c)\ _ = Zero_{\mathrm{c}}\,\}
\end{array}$$

$$
\begin{array}{ll}
& \mathbf{do}\ Num\ n \leftarrow eval\ x\ e;\ v \leftarrow eval\ y\ e \\
& \quad exec\ (ADD\ c)\ (VAL\ (conv\ v) : VAL\ (Num'\ n) : s, conv_E\ e) \\
\cong_i & \quad \{\ \text{induction hypothesis for } y\ \} \\
& \mathbf{do}\ Num\ n \leftarrow eval\ x\ e \\
& \quad exec\ (comp\ y\ (ADD\ c))\ (VAL\ (Num'\ n) : s, conv_E\ e) \\
\cong_i & \quad \left\{ \begin{array}{l} \text{define: } exec\ (ISNUM\ c)\ (VAL\ (Num'\ n) : s, e) = exec\ c\ (VAL\ (Num'\ n) : s, e) \\ \qquad\qquad exec\ (ISNUM\ c)\ \_ = Zero_{\mathrm{c}} \end{array} \right\} \\
& \mathbf{do}\ v \leftarrow eval\ x\ e \\
& \quad exec\ (ISNUM\ (comp\ y\ (ADD\ c)))\ (VAL\ (conv\ v) : s, conv_E\ e)) \\
\cong_i & \quad \{\ \text{induction hypothesis for } x\ \} \\
& exec\ (comp\ x\ (ISNUM\ (comp\ y\ (ADD\ c))))\ (s, conv_E\ e)
\end{array}
$$

In the above calculation, in addition to the *ADD* instruction that adds together two numbers on top of the stack, we introduce an *ISNUM* instruction that checks if the top of the stack is a numeric value. The introduction of this latter instruction is driven by the need to manipulate the term so that we can apply the induction hypothesis for x. Intuitively, including *ISNUM* after the compiled code for x is required because the generated code must have the same semantics as the source expression $Add\ x\ y$ and hence fail as early as possible. Otherwise, the generated code would exhibit the computational effects of the expression y even though the source expression $Add\ x\ y$ would not. Importantly, we did not have to make this observation, as the need for an instruction with the semantics of *ISNUM* falls out naturally as part of the calculation.

In turn, we can calculate the top-level function *compile* from its correctness property. In this case we don't need step-counting or induction, as a simple equational reasoning suffices:

$$
\begin{array}{ll}
& \mathbf{do}\ v \leftarrow eval'\ x;\ return\ ([\,VAL\ (conv\ v)\,], [\,]) \\
\cong & \quad \{\ \text{definition of } eval'\ \} \\
& \mathbf{do}\ v \leftarrow interpSt_{\mathrm{c}}\ 0\ hanChan\ (eval\ x\ [\,])\ ;\ return\ ([\,VAL\ (conv\ v)\,], [\,]) \\
\cong & \quad \{\ interpSt_{\mathrm{c}}\text{-}fmap \text{ law}\ \} \\
& interpSt_{\mathrm{c}}\ 0\ hanChan\ (\mathbf{do}\ v \leftarrow eval\ x\ [\,];\ return\ ([\,VAL\ (conv\ v)\,], [\,])) \\
\cong & \quad \{\ \text{define: } exec\ HALT\ (s, e) = return\ (s, e)\ \} \\
& interpSt_{\mathrm{c}}\ 0\ hanChan\ (\mathbf{do}\ v \leftarrow eval\ x\ [\,];\ exec\ HALT\ ([\,VAL\ (conv\ v)\,], [\,])) \\
\cong & \quad \{\ \text{compiler correctness for } comp\ \} \\
& interpSt_{\mathrm{c}}\ 0\ hanChan\ (exec\ (comp\ x\ HALT)\ ([\,], [\,])) \\
\cong & \quad \{\ \text{define: } execute\ c\ (s, e) = interpSt_{\mathrm{c}}\ 0\ hanChan\ (exec\ c\ (s, e))\ \} \\
& execute\ (comp\ x\ HALT)\ ([\,], [\,])
\end{array}
$$

In summary, we have calculated the definitions in Figure 4.

The configuration of the virtual machine for our concurrent lambda calculus is similar to that of the virtual machine calculated in Section 6, with the difference that the machine now also has a global state *ch* that denotes the next available channel, and each thread has an environment in addition to a stack. An execution sequence of the machine has form

$$execute\ c\ (s, e) \overset{\tau}{\Longrightarrow} ch_1 \otimes p_1 \overset{\tau}{\Longrightarrow} ch_2 \otimes p_2 \overset{\tau}{\Longrightarrow} \cdots \overset{(s,e)}{\Longrightarrow} Zero_{\mathrm{c}}$$

where $ch \otimes p$ abbreviates $interpSt_{\mathrm{c}}\ ch\ hanChan\ p$, and each p_i is bisimilar to an expression:

$$exec\ c_1\ (s_1, e_1)\ \vec{\shortparallel}_{\mathrm{c}}\ exec\ c_2\ (s_2, e_2)\ \vec{\shortparallel}_{\mathrm{c}}\ \cdots\ \vec{\shortparallel}_{\mathrm{c}}\ exec\ c_{n+1}\ (s_{n+1}, e_{n+1})$$

Note that because all effects are handled by $interpSt_{\mathrm{c}}$, the only externally observable effects are silent τ transitions. However, in a similar manner to the language in Section 6, we could have included a

Target language

```
data Code =
    PUSH Int Code | ADD Code |
  | ISNUM Code | LOOKUP Int Code
  | ABS Code Code | RET
  | ISCLO Code | APP Code
  | SEND Code | RECEIVE Code
  | FORK Code Code | HALT
```

Compiler

```
compile :: Expr → Code
compile e = comp e HALT

comp :: Expr → Code → Code
comp (Val n)     c = PUSH n c
comp (Add x y)   c = comp x (ISNUM (comp y (ADD c)))
comp (Var i)     c = LOOKUP i c
comp (Abs x)     c = ABS (comp x RET) c
comp (App x y)   c = comp x (ISCLO (comp y (APP c)))
comp (Send x y)  c = comp x (ISNUM (comp y (SEND c)))
comp (Receive x) c = comp x (RECEIVE c)
comp (Fork x)    c = FORK (comp x HALT) c
```

Virtual machine

```
type Conf = (Stack, Env′)
type Stack = [Elem]
type Env′  = [Value′]

data Elem   = VAL Value′ | CLO Code Env′
data Value′ = Num′ Int | Clo′ Code Env′

execute :: Code → Conf → CTree_c NoEff Conf
execute c (s, e) = interpSt_c 0 hanChan (exec c (s, e))

exec :: Code → Conf → CTree_c ChanEff Conf
exec (PUSH n c)     (s, e)                          = exec c (VAL (Num′ n) : s, e)
exec (ADD c)        (VAL (Num′ m) : VAL (Num′ n) : s, e) = exec c (VAL (Num′ (n + m)) : s, e)
exec (ISNUM c)      (VAL (Num′ n) : s, e)           = exec c (VAL (Num′ n) : s, e)
exec (LOOKUP n c)   (s, e)                          = do v ← lookup n e; exec c (VAL v : s, e)
exec (ABS c′ c)     (s, e)                          = exec c (VAL (Clo′ c′ e) : s, e)
exec RET            (VAL u : CLO c e′ : s, _)       = exec c (VAL u : s, e′)
exec (ISCLO c)      (VAL (Clo′ c′ e′) : s, e)       = exec c (VAL (Clo′ c′ e′) : s, e)
exec (APP c)        (VAL v : VAL (Clo′ c′ e′) : s, e) = Later_c (exec c′ (CLO c e : s, v : e′))
exec (SEND c)       (VAL (Num′ n) : VAL (Num′ ch) : s, e) = do send ch n; exec c (VAL (Num′ n) : s, e)
exec (RECEIVE c)    (VAL (Num′ ch) : s, e)          = do n ← receive ch; exec c (VAL (Num′ n) : s, e)
exec (FORK c′ c)    (s, e)                          = do ch ← newChan
                                                         exec c′ ([ ], Num′ ch : e) ⫴_c
                                                           exec c (VAL (Num′ ch) : s, e)
exec HALT           (s, e)                          = return (s, e)
exec _              _                               = Zero_c
```

Fig. 4. Compiler and virtual machine for the concurrent lambda calculus.

Print primitive in the lambda calculus, which the effect handler would have left uninterpreted and which therefore would appear as *PrintInt* transitions in addition to the τ transitions.

9 RELATED WORK

The codensity construction is a common trick in the functional programming literature, which is typically used to improve efficiency; for example, see Hinze [2012] for an overview. Curiously, the form of codensity choice tree without $Later_c$ that was presented in Section 4 is very close to

the definition of an efficient parallel parser by Claessen [2004]. However, because it implements a parser Claessen's definition only supports one effect, namely reading a symbol, and to improve efficiency the *Now* and $\oplus$ constructors are fused together.

As noted in Section 4, the semantics of Haskell's *forkIO* primitive [Jones et al. 1996] provided the initial inspiration for our continuation-passing style semantics, which in turn motivated the use of the codensity construction. In the remainder of this section, we review related work on compiler verification, and compare the original notion of choice trees with our version.

9.1 Compiler verification

Wand [1982a] pioneered a compiler verification technique that uses a suitably expressive lambda calculus as a common language in which to define both the source and target language of the compiler. This idea has its origins in Reynolds' seminal work on definitional interpreters [Reynolds 1972], and has proved fruitful for deriving correct-by-construction compilers [Ager et al. 2003a,b; Bahr and Hutton 2015, 2020; Gibbons 2021; Wand 1982a,b]. While this line of work is limited to sequential programming languages, Wand [1995] later demonstrated how to prove compiler correctness for concurrent languages with the help of a higher-order process calculus called HOCC, which extends the lambda calculus with primitives for spawning parallel processes as well as sending and receiving messages. However, we are not aware of any work that derives correct-by-construction compilers for concurrent languages using this technique or others.

Considerable progress has been made in subsequent work on compiler verification, leveraging the use of proof assistants to verify compilers for realistic languages, culminating in the landmark CompCert C compiler [Leroy 2006, 2009]. This work inspired many further projects on compiler verification [Patterson and Ahmed 2019], such as Chlipala's [2010] verified compiler, CakeML [Kumar et al. 2014] and DeepSpec [2023]. CompCert and its correctness proof have since been extended to support concurrency [Ševčík et al. 2013]. Rather than an intermediate language like HOCC or choice trees, Ševčík et al. specify the semantics of source and target languages directly using a small-step semantics. A key challenge for the verification of realistic compilers for concurrent languages is devising a suitable memory model [Kang et al. 2017].

9.2 Choice trees

Choice trees [Chappe et al. 2023] offer an alternative language to serve as the common semantic domain for reasoning, and Chappe et al. have demonstrated how this language can be used to model concurrent languages. Unlike Wand's higher-order process calculus HOCC, choice trees don't explicitly feature parallelism or higher-order constructs, but these features can be encoded [Chappe et al. 2023; Danielsson 2012; Xia et al. 2019]. Indeed, the concurrent lambda calculus from Section 8 is very similar to HOCC and the latter can be given a semantics in terms of (codensity) choice trees. The only notable differences are that HOCC also allows sending and receiving closed lambda terms, and that communication is via thread identifiers rather than channel identifiers.

The idea to use the effect handler operation *interpSt* to model the restriction of effects to a local context, and the parallelism operator $\|$ for choice trees, are both due to [Chappe et al. 2023]. Our addition of concurrent effect handlers is a natural generalisation. However, our syntax and semantics for choice trees is different from Chappe et al.'s in crucial ways that enable the equational reasoning that underlies our methodology. We discuss these differences in turn below.

Syntax. First of all, there are two superficial syntactic differences: instead of a binary choice operator $\oplus$ with a unit $Zero$, Chappe et al. [2023] use a choice operator *brS* of arbitrary finite arity, and instead of *Later*, they have an operator *brD* that is the composition of *brS* and *Later*. However,

brS and *brD* are interdefinable with $\oplus$, *Zero* and *Later*. Adjusting for these differences and using our notation, Chappe et al.'s definition of choice trees is equivalent to the following:

```
codata CTree′ e a where
  Now  :: a → CTree′ e a
  Later :: CTree′ e a → CTree′ e a
  (⊕)  :: CTree′ e a → CTree′ e a → CTree′ e a
  Zero :: CTree′ e a
  Eff  :: e b → (b → CTree′ e a) → CTree′ e a
```

This definition looks almost identical to ours, with one key difference: instead of an inductive definition with a nested coinductive definition for *Later*, the entire type is defined coinductively. As a result, a *CTree′* might be infinite even though it contains no *Later*, which is different from *CTree*, where all infinite behaviour must be due to *Later*. This means that while on the surface *CTree′* only permits finite non-determinism, it can in fact encode infinite non-deterministic choice:

```
infChoice :: (Int → CTree′ e a) → CTree′ e a
infChoice ps = ps 0 ⊕ infChoice (λn → ps (n + 1))
```

This definition doesn't work for *CTree* because $\oplus$ is an inductive constructor for *CTree* and hence *infChoice* would not be well-founded. Having infinite non-deterministic choice for *CTree′* makes the notion of step-indexed bisimilarity $\cong_i$ that underpins our methodology unsound, in the sense that $p \cong_i q$ for all i no longer implies $p \cong q$. For example, consider the choice trees $p, q :: \mathit{CTree'}\ e\ a$ defined by $p = \mathit{Later}\ p$ and $q = \mathit{infChoice}\ qs$, where $qs\ 0 = \mathit{Zero}$ and $qs\ n = \mathit{Later}\ (qs\ (n-1))$. Then we have $p \oplus q \cong_i q$ for all i, but not $p \oplus q \cong q$. Failure of the latter can be seen by the fact that $p \oplus q$ has non-terminating behaviour whereas q does not, while the former can be proved by first showing that $p \cong_i qs\ i$ by induction on i and then using this result to show $p \oplus q \cong_i q$.

Semantics. As we have observed in Section 2, every effectful transition is always immediately followed by an input transition that consumes the resulting value:

$$\mathit{Eff}\ o\ c \overset{\uparrow o}{\Longrightarrow} c \overset{\downarrow i}{\Longrightarrow} c\ i$$

In contrast, Chappe et al. [2023] combine the two transitions into one:

$$\mathit{Eff}\ o\ c \overset{\uparrow o \downarrow i}{\Longrightarrow} c\ i$$

This has the benefit of avoiding the need for two kinds of states in the semantics, namely choice trees and continuations. However, the resulting notion of bisimilarity is too coarse, which in turn causes the congruence property for *interp* to fail. For example, suppose we have an effect that flips a coin, and returns the result as a value of type *Bool*:

```
data CoinEff b where
  Flip :: CoinEff Bool
```

Then with the semantics of Chappe et al. the following two choice trees are bisimilar

```
ignore = Eff Flip (λb → return True) ⊕ Eff Flip (λb → return False)
negate = Eff Flip (λb → return b) ⊕ Eff Flip (λb → return (¬ b))
```

Both examples start with a non-deterministic choice, and each choice component starts with a coin flip. However, the two components of *ignore* both ignore the result of the coin flip and return the fixed results *True* and *False*, respectively. On the other hand, the two components of *negate* return the result of the coin flip unchanged and negated, respectively. The two examples have

diffferent observable behaviours and hence should not be considered bisimilar, and indeed they are not bisimilar with the semantics for choice trees that keeps $\uparrow o$ and $\downarrow i$ transitions separate.

However, with the transition semantics in the combined style of Chappe et al. [2023], the two examples are bisimilar because they have the same four possible transitions:

$$\mathit{ignore} \overset{\uparrow \mathit{Flip} \downarrow b}{\Longrightarrow} \mathit{return}\ b' \quad \text{and} \quad \mathit{negate} \overset{\uparrow \mathit{Flip} \downarrow b}{\Longrightarrow} \mathit{return}\ b' \qquad \text{for all } b, b' :: \mathit{Bool}$$

As a result of this coarser notion of bisimilarity, the effect handler operations *interp* and *interpSt* do not satisfy the congruence property, which does hold for our notion of bisimilarity. Instead, Chappe et al. [2023] prove congruence for *interp* and *interpSt* only for effect handlers *han* with $\mathit{han}\ e \cong \mathit{return}\ v$ or $\mathit{han}\ e \cong \mathit{Eff}\ f\ \mathit{return}$, which excludes the effect handler we used in Section 8. Indeed, if we define $\mathit{han\ Flip} = \mathit{Later}\ (\mathit{Eff\ Flip\ return})$, then congruence fails using Chappe et al.'s semantics, because $\mathit{interp\ han\ ignore} \ncong \mathit{interp\ han\ negate}$ even though $\mathit{ignore} \cong \mathit{negate}$.

10 CONCLUSION AND FURTHER WORK

In recent years, choice trees have proved to be a flexible, expressive and modular approach to mechanising programming language meta-theory [Chappe et al. 2023; Hur et al. 2020; Xia et al. 2019; Yoon et al. 2022]. In this article, we showed how the notion of choice trees can also be adapted to support an equational reasoning style that is key to deriving correct-by-construction compilers for concurrent languages. In particular, in combination with subtle changes in the syntax and semantics of choice trees, the use of a codensity construction allows the semantics of concurrent languages to be concisely captured in a monadic style, and supports a high-level, algebraic approach to transforming the resulting semantics into compilers.

In terms of further work, the use of (codensity) choice trees with explicit effect types opens up the opportunity for deriving multi-stage compilers, where each stage compiles away some effects and leaves others untouched. The concurrent lambda calculus compiler derived in Section 8 provides an example of this, where the print effect remains uninterpreted in the type $\mathit{CTree}_{\mathrm{c}}\ \mathit{PrintEff}$ of the codensity choice tree that is used for both the *Expr* and *Code* languages. Thus one could see this compiler as the first stage of a multi-stage compiler. The semantics of *Code* could be refined by an effect handler that handles the print effect so that a second compiler calculation could produce the second stage compiler that translates *Code* to a lower-level language.

REFERENCES

Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. 2003a. A Functional Correspondence Between Evaluators and Abstract Machines. In *Proceedings of the International Conference on Principles and Practice of Declarative Programming*.

Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. 2003b. *From Interpreter to Compiler and Virtual Machine: A Functional Derivation.* Technical Report RS-03-14. BRICS, Department of Computer Science, University of Aarhus.

Patrick Bahr and Graham Hutton. 2015. Calculating Correct Compilers. *Journal of Functional Programming* 25 (2015).

Patrick Bahr and Graham Hutton. 2020. Calculating Correct Compilers II: Return of the Register Machines. *Journal of Functional Programming* 30 (2020).

Patrick Bahr and Graham Hutton. 2022. Monadic Compiler Calculation. *Proceedings of the ACM on Programming Languages* 6, ICFP (2022).

Nicolas Chappe, Paul He, Ludovic Henrio, Yannick Zakowski, and Steve Zdancewic. 2023. Choice Trees: Representing Nondeterministic, Recursive, and Impure Programs in Coq. *Proceedings of the ACM on Programming Languages* 4, POPL (2023).

Adam Chlipala. 2010. A Verified Compiler for an Impure Functional Language. In *Proceedings of the Symposium on Principles of Programming Languages*.

Koen Claessen. 2004. Functional Pearl: Parallel Parsing Processes. *Journal of Functional Programming* 14, 6 (2004).

Nils Anders Danielsson. 2012. Operational Semantics Using the Partiality Monad. In *Proceedings of the International Conference on Functional Programming*.

DeepSpec 2023. The Science of Deep Specification. (2023). https://deepspec.org/.

Jeremy Gibbons. 2021. Continuation-Passing Style, Defunctionalization, Accumulations, and Associativity. *The Art, Science, and Engineering of Programming* 6, 2 (2021).

Ralf Hinze. 2012. Kan Extensions for Program Optimisation Or: Art and Dan Explain an Old Trick. In *Proceedings of the International Conference on Mathematics of Program Construction*.

Chung-kil Hur, Paul He, Yannick Zakowski, and Steve Zdancewic. 2020. An Equational Theory for Weak Bisimulation via Generalized Parameterized Coinduction. In *Proceedings of the International Conference on Certified Programs and Proofs*.

Simon Peyton Jones, Andrew Gorden, and Sigbjorn Finne. 1996. Concurrent Haskell. In *Proceedings of the Symposium on Principles of Programming Languages*.

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-Memory Concurrency. In *Proceedings of the Symposium on Principles of Programming Languages*.

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Proceedings of the Symposium on Principles of Programming Languages*.

Xavier Leroy. 2006. Formal Certification of a Compiler Back-End or: Programming a Compiler with a Proof Assistant. In *Proceedings of the Symposium on Principles of Programming Languages*.

Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (2009).

Daniel Patterson and Amal Ahmed. 2019. The Next 700 Compiler Correctness Theorems. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019).

John C Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM Annual Conference*.

Janis Voigtländer. 2008. Asymptotic Improvement of Computations over Free Monads. In *Proceedings of the International Conference on Mathematics of Program Construction*.

Mitchell Wand. 1982a. Deriving Target Code as a Representation of Continuation Semantics. *Transactions on Programming Languages and Systems* 4, 3 (1982).

Mitchell Wand. 1982b. Semantics-Directed Machine Architecture. In *Proceedings of the Symposium on Principles of Programming Languages*. Albuquerque, New Mexico.

Mitchell Wand. 1995. Compiler Correctness for Parallel Languages. In *Proceedings of International Conference on Functional Programming Languages and Computer Architecture*.

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2019. Interaction Trees: Representing Recursive and Impure Programs in Coq. *Proceedings of the ACM on Programming Languages* 4, POPL (2019).

Irene Yoon, Yannick Zakowski, and Steve Zdancewic. 2022. Formal reasoning about layered monadic interpreters. *Proceedings of the ACM on Programming Languages* 6, ICFP (2022).

Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM* 60, 3 (2013).