A Calculus for Esterel

“If can, can. If no can, no can.” —Hawaiian pidgin proverb

SPENCER P. FLORENCE, Northwestern University
SHU-HUNG YOU, Northwestern University
JESSE A. TOV, Northwestern University
ROBERT BRUCE FINDLER, Northwestern University

The language Esterel has found success in many safety-critical applications, such as fly-by-wire systems and nuclear power plant control software. Its imperative style is natural to programmers building such systems and its precise semantics makes it work well for reasoning about programs.

Existing semantics of Esterel generally fall into two categories: translation to Boolean circuits, or operational semantics that give a procedure for running a whole program. In contrast, equational theories enable reasoning about program behavior via equational rewrites at the source level. Such theories form the basis for proofs of transformations inside compilers or for program refactorings, and defining program evaluation syntactically.

This paper presents the first such equational calculus for Esterel. It also illustrates the calculus’s usefulness with a series of example equivalences and discuss how it enabled us to find bugs in Esterel implementations.

ACM Reference Format:

Spencer P. Florence, Shu-Hung You, Jesse A. Tov, and Robert Bruce Findler. 2018. A Calculus for Esterel: “If can, can. If no can, no can.” —Hawaiian pidgin proverb. 1, 1 (October 2018), 29 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The language Esterel has found success in many safety-critical applications. It has been used in the creation and verification of the maintenance and test computer, landing gear control computer, and virtual display systems of civilian and military aircraft at Dassault Aviation (Berry et al. 2000); the control software of the N4 nuclear power plants; the Airbus A320 fly-by-wire system; and the specification of part of Texas Instrument’s digital signal processors (Benveniste et al. 2002).

This success with real time and embedded systems in domains that need strong guarantees can be partially attributed to its computational model. Esterel treats computation as a series of deterministic reactions to external stimuli. All parts of a reaction complete in a single, discrete time step, called an *instant*. Furthermore, in this synchronous reactive paradigm (Benveniste and Berry 1991; Benveniste et al. 2002), each instant is isolated from interference by the outside environment once the reaction begins. In addition, instants exhibit deterministic concurrency; each reaction may contain concurrent threads without execution order effecting the result of the computation.

Authors’ addresses: Spencer P. Florence, Northwestern University, spencer.florence@eecs.northwestern.edu; Shu-Hung You, Northwestern University, shu{-}hung.you@eecs.northwestern.edu; Jesse A. Tov, Northwestern University, jesse@eecs.northwestern.edu; Robert Bruce Findler, Northwestern University, robby@eecs.northwestern.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2018 Association for Computing Machinery.
XXXX-XXXX/2018/10-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

This combination of synchronous reactions with deterministic concurrency makes formulating the semantics a challenging problem. Existing semantics tend to take two forms. The first, and most widely used, are semantics that give meaning to programs through a translation to circuits. These semantics are excellent for compilation and optimization. However they are not ideal for programmers, who would rather reason in terms of the source-level program not its compiled form.

The second form are operational semantics that eschew term rewriting in favor of decorating terms with various flavors of code pointers and state annotations to track execution. These semantics are easier for programmers to reason with but give meaning only to whole programs. They do not lend themselves to compositional reasoning about program fragments, which programmers need.

To obtain the best of both of these approaches, we build on Plotkin (1975) and Felleisen and Hieb (1992)'s work on equational theories of programming languages. These theories model languages with a set of axioms that specify when source-level terms are equivalent. As a result, they provide a single framework for both reasoning about how a program will run (e.g. reduce to an answer) using only the source text of the program, and for justifying program transformations in host of applications: compiler transformations, refactorings, program derivations, etc.

This paper reports on the first equational theory of Kernel Esterel (Berry 2002). Developing such a theory is tricky because of the highly non-local nature of evaluation in Esterel. To maintain determinism and synchrony, evaluation in one thread of execution may affect code arbitrarily far away away in the program, even if that evaluation does not directly modify shared state. For instance, the selection of a particular branch of execution in one thread may immediately unblock a different thread of execution. The selection of the other branch may render the entire program invalid. These non-local execution and correctness issues are at the heart of Esterel's notions of *Logical Correctness* and *Constructiveness*, and have informed the choice of techniques used for previous semantics. The circuit semantics match both notions well because they are intimately tied to whether or not a given cyclic circuit settles. The operational semantics handle these properties by performing full program passes on each evaluation step to both propagate execution information to the entire program, and determine which locations in the program are safe to evaluate. A calculus, however, cannot use either of those techniques. To this end our calculus borrows from Felleisen and Hieb (1992)'s equational theory of state and Potop-Butucaru (2002)'s Constructive Operational Semantics to give the first calculus for Esterel.

The remainder of this paper consists of seven sections. Section 2 provides an introduction to Esterel and to the specific syntax we use for Kernel Esterel. Section 3 explains the semantics and our central results, which have all been checked in Agda. With the semantics defined, the paper moves on to discuss implications of specific aspects of our semantics. Section 4 discusses constructiveness and how it interacts with our semantics. Section 5 gives some example equivalences that our calculus supports and discusses others that it does not. Our semantics is executable and section 6 discusses how we test that our semantics is faithful to preexisting semantics and implementations. In short, we designed and implemented an executable version of our semantics and used it to find bugs in Esterel implementations. We also automatically typeset the figures in the paper from the semantics and use it to test all of the examples in the paper. Section 7 discusses a standard reduction that we designed to aid in testing but have not proven, and we conclude with a discussion of related work in section 8.

2 A SENSE OF ESTEREL

This section provides some background on Esterel both to introduce the language to those not familiar with it and to orient Esterel experts with the particular notation we have chosen for Kernel Esterel. Figure 1 shows the syntax we use for our Esterel calculus.

$$\begin{aligned}
\mathbf{p}, \mathbf{q} ::= {} & (\texttt{signal}\ \mathbf{S}\ \mathbf{p}) \mid (\texttt{seq}\ \mathbf{p}\ \mathbf{q}) \mid (\texttt{emit}\ \mathbf{S}) \mid (\texttt{present}\ \mathbf{S}\ \mathbf{p}\ \mathbf{q}) \mid (\texttt{par}\ \mathbf{p}\ \mathbf{q}) \\
& \mid \texttt{nothing} \mid \texttt{pause} \mid (\texttt{loop}\ \mathbf{p}) \mid (\texttt{suspend}\ \mathbf{p}\ \mathbf{S}) \mid (\texttt{trap}\ \mathbf{p}) \mid (\texttt{exit}\ n) \\
& \mid (\texttt{shared}\ \mathbf{s} \coloneqq \mathbf{e}\ \mathbf{p}) \mid (\texttt{+=}\ \mathbf{s}\ \mathbf{e}) \mid (\texttt{var}\ \mathbf{x} \coloneqq \mathbf{e}\ \mathbf{p}) \mid (\coloneqq\ \mathbf{x}\ \mathbf{e}) \mid (\texttt{if}\ \mathbf{x}\ \mathbf{p}\ \mathbf{q})
\end{aligned}$$

$$\mathbf{S} \in \text{signal variables} \qquad \mathbf{x} \in \text{sequential variables}$$
$$\mathbf{s} \in \text{shared variables} \qquad \mathbf{e} \in \text{host expressions}$$

Fig. 1. Esterel Syntax

Evaluation of an Esterel program is unlike conventional programming languages in that it proceeds in a series of *instants*, each of which is intended to happen in, essentially, no time. Once an instant has completed, the Esterel program waits for the state of the outside world to change, which triggers another instant of computation. The external state changes of the world are reflected in Esterel via *signals*. Each signal can either be present (set), absent (not set), or in an indeterminate state, where we do not know yet if it will be present or absent. Once a signal's value becomes known (in a specific instant), it cannot change. Accordingly, the outside world may set a signal to present or it may not, indicating that its value is as yet undetermined (as the program itself may set it).[1] Once an instant completes, the Esterel program will have emitted some set of signals, which the outside world can observe and react to (by setting different input signals), in order to provoke different behavior in the next instant when the Esterel program runs.

Esterel is typically used as an embedded language, where the outside world is some other programming language, *e.g.*, C for reactive, real-time systems (Potop-Butucaru et al. 2007), Bigloo (Serrano and Weis 1995) and JavaScript for GUIs (Berry et al. 2011), or Racket (Flatt and PLT 2010) for medical prescriptions (Florence et al. 2015). The external language generally controls when instants take place and sets up the signal environment for each instant.

2.1 Conditioning on signals: `present`

Esterel programs can also have local signals that they use to communicate internally. Let us consider a few example programs that use internal signals to get a sense of how Esterel programs evaluate. Figure 2 shows a first example.

```
(signal SL
  (seq (emit SL)
       (present SL
                (emit SO1)
                (emit SO2))))
```

Fig. 2: A First Example

This program has no input signals and two output signals, SO1 and SO2; we prefix all signal names with an S. The `signal` form is a binding form that introduces a local signal (here named SL) available in its body. Signals that are free in the entire program are the ones that support communication with the host language, external to Esterel.

```
(signal SL
  (par (emit SL)
       (present SL
                (emit SO1)
                (emit SO2))))
```

Fig. 3: This time with `par`

The `seq` form is simply sequential composition, so the first thing this program does in the body of the `signal` form is `emit` SL, which means that the signal SL is now known to be present for this entire instant. Next, the program evaluates a signal conditional, written using the `present` keyword in Esterel. When a signal is known to be present, a `present` form is equivalent to its first subexpression, in this case (`emit` SO1). So this program `emit`s SO1 and then terminates, ending the instant with SO1 set and SO2 not set.

[1]For those familiar with Esterel: free signals in programs in our calculus correspond to input-output signals in Esterel.

Esterel also supports a deterministic form of parallelism and indeed if we replace the sequential composition in figure 2 with parallel composition, as shown in figure 3, the program is guaranteed to behave identically. Specifically, the `present` form in the second arm of the `par` (conceptually) blocks until the signal `SL` is emitted or we learn it cannot be emitted (in this instant). So the first arm of the `par` is the only part of the program that can progress, and once it performs the `(emit SL)`, that unblocks `present` form, enabling `(emit SO1)` to happen.

In order for a `present` expression to become unblocked and evaluate the second sub-expression, the Esterel program must know that there is no way that the given signal can be emitted (in this instant). One way this can happen is that there are no occurrences of `(emit SL)`. So, if we remove the `(emit SL)` from our running example, as shown in figure 4, then the program will `emit` the signal `SO2`, as the `present` form reduces to its second branch since there is no way to emit `SL`.

```
(signal SL
  (present SL
           (emit SO1)
           (emit SO2)))
```

Fig. 4: A signal never emitted

The way that `present` works helps guarantee Esterel's form of deterministic concurrency. Until a particular signal's value becomes known, the program simply refuses to make a choice about which branch to run. This style of conditional raises many interesting questions about how apparent cyclic references interact with each other, however. For example, what should the program in figure 5 do? (`nothing` is the Esterel equivalent of `unit` or `void` in other languages.) How such programs behave is well-studied in the Esterel community and touches on the notions of logical correctness and constructiveness, which we return to in section 2.4.

```
(signal SL1
  (signal SL2
    (par (present SL1 (emit SL2) nothing)
         (present SL2 (emit SL1) nothing))))
```

Fig. 5: Cyclic signal dependencies

2.2 Running for multiple instants: `pause`

So far, all of the example programs have terminated in a single instant but, in general, an Esterel program might run to some intermediate state and then `pause`. When all of the parallel branches of some program have paused or terminated, then the instant terminates. During the next instant, however, evaluation picks up right where it left off, with whatever remains of the program.

```
(signal SL
  (par (seq pause
            (emit SL))
       (present SL
                (emit SO1)
                (emit SO2))))
```

Fig. 6: Multiple instants

As an example, consider the program in figure 6. As it starts, the second arm of the `par` blocks, as with the example in figure 3. The first arm of the `par` first evaluates `pause`, which means that that arm of the `par` has terminated for the instant and thus the `(emit SL)` is not going to happen in this instant. Accordingly the `present` can take the else branch, safe in the knowledge that no `(emit SL)` can happen this instant.

In the next instant, the only thing that remains is the `(emit SL)`, so only `SL` is emitted.

2.3 Determining That a Signal Cannot be Emitted: `Can`

Determining whether or not a signal can be emitted is not simply a matter of eliminating untaken branches in `present` expressions and then checking the remaining `emit` expressions. Sometimes, a `present` expression may be blocked on some as-yet indeterminate signal, but portions of its branches are known to not be able to run, which enables us to declare that some signal will not be emitted.

For example, consider the program in figure 7. The `par` expression's first sub-expression is a `present` expression and its second sub-expression is a `seq` expression. The `present` expression is blocked on `SL1`. Of course, the last expression in the `seq` expression emits `SL1` but beware: it is preceded by another `present` expression that may or may not pause. If it does pause, then the `(emit SL1)` happens in a future instant (so we take the "else" branch of the `present` on `SL1`). If it does not pause, then the `(emit SL1)` happens in the current instant (and so we take the "then" branch of the `present` on `SL1`).

```
(signal SL1
  (signal SL2
    (signal SL3
      (par (present SL1
                    (present SL2
                             (emit SO1)
                             (emit SL3))
                    (present SL2
                             (emit SO2)
                             (emit SL3)))
           (seq
            (emit SL2)
            (seq
             (present SL3 pause nothing)
             (emit SL1))))))))
```

Fig. 7: Can

This complex interplay of signals and branches of `par` expressions is completely fine in Esterel. Let us work through how Esterel evaluates this program.

The first `emit` that happens is `(emit SL2)`. Once that happens, we know how the inner `present` expressions will go, even though they cannot yet fire because we do not yet know about `SL1`. In particular, we know that neither one takes their second sub-expression and thus none of the `(emit SL3)` expressions will be evaluated. Accordingly we now know that the `(present SL3 pause nothing)` in the other side of the `par` expression reduces to `nothing` and we can evaluate `(emit SL1)` which unblocks the `present` on `SL1`, which triggers the `emit` on the output signal `SO1`.

The most important step in this sequence was when Esterel decided that `SL3` cannot be emitted. The decision procedure for determining when a signal cannot be emitted is called `Can`. It follows the same reasoning we have here, but accounts for other details of the core language of Esterel. For example, it reasons about the first sub-expression of `seq` expressions in order to determine if they might pause, in order to reason about `emit` expressions that follow.

The full definition is given in figure 15 and is explained in section 3.2.

2.4 Getting stuck: Logical Correctness and Constructivity

The style of instantaneous decision making in Esterel facilitated via the `Can` function leads to programs that have no meaning, even though a traditional programming language would given them meaning. Such programs are called *logically incorrect.*

```
(signal S1
  (present S1 nothing (emit S1)))
```

Fig. 8: No possible value for S1

```
(signal S1
  (present S1 (emit S1) nothing))
```

Fig. 9: Too many values for S1

Logical correctness can be thought of as a consequence of the instantaneous nature of decision making in Esterel: if decisions about the value of a signal are communicated instantly and that value cannot change, then the program should behave as if that value was determined at the start of the instant. Therefore, there should only be one possible value for each signal. Some programs, however, have zero or multiple possible assignments. Consider the program in figure 8. No matter the definition of `Can`, `S1` cannot be set to either present or absent. If `S1` were present, the program would take the first branch of the condition, and the program would terminate without having emitted `S1`. If `S1` were set to absent, the program would chose the second branch and emitting the signal. Both of these executions lead to a contradiction, therefore there are no valid assignments of signals in this program. This program is logically incorrect.

The opposite is true for the program in figure 9. Here, if S1 is chosen to be present, the conditional will take the first branch and S1 will be emitted, justifying the choice of signal value. However, if the signal is chosen to be absent, the signal will not be emitted and the choice of absence is also justified. Thus there are two possible assignments to the signals in this program, and this program too is rejected as logically incorrect.

A related notion, *constructiveness*, arises from the order of execution imposed by `seq` and `present`. All decisions in the first part of a `seq` must be made before decisions in the second part and the value of a signal being conditioned on by `present` must be determined before decisions within either branch of the `present` can be made. Decisions that may affect sibling branches in a `par` expression, however, may happen in any order.

```
(signal S1
  (present S1 (emit S1) (emit S1)))
(signal S1
  (seq (present S1
                nothing
                nothing)
       (emit S1)))
```

Fig. 10: Constructiveness examples

To ensure these ordering constraints, Esterel imposes an order on information propagation: decisions about the value of a signal can only be used by the portion of the program that is after (in the sense of the ordering imposed by `seq` and `present`) it is emitted. Thus, programs that are logically correct may still be rejected because there is no order in which to run the program that will arrive at the single, valid assignment. Such programs are called non-constructive.[2] Accordingly, not all logically correct programs are constructive, but the converse is true: all constructive programs are logically correct. Put another way, making a guess about the value of a signal and backtracking if the guess turns out to be wrong is a match for logical correctness, but would admit programs that are non-constructive.

Succinctly, a program is constructive if it is logically correct, and the values of signals can be determined without any speculation: a signal is present only after it has been emitted, and a signal is absent only after `Can` determines it cannot be emitted without speculating about the value of other signals.

```
(signal SL1
  (signal SL2
    (par (present SL1 (emit SL2) nothing)
         (seq (present SL2 pause nothing)
              (emit SL1)))))
```

Fig. 11: Getting stuck

Example non-constructive programs are shown in figure 10. The first program has only one possible assignment for S1, as it is emitted by both branches of the conditional. Because `present` requires that the value of S1 be known before executing a sub-expression, however, there is no valid order in which to execute the code, and the program is rejected as non-constructive. A similar phenomena can be seen in the second program in figure 10, but with `seq`.

The two ordering constraints can interact in complex ways. In the example in figure 11, the `(emit SL1)` is in a `seq` that may or may not `pause`, which prevents us from determining if SL2 is emitted.

Non-constructive programs are handled two different ways by Esterel implementations. Some approximate constructiveness with a conservative static analysis and reject programs they cannot prove constructive on all inputs. This is the default behavior of Esterel v5 (Berry 2000). Others treat non-constructivity as as runtime error, raising an error if, during an instant, the program cannot determine a value for all signals. This is the behavior of HipHop.js (Berry et al. 2011), and Esterel v5 when used with the `-I` flag.

[2]The use of the name “constructive” arises from connections to constructive logic (Mendler et al. 2012).

In the circuit semantics for Esterel, a non-constructive programs is one that, when compiled to a circuit, will cause the circuit to misbehave, never settling because of cyclic dependencies between inputs and outputs of some of the gates. That is, a program is constructive if and only if its circuit stabilizes within some fixed delay (Berry 2002; Mendler et al. 2012).

Non-constructive programs usually get stuck in our calculus, but they do not always. The issues here are subtle and revisited in section 4.

2.5 Loops, `suspend`, non-local exits, variables, and the host language

Our calculus also covers the rest of Kernel Esterel. The (`trap` **p**) and (`exit` n) forms allow non-local control. Roughly speaking, (`exit` n) will abort execution up the the n+1th enclosing (`trap` **p**), reducing it to `nothing`. These can be used for exception handling, but also for non-exceptional control flow. For example, it may be simpler to express some repeating task on the assumption it never terminates and then, in parallel to it, use `exit` to abort it (with a `trap` that is outside both). Kernel Esterel's trap is a simplified form of Esterel's trap where traps are named and escaping uses the names.

The `loop` form is an infinite loop, running its body, **p**, over and over, but with a constraint that the loop's body can be started at most once in any instant. This means that the body of a loop must either pause or exit at least once in every iteration, thereby ensuring that instants always terminate. One subtle ramification of this point is that two separate iterations of a loop may run within a single instant, but only in the situation where we finish an iteration that was started in a previous instant and then start a new one in the current instant (which must then pause or exit). We return to this point in section 3.3.

Loops that fail this condition are called *instantaneous* and programs with such loops are not constructive. In our calculus, we handle this by reducing a loop in such a way that the program gets stuck if the loop were to be instantaneous.

The `suspend` form has a subtle semantics. If we are starting a `suspend` for the first time, it simply runs the body. But, if we are picking up from a previous instant where we `pause`d in the body of a `suspend`, then we test the signal. If it is present, the entire `suspend` is paused until the next instant. If it is not present, evaluation continues within the `suspend`, picking up at the `pause`.

The `suspend` form is used to implement many useful, high-level behaviors. One straight-forward use is to implement a form of multiplexing, where some portion of the input signals are used directly by several different sub-pieces of the computation at once, and another portion of the input determines which of those computation is the desired output. For example, an ALU might, in parallel, both add and multiply its inputs and store the output in the same place. The `suspend` form can be used to control whether the addition or multiplication computation happens.

Another use of `suspend` is in task management. As a workflow is progressing there may be a task that runs at some interval, but where the interval may change over time. This repeating task is important, but there may be an occasional situation where some much more important task takes precedence. So, we wish to pause the subcomputation corresponding to the repeating task with the intention of resuming it with its current state, but at a later moment in time. This pattern is captured easily with `suspend`.

And finally, Esterel has two forms of variables: shared variables (lowercase **s**) and sequential variables (**x**). Both of these variables refer to values and expressions in a host language, into which Esterel is embedded. For example, in Esterel v5 (Berry 2000) the host language is a subset of C, whereas in HipHop.js (Berry et al. 2011) the host language is Javascript.

Shared variables may be looked at or modified at by multiple branches of a `par` expression, but the program's execution cannot be influenced by the value of the variable until it can no

`Eval` : $\mathbf{p}\ \boldsymbol{\theta} \longrightarrow$ (Setof $\mathbf{S}$)
Runs the program for a single instant and returns the emitted signals

$\looparrowright$: **complete** $\longrightarrow$ **p**
Transforms a fully-reduced program to prepare it for the next instant

$\mathbf{p} \rightharpoonup \mathbf{q}$
Our notion of reduction; the primitive computational steps of our calculus

$\mathbf{p} \equiv_e \mathbf{q}$
The reflexive, transitive, symmetric, context closure of $\rightharpoonup$

`Can` : $\mathbf{p}\ \boldsymbol{\theta} \longrightarrow$ { S: (Setof $\mathbf{S}$), K: (Setof $\boldsymbol{\kappa}$), sh: (Setof $\mathbf{s}$) }
Determines the signals an expression can emit

$\vdash_{CB}\ \mathbf{p}$
A well-formedness condition on programs ensuring that signals and variables are well-behaved

Fig. 12. An overview of the main definitions

longer be written in the current instant (in a manner reminiscent of, but simpler than, Kuper and Newton (2013)'s LVars). Tracking if a shared variable is writable uses the same mechanism as tracking whether or not a signal has been emitted, and shared variables are subject to the logical correctness and constructivness constraints.

Multiple writes are allowed, but only via an associative and commutative combining operation, ensuring the order of writes is not observable. In our calculus we restrict shared variables to always be natural numbers and use + as the only combining operation.

Sequential variables may be used only sequentially (any given variable may not appear free in both branches of any specific `par` expression). This frees them from the constraints imposed on shared variables, allowing them to act as typical mutable variables without sacrificing deterministic concurrency. In our calculus they are bound to natural numbers and support a conventional conditional expression, `if`, that tests if the value is 0 or not.

For a fuller explanation of these features and how they behave, start with Potop-Butucaru et al. (2007)'s book *Compiling Esterel*, especially the first two chapers. The semantic rules in figure 14 also provide more details on how these constructs work.

3 THE ESTEREL CALCULUS

We define a reduction relation on program expressions that corresponds to a single-step of computation. This relation captures a notion of simplification, where each computational step brings us closer to a final answer. The relation is then closed over arbitrary contexts, and closed reflexively, symmetrically, and transitively, giving rise to a notion of when two programs are equivalent. The evaluator for the language is then defined simply by finding a fully simplified program that is equivalent to the input and returning it.

The remainder of this section explores the definitions that comprise the calculus, specifically the definitions shown in figure 12. Section 3.1 shows the basic notion of reduction that our calculus supports and section 3.2 describes our our `Can` function. The judgment form $\vdash_{CB}$ captures how

$$
\begin{array}{rl}
\mathbf{p, q} ::= & \ldots. \\
 & \mid (\varrho\ \boldsymbol{\theta}.\ \mathbf{p}) \\
 & \mid (\overline{\texttt{loop}}\ \mathbf{p}\ \mathbf{q}) \\
\mathbf{status} ::= & \texttt{present} \\
 & \mid \texttt{absent} \\
 & \mid \texttt{unknown} \\
\mathbf{shared\text{-}status} ::= & \texttt{ready} \mid \texttt{old} \mid \texttt{new} \\
\mathbf{complete} ::= & \mathbf{done} \mid (\varrho\ \boldsymbol{\theta}^{c}.\ \mathbf{done}) \\
\mathbf{done} ::= & \mathbf{stopped} \mid \mathbf{paused} \\
\mathbf{stopped} ::= & \texttt{nothing} \mid (\texttt{exit}\ \mathrm{n}) \\
\mathbf{paused} ::= & \texttt{pause} \\
 & \mid (\texttt{seq}\ \mathbf{paused}\ \mathbf{q}) \\
 & \mid (\overline{\texttt{loop}}\ \mathbf{paused}\ \mathbf{q}) \\
 & \mid (\texttt{par}\ \mathbf{paused}\ \mathbf{paused}) \\
 & \mid (\texttt{suspend}\ \mathbf{paused}\ \mathbf{S}) \\
 & \mid (\texttt{trap}\ \mathbf{paused}) \\
\mathbf{E} ::= & (\texttt{seq}\ \mathbf{E}\ \mathbf{q}) \\
 & \mid (\overline{\texttt{loop}}\ \mathbf{E}\ \mathbf{q}) \\
 & \mid (\texttt{par}\ \mathbf{E}\ \mathbf{q}) \\
 & \mid (\texttt{par}\ \mathbf{p}\ \mathbf{E}) \\
 & \mid (\texttt{suspend}\ \mathbf{E}\ \mathbf{S}) \\
 & \mid (\texttt{trap}\ \mathbf{E}) \\
 & \mid \texttt{[]}
\end{array}
$$

Metafunctions:

$$
\begin{array}{lcl}
\downarrow^{p}\ : \mathbf{stopped} \longrightarrow \mathbf{stopped} & & \\
\downarrow^{p} \texttt{nothing} & = & \texttt{nothing} \\
\downarrow^{p} (\texttt{exit 0}) & = & \texttt{nothing} \\
\downarrow^{p} (\texttt{exit n}) & = & (\texttt{exit n-1})
\end{array}
$$

Empty Environment: {}

Singleton Environments: { « var » $\mapsto$ « val » }
{ **S** $\mapsto$ **status** }
{ **s** $\mapsto$ ⟨n , **shared-status**⟩ }
{ **x** $\mapsto$ n }

Environment Composition: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}$
$(\boldsymbol{\theta}_1 \leftarrow \boldsymbol{\theta}_2)(\mathbf{S}) = \boldsymbol{\theta}_2(\mathbf{S})$ if $\mathbf{S} \in \mathrm{dom}(\boldsymbol{\theta}_2)$
$(\boldsymbol{\theta}_1 \leftarrow \boldsymbol{\theta}_2)(\mathbf{S}) = \boldsymbol{\theta}_1(\mathbf{S})$ if $\mathbf{S} \notin \mathrm{dom}(\boldsymbol{\theta}_2)$
... ditto for **s** and **x**

Complete Environments: $\boldsymbol{\theta}^{c}$
A complete environment is one where no signals are `unknown` and all shared variables are `ready`

Resetting Environments: $\lfloor \boldsymbol{\theta}^{c} \rfloor$
Resetting a complete environment updates all signals to `unknown` and all shared variables to `old`

Restricting the Domain: $(\boldsymbol{\theta} \setminus \{\mathbf{S}\})$
Restricting the domain of an environment removes the binding for **S**

Embedded host language expressions
e: host expressions
FV(**e**): all **x** and **s** that appear free in **e**
$\mathcal{E}val[\![\mathbf{e}\ ,\ \boldsymbol{\theta}]\!]$: evaluation; produces n

Fig. 13. Supplemental Structures

signals are to be used in Esterel programs, and is described in section 3.3. Finally section 3.4 gives the `Eval` and the $\rightsquigarrow$ functions and the central result of this work, namely that `Eval` is a function.

Before diving into the rules, however, we need a to extend the **p** non-terminal to track information about the term as it reduces. Figure 13 shows the two extensions. First, the $(\overline{\texttt{loop}}\ \mathbf{p}\ \mathbf{q})$ expression form is similar to a (`seq` **p** (`loop` **q**)) and is used by the loop reduction rule (discussed in section 3.1).

The other extension is the $(\varrho\ \boldsymbol{\theta}.\ \mathbf{p})$ expression form, the heart of our calculus. It pairs an environment ($\boldsymbol{\theta}$) with an Esterel expression. The environment records what we have learned about the signals and variables in this instant for the contained subexpression, and various rules either add information to the $\boldsymbol{\theta}$ or exploit information recorded as the program reduces. We keep the environments local to specific expressions in order to facilitate local reasoning.

3.1 Reduction Rules

The rules given in figure 14 govern how computation takes place within a single instant.

signals

$(\mathtt{signal}\ \mathbf{S}\ \mathbf{p}) \rightharpoonup (\rho\ \{\mathbf{S} \mapsto \mathtt{unknown}\}.\ \mathbf{p})$ **[signal]**

$(\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\mathtt{emit}\ \mathbf{S})]) \rightharpoonup (\rho\ (\boldsymbol{\theta} \leftarrow \{\mathbf{S} \mapsto \mathtt{present}\}).\ \mathbf{E}[\mathtt{nothing}])$ **[emit]**
where $\boldsymbol{\theta}(\mathbf{S}) \in \{\mathtt{present}, \mathtt{unknown}\}$

$(\rho\ \boldsymbol{\theta}.\ \mathbf{p}) \rightharpoonup (\rho\ (\boldsymbol{\theta} \leftarrow \{\mathbf{S} \mapsto \mathtt{absent}\}).\ \mathbf{p})$ **[absence]**
where $\mathbf{S} \in \mathrm{dom}(\boldsymbol{\theta})$, $\mathbf{S} \notin \mathtt{Can}_\rho[\![(\rho\ \boldsymbol{\theta}.\ \mathbf{p}), \{\}]\!].S$, $\boldsymbol{\theta}(\mathbf{S}) = \mathtt{unknown}$

$(\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\mathtt{present}\ \mathbf{S}\ \mathbf{p}\ \mathbf{q})]) \rightharpoonup (\rho\ \boldsymbol{\theta}.\ \mathbf{E}[\mathbf{p}])$ where $\boldsymbol{\theta}(\mathbf{S}) = \mathtt{present}$ **[is-present]**

$(\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\mathtt{present}\ \mathbf{S}\ \mathbf{p}\ \mathbf{q})]) \rightharpoonup (\rho\ \boldsymbol{\theta}.\ \mathbf{E}[\mathbf{q}])$ where $\boldsymbol{\theta}(\mathbf{S}) = \mathtt{absent}$ **[is-absent]**

shared variables

$(\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\mathtt{shared}\ \mathbf{s} \coloneqq \mathbf{e}\ \mathbf{p})]) \rightharpoonup (\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\rho\ \{\mathbf{s} \mapsto \langle \mathrm{n}, \mathtt{old}\rangle\}.\ \mathbf{p})])$ **[shared]**
where $\mathrm{FV}(\mathbf{e}) \subset \mathrm{dom}(\boldsymbol{\theta})$, $\forall\ \mathrm{s} \in \mathrm{FV}(\mathbf{e}).\ \boldsymbol{\theta}(\mathrm{s}) = \langle \_, \mathtt{ready}\rangle$, $\mathrm{n} = \mathcal{E}val[\![\mathbf{e}, \boldsymbol{\theta}]\!]$

$(\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\mathtt{+=}\ \mathbf{s}\ \mathbf{e})]) \rightharpoonup (\rho\ (\boldsymbol{\theta} \leftarrow \{\mathbf{s} \mapsto \langle \mathcal{E}val[\![\mathbf{e}, \boldsymbol{\theta}]\!], \mathtt{new}\rangle\}).\ \mathbf{E}[\mathtt{nothing}])$ **[set-old]**
where $\boldsymbol{\theta}(\mathbf{s}) = \langle \_, \mathtt{old}\rangle$, $\mathrm{FV}(\mathbf{e}) \subset \mathrm{dom}(\boldsymbol{\theta})$, $\forall\ \mathrm{s} \in \mathrm{FV}(\mathbf{e}).\ \boldsymbol{\theta}(\mathrm{s}) = \langle \_, \mathtt{ready}\rangle$

$(\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\mathtt{+=}\ \mathbf{s}\ \mathbf{e})]) \rightharpoonup (\rho\ (\boldsymbol{\theta} \leftarrow \{\mathbf{s} \mapsto \langle \mathrm{n} + \mathcal{E}val[\![\mathbf{e}, \boldsymbol{\theta}]\!], \mathtt{new}\rangle\}).\ \mathbf{E}[\mathtt{nothing}])$ **[set-new]**
where $\boldsymbol{\theta}(\mathbf{s}) = \langle \mathrm{n}, \mathtt{new}\rangle$, $\mathrm{FV}(\mathbf{e}) \subset \mathrm{dom}(\boldsymbol{\theta})$, $\forall\ \mathrm{s} \in \mathrm{FV}(\mathbf{e}).\ \boldsymbol{\theta}(\mathrm{s}) = \langle \_, \mathtt{ready}\rangle$

$(\rho\ \boldsymbol{\theta}.\ \mathbf{p}) \rightharpoonup (\rho\ (\boldsymbol{\theta} \leftarrow \{\mathbf{s} \mapsto \langle \mathrm{n}, \mathtt{ready}\rangle\}).\ \mathbf{p})$ **[readyness]**
where $\mathbf{s} \in \mathrm{dom}(\boldsymbol{\theta})$, $\mathbf{s} \notin \mathtt{Can}_\rho[\![(\rho\ \boldsymbol{\theta}.\ \mathbf{p}), \{\}]\!].sh$, $\boldsymbol{\theta}(\mathbf{s}) = \langle \mathrm{n}, \mathbf{shared\text{-}status}\rangle$,
$\mathbf{shared\text{-}status} \in \{\mathtt{old}, \mathtt{new}\}$

sequential variables

$(\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\mathtt{var}\ \mathbf{x} \coloneqq \mathbf{e}\ \mathbf{p})]) \rightharpoonup (\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\rho\ \{\mathbf{x} \mapsto \mathcal{E}val[\![\mathbf{e}, \boldsymbol{\theta}]\!]\}.\ \mathbf{p})])$ **[var]**
where $\mathrm{FV}(\mathbf{e}) \subset \mathrm{dom}(\boldsymbol{\theta})$, $\forall\ \mathrm{s} \in \mathrm{FV}(\mathbf{e}).\ \boldsymbol{\theta}(\mathrm{s}) = \langle \_, \mathtt{ready}\rangle$

$(\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\coloneqq\ \mathbf{x}\ \mathbf{e})]) \rightharpoonup (\rho\ (\boldsymbol{\theta} \leftarrow \{\mathbf{x} \mapsto \mathcal{E}val[\![\mathbf{e}, \boldsymbol{\theta}]\!]\}).\ \mathbf{E}[\mathtt{nothing}])$ **[set-var]**
where $\mathbf{x} \in \mathrm{dom}(\boldsymbol{\theta})$, $\mathrm{FV}(\mathbf{e}) \subset \mathrm{dom}(\boldsymbol{\theta})$, $\forall\ \mathrm{s} \in \mathrm{FV}(\mathbf{e}).\ \boldsymbol{\theta}(\mathrm{s}) = \langle \_, \mathtt{ready}\rangle$

$(\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\mathtt{if}\ \mathbf{x}\ \mathbf{p}\ \mathbf{q})]) \rightharpoonup (\rho\ \boldsymbol{\theta}.\ \mathbf{E}[\mathbf{p}])$ where $\mathbf{x} \in \mathrm{dom}(\boldsymbol{\theta})$, $\boldsymbol{\theta}(\mathbf{x}) \neq 0$ **[if-true]**

$(\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\mathtt{if}\ \mathbf{x}\ \mathbf{p}\ \mathbf{q})]) \rightharpoonup (\rho\ \boldsymbol{\theta}.\ \mathbf{E}[\mathbf{q}])$ where $\boldsymbol{\theta}(\mathbf{x}) = 0$ **[if-false]**

ρ

$(\rho\ \boldsymbol{\theta}_1.\ \mathbf{E}[(\rho\ \boldsymbol{\theta}_2.\ \mathbf{p})]) \rightharpoonup (\rho\ (\boldsymbol{\theta}_1 \leftarrow \boldsymbol{\theta}_2).\ \mathbf{E}[\mathbf{p}])$ **[merge]**

seq

$(\mathtt{seq}\ \mathtt{nothing}\ \mathbf{q}) \rightharpoonup \mathbf{q}$ **[seq-done]**

$(\mathtt{seq}\ (\mathtt{exit}\ \mathrm{n})\ \mathbf{q}) \rightharpoonup (\mathtt{exit}\ \mathrm{n})$ **[seq-exit]**

trap

$(\mathtt{trap}\ \mathbf{stopped}) \rightharpoonup \downarrow^{\mathrm{p}} \mathbf{stopped}$ **[trap]**

par

$(\mathtt{par}\ \mathtt{nothing}\ \mathbf{done}) \rightharpoonup \mathbf{done}$ **[par-nothing]**

$(\mathtt{par}\ (\mathtt{exit}\ \mathrm{n})\ \mathbf{paused}) \rightharpoonup (\mathtt{exit}\ \mathrm{n})$ **[par-1exit]**

$(\mathtt{par}\ (\mathtt{exit}\ \mathrm{n}_1)\ (\mathtt{exit}\ \mathrm{n}_2)) \rightharpoonup (\mathtt{exit}\ \max(\mathrm{n}_1, \mathrm{n}_2))$ **[par-2exit]**

$(\mathtt{par}\ \mathbf{p}\ \mathbf{q}) \rightharpoonup (\mathtt{par}\ \mathbf{q}\ \mathbf{p})$ **[par-swap]**

$(\mathtt{suspend}\ \mathbf{stopped}\ \mathbf{S}) \rightharpoonup \mathbf{stopped}$ **[suspend]**

loop

$(\mathtt{loop}\ \mathbf{p}) \rightharpoonup (\overline{\mathtt{loop}}\ \mathbf{p}\ \mathbf{p})$ **[loop]**

$(\overline{\mathtt{loop}}\ (\mathtt{exit}\ \mathrm{n})\ \mathbf{q}) \rightharpoonup (\mathtt{exit}\ \mathrm{n})$ **[loop^stop-exit]**

Fig. 14. Reduction Rules

The first rule, [**signal**], reduces a `signal` expression to a ϱ expression by introducing a singleton **θ** that binds the signal to `unknown`.

Once a signal has an entry in a relevant **θ**, the [**emit**] rule records that a signal is present (using the composition operator ⟵ from figure 13) and eliminates the `emit` expression. The side-condition ensures that the environment **θ** does not already indicate that the signal is absent.

In order for an `emit` to fire, it must be in the body of a ϱ in only a specific set of positions, as captured by the **E** contexts, shown in figure 13. They include the first sub-expression of a `seq` expression, the first sub-expression of a ($\overline{\texttt{loop}}$ **p** **q**) expression, either branch of a `par`, the body of a `suspend` or a `trap`. Notably this rule does not allow an expression like (ϱ **θ**. (`seq` **p** (`emit` **S**))) to reduce its `emit` expression because the **p** could be `pause`, delaying the (`emit` **S**) to the next instant. More generally, the expressions captured by **E** are guaranteed to happen in the current instant.

As we saw in section 2.3, `Can` determines if a signal cannot be emitted. The rule [**absence**] uses $\texttt{Can}_{\varrho}$ (a variation of `Can` that is explained in section 3.2) to determine that a signal cannot be emitted and records that information in a **θ** expression, if that information is not yet recorded.

Once the status of a signal is recorded as either present or absent, the [**is-present**] and [**is-absent**] rules can reduce `present` expressions.

The rules [**shared**], [**set-old**], [**set-new**], and [**readyness**] handle shared variables in a manner similar to how the previous set of rules handle signals. The [**shared**] rule introduces a new environment that binds the shared variable using the **e** in the `shared` expression to determine the default value of the variable using the host language's evaluation function. The rules [**set-old**] and [**set-new**] modify a shared variable depending on whether it has been modified in the current instant or not. If the status of a shared variable in the environment is `old`, it is being modified for the first time in the current instant and the rule [**set-old**] replaces the old value in the environment with the new value. If the status of a shared variable is `new`, it has already been modified in the current instant and the rule [**set-new**] adds the current value and the new value in the += expression and stores the result in the environment. Finally, the [**readyness**] rule makes a variable change from writable to readable. This occurs if $\texttt{Can}_{\varrho}$'s result does not contain the shared variable **s**, which means it will not be modified in this instant and thus we can update the environment to mark the variable as `ready`. Furthermore, the side-conditions on the [**shared**], [**set-new**], and [**set-old**] rules (as well as the corresponding rules for sequential variables) ensure that these rules can fire only if, for every shared variable used in the host language expression, that variable safe to be read, e.g. is marked as `ready` in **θ**.

The rules [**var**], [**set-var**], [**if-true**], and [**if-false**] cover sequential variables. Unlike the rules for signals or shared variables, these rules do not refer to `Can`. These variables are not allowed to be free in two different arms of any `par` expression, so they can be freely read and written.

The final rule that handles ϱ expressions is [**merge**]. It combines two environments, lifting an inner environment out to an outer one and composing them into a single environment.

There are two rules for sequential composition. If the first sub-expression is `nothing`, then we replace the entire expression with the second branch. If the first sub-expression is an `exit` expression, however, then the entire sequence exits, discarding the second part of the `seq` expression.

The next rule handles `trap`. Once the body of a `trap` has finished evaluating, it will either be an `exit` expression or `nothing`, and the $\downarrow^{\mathrm{p}}$ (figure 13) function handles them.

The `par` rules are a little more interesting. The first three refer to to the definitions of **stopped** and **done** in figure 13 and handle the situations when both branches are finished for the instant. If one side has reduced to nothing, the [**par-nothing**] rule reduces to the other one. If one side has `exited` and the other is **paused**, the [**par-1exit**] rule preempts the other branch of the par by bubbling the `exit` up. If both sides have `exited` the [**par-2exit**] rule reduces the expression to whichever `exit` will reach the farthest up `trap`. The [**par-swap**] rule switches the branches, allowing the [**par-nothing**] and [**par-1exit**] rule to match regardless of which branch is `exit` or `nothing`.

The [**suspend**] rule reduces to its body when its body has either `exited` or reduced to `nothing`.

This leaves us with one last pair of rules: [**loop**] and [**loop^stop-exit**]. Intuitively, we would like an expression like (`loop` **p**) to reduce simply to just (`seq` **p** (`loop` **p**)), duplicating the body **p** into a `seq` expression which becomes the current iteration of the loop.

Such a rule could give rise to infinite loops within a single instant, however, which is forbidden in Esterel. We capture this constraint in our calculus with the $\overline{\texttt{loop}}$ expression form. It is introduced only by the reduction rule for `loop`, and is meant to capture a single unrolling of the loop; the first sub-expression is the part of the loop that runs in the current instant and the second sub-expression is the body of the loop, saved to be used in the next instant. There is no rule that eliminates a $\overline{\texttt{loop}}$ when the first sub-expression is `nothing` (unlike `seq`, which has the [**seq-done**] rule). As such, programs get stuck when they contain instantaneous loops.

One thing to note about these rules: with the exception of [**par-swap**], they are strongly normalizing. The proof is given as `noetherian` in Agda code in the supplementary material.

3.2 The Can Function

This section describes the function `Can`, a conservative analysis of the state of an Esterel program that determines its behavior. Our definition is inspired by Berry (2002)'s definition, generalized to support ϱ expressions and modified to handle a reduction semantics rather than one based on annotating the program with program counters.

This function computes a conservative approximation to the behavior of some given Esterel expression with respect to some knowledge about signals and shared variables that is encapsulated in an environment, **θ**. In particular, it computes a set of signals (**S**), a set of exit codes (**κ**), and a set of shared variables (**s**). Any **S** that is not in the result is guaranteed not to be emitted in the current instant (although if some **S** is in the result, it may or may not be emitted in the current instant). Ditto for any shared variable in the result: if an **s** is not the result, then it is guaranteed that **s** is not going to be updated again in the current instant. If the **s** is in the result, then it may or may not be updated. The exit codes capture whether or not the given expression pauses, reduces to `nothing`, or exits. If the expression may reduce to `nothing`, then the code `nothin` will be in the result. If the expression may pause, then the code `paus` will be in the result. If the expression may exit with the code n, then the code n will be in the result. Thus, if any of those specific codes are *not* in the result, then we know the expression does not have the corresponding behavior.

The notation we use for the records in the definition of `Can` is similar to many record notations, but we use the precise one in Pierce (2002)'s book *Types and Programming Languages*. We write `Can`'s result as a record with three fields, where curly braces construct records, e.g., the `emit` case of `Can` returns a record with a singleton set of signals (containing **S**), a singleton set of exit codes (containing `nothin`) and the empty set of shared variables. Selecting a field from a record uses dot notation. For example, `Can`⟦**p**, **θ**⟧.S selects the "S" field from a call to `Can`.

The three results from `Can` interact with each other in order to determine the overall result. Consider the two `seq` cases. In the first one, the side-condition says that `nothin` is not in the **K** field for the first sub-expression of the `seq`, **p**. Accordingly, we know that **p** does not reduce to `nothing`, thus it must either exit or pause. Since it exits or pauses, we know that none of the

$\boldsymbol{\kappa} ::= \mathtt{nothin} \mid \mathtt{paus} \mid \mathrm{n}$

$$\begin{aligned}
&\mathtt{Can} : \mathbf{p}\ \boldsymbol{\theta} \longrightarrow \{\ \mathrm{S}\text{: (Setof } \mathbf{S}),\ \mathrm{K}\text{: (Setof } \boldsymbol{\kappa}),\ \mathrm{sh}\text{: (Setof } \mathbf{s})\ \}\\
&\mathtt{Can}[\![(\varrho\ \boldsymbol{\theta}_1.\ \mathbf{p}),\ \boldsymbol{\theta}_2]\!] && = \{\ \mathrm{S} = \mathtt{Can}_\varrho[\![(\varrho\ \boldsymbol{\theta}_1.\ \mathbf{p}),\ \boldsymbol{\theta}_2]\!].\mathrm{S} \setminus \mathrm{dom}(\boldsymbol{\theta}_1),\\
& && \quad \mathrm{K} = \mathtt{Can}_\varrho[\![(\varrho\ \boldsymbol{\theta}_1.\ \mathbf{p}),\ \boldsymbol{\theta}_2]\!].\mathrm{K},\\
& && \quad \mathrm{sh} = \mathtt{Can}_\varrho[\![(\varrho\ \boldsymbol{\theta}_1.\ \mathbf{p}),\ \boldsymbol{\theta}_2]\!].\mathrm{sh} \setminus \mathrm{dom}(\boldsymbol{\theta}_1)\ \}\\
&\mathtt{Can}[\![\mathtt{nothing},\ \boldsymbol{\theta}]\!] && = \{\ \mathrm{S} = \emptyset,\ \mathrm{K} = \{\ \mathtt{nothin}\ \},\ \mathrm{sh} = \emptyset\ \}\\
&\mathtt{Can}[\![\mathtt{pause},\ \boldsymbol{\theta}]\!] && = \{\ \mathrm{S} = \emptyset,\ \mathrm{K} = \{\ \mathtt{paus}\ \},\ \mathrm{sh} = \emptyset\ \}\\
&\mathtt{Can}[\![(\mathtt{exit}\ \mathrm{n}),\ \boldsymbol{\theta}]\!] && = \{\ \mathrm{S} = \emptyset,\ \mathrm{K} = \{\ \mathrm{n}\ \},\ \mathrm{sh} = \emptyset\ \}\\
&\mathtt{Can}[\![(\mathtt{emit}\ \mathbf{S}),\ \boldsymbol{\theta}]\!] && = \{\ \mathrm{S} = \{\ \mathbf{S}\ \},\ \mathrm{K} = \{\ \mathtt{nothin}\ \},\ \mathrm{sh} = \emptyset\ \}\\
&\mathtt{Can}[\![(\mathtt{present}\ \mathbf{S}\ \mathbf{p}\ \mathbf{q}),\ \boldsymbol{\theta}]\!] && = \mathtt{Can}[\![\mathbf{p},\ \boldsymbol{\theta}]\!]\\
&\quad \text{where } \boldsymbol{\theta}(\mathbf{S}) = \mathtt{present}\\
&\mathtt{Can}[\![(\mathtt{present}\ \mathbf{S}\ \mathbf{p}\ \mathbf{q}),\ \boldsymbol{\theta}]\!] && = \mathtt{Can}[\![\mathbf{q},\ \boldsymbol{\theta}]\!]\\
&\quad \text{where } \boldsymbol{\theta}(\mathbf{S}) = \mathtt{absent}\\
&\mathtt{Can}[\![(\mathtt{present}\ \mathbf{S}\ \mathbf{p}\ \mathbf{q}),\ \boldsymbol{\theta}]\!] && = \{\ \mathrm{S} = \mathtt{Can}[\![\mathbf{p},\ \boldsymbol{\theta}]\!].\mathrm{S} \cup \mathtt{Can}[\![\mathbf{q},\ \boldsymbol{\theta}]\!].\mathrm{S},\\
& && \quad \mathrm{K} = \mathtt{Can}[\![\mathbf{p},\ \boldsymbol{\theta}]\!].\mathrm{K} \cup \mathtt{Can}[\![\mathbf{q},\ \boldsymbol{\theta}]\!].\mathrm{K},\\
& && \quad \mathrm{sh} = \mathtt{Can}[\![\mathbf{p},\ \boldsymbol{\theta}]\!].\mathrm{sh} \cup \mathtt{Can}[\![\mathbf{q},\ \boldsymbol{\theta}]\!].\mathrm{sh}\ \}\\
&\mathtt{Can}[\![(\mathtt{suspend}\ \mathbf{p}\ \mathbf{S}),\ \boldsymbol{\theta}]\!] && = \mathtt{Can}[\![\mathbf{p},\ \boldsymbol{\theta}]\!]\\
&\mathtt{Can}[\![(\mathtt{seq}\ \mathbf{p}\ \mathbf{q}),\ \boldsymbol{\theta}]\!] && = \mathtt{Can}[\![\mathbf{p},\ \boldsymbol{\theta}]\!]\\
&\quad \text{where } \mathtt{nothin} \notin \mathtt{Can}[\![\mathbf{p},\ \boldsymbol{\theta}]\!].\mathrm{K}\\
&\mathtt{Can}[\![(\mathtt{seq}\ \mathbf{p}\ \mathbf{q}),\ \boldsymbol{\theta}]\!] && = \{\ \mathrm{S} = \mathtt{Can}[\![\mathbf{p},\ \boldsymbol{\theta}]\!].\mathrm{S} \cup \mathtt{Can}[\![\mathbf{q},\ \boldsymbol{\theta}]\!].\mathrm{S},\\
& && \quad \mathrm{K} = \mathtt{Can}[\![\mathbf{p},\ \boldsymbol{\theta}]\!].\mathrm{K} \setminus \{\ \mathtt{nothin}\ \} \cup \mathtt{Can}[\![\mathbf{q},\ \boldsymbol{\theta}]\!].\mathrm{K},\\
& && \quad \mathrm{sh} = \mathtt{Can}[\![\mathbf{p},\ \boldsymbol{\theta}]\!].\mathrm{sh} \cup \mathtt{Can}[\![\mathbf{q},\ \boldsymbol{\theta}]\!].\mathrm{sh}\ \}\\
&\mathtt{Can}[\![(\mathtt{loop}\ \mathbf{p}),\ \boldsymbol{\theta}]\!] && = \mathtt{Can}[\![\mathbf{p},\ \boldsymbol{\theta}]\!]\\
&\mathtt{Can}[\![(\overline{\mathtt{loop}}\ \mathbf{p}\ \mathbf{q}),\ \boldsymbol{\theta}]\!] && = \mathtt{Can}[\![\mathbf{p},\ \boldsymbol{\theta}]\!]\\
&\mathtt{Can}[\![(\mathtt{par}\ \mathbf{p}\ \mathbf{q}),\ \boldsymbol{\theta}]\!] && = \{\ \mathrm{S} = \mathtt{Can}[\![\mathbf{p},\ \boldsymbol{\theta}]\!].\mathrm{S} \cup \mathtt{Can}[\![\mathbf{q},\ \boldsymbol{\theta}]\!].\mathrm{S},\\
& && \quad \mathrm{K} = \{\ \max(\boldsymbol{\kappa}_1,\ \boldsymbol{\kappa}_2) \mid \boldsymbol{\kappa}_1 \in \mathtt{Can}[\![\mathbf{p},\ \boldsymbol{\theta}]\!].\mathrm{K},\ \boldsymbol{\kappa}_2 \in \mathtt{Can}[\![\mathbf{q},\ \boldsymbol{\theta}]\!].\mathrm{K}\ \},\\
& && \quad \mathrm{sh} = \mathtt{Can}[\![\mathbf{p},\ \boldsymbol{\theta}]\!].\mathrm{sh} \cup \mathtt{Can}[\![\mathbf{q},\ \boldsymbol{\theta}]\!].\mathrm{sh}\ \}\\
&\mathtt{Can}[\![(\mathtt{trap}\ \mathbf{p}),\ \boldsymbol{\theta}]\!] && = \{\ \mathrm{S} = \mathtt{Can}[\![\mathbf{p},\ \boldsymbol{\theta}]\!].\mathrm{S},\ \mathrm{K} = \{\ {\downarrow^{\kappa}}\ \mathrm{x} \mid \mathrm{x} \in \mathtt{Can}[\![\mathbf{p},\ \boldsymbol{\theta}]\!].\mathrm{K}\ \},\ \mathrm{sh} = \mathtt{Can}[\![\mathbf{p},\ \boldsymbol{\theta}]\!].\mathrm{sh}\ \}\\
&\mathtt{Can}[\![(\mathtt{signal}\ \mathbf{S}\ \mathbf{p}),\ \boldsymbol{\theta}]\!] && = \{\ \mathrm{S} = \mathtt{Can}[\![\mathbf{p},\ \boldsymbol{\theta} \longleftarrow \{\ \mathbf{S} \mapsto \mathtt{absent}\ \}]\!].\mathrm{S} \setminus \{\ \mathbf{S}\ \},\\
& && \quad \mathrm{K} = \mathtt{Can}[\![\mathbf{p},\ \boldsymbol{\theta} \longleftarrow \{\ \mathbf{S} \mapsto \mathtt{absent}\ \}]\!].\mathrm{K},\\
& && \quad \mathrm{sh} = \mathtt{Can}[\![\mathbf{p},\ \boldsymbol{\theta} \longleftarrow \{\ \mathbf{S} \mapsto \mathtt{absent}\ \}]\!].\mathrm{sh}\ \}\\
&\quad \text{where } \mathbf{S} \notin \mathtt{Can}[\![\mathbf{p},\ \boldsymbol{\theta} \longleftarrow \{\ \mathbf{S} \mapsto \mathtt{unknown}\ \}]\!].\mathrm{S}\\
&\mathtt{Can}[\![(\mathtt{signal}\ \mathbf{S}\ \mathbf{p}),\ \boldsymbol{\theta}]\!] && = \{\ \mathrm{S} = \mathtt{Can}[\![\mathbf{p},\ \boldsymbol{\theta}_2]\!].\mathrm{S} \setminus \{\ \mathbf{S}\ \},\ \mathrm{K} = \mathtt{Can}[\![\mathbf{p},\ \boldsymbol{\theta}_2]\!].\mathrm{K},\ \mathrm{sh} = \mathtt{Can}[\![\mathbf{p},\ \boldsymbol{\theta}_2]\!].\mathrm{sh}\ \}\\
&\quad \text{where } \boldsymbol{\theta}_2 = \boldsymbol{\theta} \longleftarrow \{\ \mathbf{S} \mapsto \mathtt{unknown}\ \}\\
&\mathtt{Can}[\![(\mathtt{shared}\ \mathbf{s} \coloneqq \mathbf{e}\ \mathbf{p}),\ \boldsymbol{\theta}]\!] && = \{\ \mathrm{S} = \mathtt{Can}[\![\mathbf{p},\ \boldsymbol{\theta}]\!].\mathrm{S},\ \mathrm{K} = \mathtt{Can}[\![\mathbf{p},\ \boldsymbol{\theta}]\!].\mathrm{K},\ \mathrm{sh} = \mathtt{Can}[\![\mathbf{p},\ \boldsymbol{\theta}]\!].\mathrm{sh} \setminus \{\ \mathbf{s}\ \}\ \}\\
&\mathtt{Can}[\![(\mathtt{+=}\ \mathbf{s}\ \mathbf{e}),\ \boldsymbol{\theta}]\!] && = \{\ \mathrm{S} = \emptyset,\ \mathrm{K} = \{\ \mathtt{nothin}\ \},\ \mathrm{sh} = \{\ \mathbf{s}\ \}\ \}\\
&\mathtt{Can}[\![(\mathtt{var}\ \mathbf{x} \coloneqq \mathbf{e}\ \mathbf{p}),\ \boldsymbol{\theta}]\!] && = \mathtt{Can}[\![\mathbf{p},\ \boldsymbol{\theta}]\!]\\
&\mathtt{Can}[\![(\coloneqq\ \mathbf{x}\ \mathbf{e}),\ \boldsymbol{\theta}]\!] && = \{\ \mathrm{S} = \emptyset,\ \mathrm{K} = \{\ \mathtt{nothin}\ \},\ \mathrm{sh} = \emptyset\ \}\\
&\mathtt{Can}[\![(\mathtt{if}\ \mathbf{x}\ \mathbf{p}\ \mathbf{q}),\ \boldsymbol{\theta}]\!] && = \{\ \mathrm{S} = \mathtt{Can}[\![\mathbf{p},\ \boldsymbol{\theta}]\!].\mathrm{S} \cup \mathtt{Can}[\![\mathbf{q},\ \boldsymbol{\theta}]\!].\mathrm{S},\\
& && \quad \mathrm{K} = \mathtt{Can}[\![\mathbf{p},\ \boldsymbol{\theta}]\!].\mathrm{K} \cup \mathtt{Can}[\![\mathbf{q},\ \boldsymbol{\theta}]\!].\mathrm{K},\\
& && \quad \mathrm{sh} = \mathtt{Can}[\![\mathbf{p},\ \boldsymbol{\theta}]\!].\mathrm{sh} \cup \mathtt{Can}[\![\mathbf{q},\ \boldsymbol{\theta}]\!].\mathrm{sh}\ \}
\end{aligned}$$

$$\boxed{\begin{aligned}
&{\downarrow^{\kappa}} : \boldsymbol{\kappa} \longrightarrow \boldsymbol{\kappa}\\
&{\downarrow^{\kappa}}\ \mathtt{nothin} = \mathtt{nothin}\\
&{\downarrow^{\kappa}}\ \mathtt{paus} = \mathtt{paus}\\
&{\downarrow^{\kappa}}\ 0 = \mathtt{nothin}\\
&{\downarrow^{\kappa}}\ \mathrm{n} = \mathrm{n}\text{-}1\\
&\text{if } \mathrm{n} > 0
\end{aligned}}$$

$$\boxed{\begin{aligned}
&\max : \boldsymbol{\kappa}\ \boldsymbol{\kappa} \longrightarrow \boldsymbol{\kappa}\\
&\quad \max(\boldsymbol{\kappa}_1,\ \boldsymbol{\kappa}_2) \text{ computes}\\
&\quad \text{the maximum of } \boldsymbol{\kappa}_1 \text{ and}\\
&\quad \boldsymbol{\kappa}_2 \text{ where we define}\\
&\quad \mathtt{nothin} < \mathtt{paus} < 0 < 1 < \ldots
\end{aligned}}$$

Fig. 15. Can Function

$$
\begin{array}{l}
\mathsf{Can}_{\varrho} : (\varrho\ \boldsymbol{\theta}.\ \mathbf{p})\ \boldsymbol{\theta} \longrightarrow \{\ \mathrm{S}\text{: (Setof } \mathbf{S}\text{), K: (Setof } \boldsymbol{\kappa}\text{), sh: (Setof } \mathbf{s}\text{)}\ \}\\
\mathsf{Can}_{\varrho}[\![(\varrho\ \boldsymbol{\theta}.\ \mathbf{p}), \boldsymbol{\theta}_2]\!] \;=\; \mathsf{Can}_{\varrho}[\![(\varrho\ (\boldsymbol{\theta} \setminus \{\mathbf{S}\}).\ \mathbf{p}), \boldsymbol{\theta}_2 \leftarrow \{\ \mathbf{S} \mapsto \mathsf{absent}\ \}]\!]\\
\quad \text{where } \mathbf{S} \in \mathrm{dom}(\boldsymbol{\theta}),\\
\qquad\qquad \boldsymbol{\theta}(\mathbf{S}) = \mathsf{unknown},\\
\qquad\qquad \mathbf{S} \notin \mathsf{Can}_{\varrho}[\![(\varrho\ (\boldsymbol{\theta} \setminus \{\mathbf{S}\}).\ \mathbf{p}), \boldsymbol{\theta}_2 \leftarrow \{\ \mathbf{S} \mapsto \mathsf{unknown}\ \}]\!].\mathrm{S}\\
\mathsf{Can}_{\varrho}[\![(\varrho\ \boldsymbol{\theta}.\ \mathbf{p}), \boldsymbol{\theta}_2]\!] \;=\; \mathsf{Can}_{\varrho}[\![(\varrho\ (\boldsymbol{\theta} \setminus \{\mathbf{S}\}).\ \mathbf{p}), \boldsymbol{\theta}_2 \leftarrow \{\ \mathbf{S} \mapsto \boldsymbol{\theta}(\mathbf{S})\ \}]\!]\\
\quad \text{where } \mathbf{S} \in \mathrm{dom}(\boldsymbol{\theta})\\
\mathsf{Can}_{\varrho}[\![(\varrho\ \boldsymbol{\theta}_1.\ \mathbf{p}), \boldsymbol{\theta}_2]\!] \;=\; \mathsf{Can}[\![\mathbf{p}, \boldsymbol{\theta}_2]\!]
\end{array}
$$

Fig. 16. The Can Function for ϱ Expressions (cases are checked in order)

behavior of **q** is relevant as it will not be evaluated in this instant and so the result of Can for the entire seq expression is just its result for the **p** expression. This means that

$$\mathsf{Can}[\![(\mathsf{seq\ pause\ (emit}\ \mathbf{S})), \boldsymbol{\theta}]\!].\mathrm{S}$$

is the empty set, since the emit must happen in the next instant.

In the second seq case, we know that nothin was a possible result code, and thus **p** might reduce to nothing so we have to combine the result of the **p** and **q** recursive calls. Mostly this amounts to simply unioning them, but note that the **K** case removes nothin from the codes in the result of **p** before performing the union. This removal accounts for the fact that, even if **p** reduces to nothing, **q** must also reduce to nothing for the seq expression to reduce to nothing. For example,

$$\mathsf{Can}[\![(\mathsf{seq\ nothing\ pause}), \boldsymbol{\theta}]\!].\mathrm{K}$$

correctly contains only the exit code paus.

The $\overline{\mathsf{loop}}$ expression form, in contrast, always ignores the second sub-expression, because we know that the second sub-expression can affect only future instants.

Various other cases in the definition of Can reflect the semantics of the different constructs in similar ways. The cases handling present consult the given **θ** to see if the status of the signal is known and look only at the corresponding branch of the present expression if so. The rule for par takes into account the same behavior that the four par rules in the reduction relation do when computing the codes for the entire expression out of the codes of the subexpressions. The trap case uses the metafunction $\downarrow^{\kappa}$ to adjust the exit codes in a manner that mimics how trap expressions reduce. Since the shared form introduces a new signal, its case in Can removes that signal from the results, as the signal is lexically scoped. In each of these cases, Can ignores the **e** expressions, as it does not reason about the behavior of the host language.

This leaves the signal and ϱ cases. Consider first the cases that handle signal expressions. The second signal case is the more straightforward one. It says that the result for the entire signal form is the same as the result for the body of a signal form when it is analyzed with no knowledge about the signal. But there would be a problem with the Can function if that were the only case.

To motivate the first signal case in Can, consider this call:

```
Can⟦(signal S2 (present S2
                         (emit S1)
                         nothing)), {}⟧.S
```

If we took the second signal case in Can, then this would return a set containing S1. It actually returns the empty set, however, because of that first case. In particular, that case first calls Can with

S2 set to unknown and checks to see if S2 is not present in the "S" portion of the result. It is not (because there are no (emit S2) expressions), so Can then sets S2 to absent and reprocesses its body. This time, because S2 is known to be absent, Can considers only the last sub-expression of the present, thereby ignoring the (emit S1) and returning the empty set of signals.

In isolation, analyzing the body twice seems like overkill, especially because it triggers exponential behavior in the number of nested signal forms.[3] But consider this call to Can:

```
Can⟦(signal S1
         (seq (present S1 pause nothing)
              (signal S2 (present S2
                                  (emit S1)
                                  nothing)))), {}⟧.S
```

This example input is a bit complex, but first notice that the inner signal expression is the same as the previous example (and there are no other (emit S1) expressions), so we know that S1 is not going to be emitted. If Can did not have that first signal case, then it could not learn that S1 cannot be emitted and thus we would not be able to use the [**absence**] rule on this expression.

Finally, for the ϱ case, the Can function dispatches to Can_ϱ. The Can_ϱ function looks complex, but it is essentially the same as the two signal cases. It is broken out into its own function because ϱ binds multiple signals at once; so Can_ϱ recurs though the structure of the environment, considering each of the signals that are bound. The first case of Can_ϱ corresponds to the first signal case in Can; the second case in Can_ϱ corresponds to the second signal case, and the last case in Can_ϱ corresponds to the situation where there are no more signals bound in $\boldsymbol{\theta}$ (and the remainder of the $\boldsymbol{\theta}_l$ can be dropped as it contains only information about **s** and **x** variables, which Can does not need, as it does not reason about host expressions).

3.3 Reincarnation, Schizophrenia, and Correct Binding

The signal form seems to be something close to a variable binding form, familiar from conventional λ-calculus based programming languages. It is, however, not the same and a significant source of subtlety in Esterel. The Esterel community has explored these issues in great detail and in this section, we try to bring across the basic points and then explain how our calculus handles them.

```
(loop
 (signal SZ
   (seq pause
        (emit SZ))))
```

Fig. 17: A Schizophrenic Loop

The two central issues are the phenomena of schizophrenic and reincarnated signals. To understand them, first recall the central tenant of Esterel signals: every signal must have exactly one value in a given instant. Now, consider the example program in figure 17. During the first instant of execution the signal SZ will be absent, as the program pauses before emitting it. In the next instant we pick up where we left off. The first thing that happens is that we emit SZ. Then the loop body restarts. Because we have re-entered the loop body and encountered the signal expression afresh, the SZ should now be absent. But this means that the signal SZ has two different values in a single instant!

In the literature, signals which are duplicated by a loop body in within one instant are called *reincarnated*. If a reincarnated signal obtains different values in each of its incarnations, it is called *schizophrenic*. Schizophrenic signals, however, merely appear to violate the single-value-per-instant rule. Because instantaneous loops are banned, the number of times a loop body can be entered is bounded. This means that the number of reincarnations of any signal is also bounded. Therefore we consider each incarnation to, in fact, be a separate signal, removing the apparent violation.

[3] This exponential behavior affected our testing of the semantics against each other; see section 6 and section 7 for more.

This resolution shows up directly in Esterel compilers and circuit semantics. Naive treatment of schizophrenic signals can cause unstable loops in the corresponding circuit, breaking the guarantee that all constructive programs translate to stable circuits. Therefore, many Esterel compilers duplicate parts of loop bodies with schizophrenic signals to remove the apparent violation of the single-value-per-instant rule, avoiding cross-loop cycles (Berry 2002; Potop-Butucaru et al. 2007; Schneider and Wenz 2001). In short, each incarnation of a signal gets a separate wire.

Esterel semantics such as the Constructive Operation Semantics (COS) (Potop-Butucaru 2002) and the Constructive Behavioral Semantics (CBS) (Berry 2002) take a different approach, handling such signals by carefully arranging to "forget" a schizophrenic signal's first value when the second one is needed.

Our semantics takes an approach inspired by the circuit perspective, meaning we do not treat signals in a conventional way. More precisely, we do not assume the variable convention (Barendregt 1984), nor do we include an α rule. Indeed, we think of signals as if they name wires.

This perspective means that schizophrenic and reincarnated signals are, at first glance, handled very simply. We just duplicate the bodies of loops in the [**loop**] rule, so each signal will end up in a different ϱ, potentially bound to a different value—akin to the strategy that circuit semantics employ. This approach, however, does raise a significant concern: what happens if the [**merge**] rule moves ϱ expressions in such a way that causes incorrect capture? Our calculus avoids this problem by working only with programs that have *correct binding*, as captured by the $\vdash_{\text{CB}}$ judgment form in figure 18. (The $\vdash_{\text{CB}}$ judgment also ensures that sequential variables are used in at most one branch of any `par`, which is not related to the concerns of schizophrenia, but does ensure determinism and is convenient to include here.)

To understand the correct binding judgment, first look at the `seq` rule. It says that the bound signals of the first sub-expression must be distinct from the free signals of the second. Since the [**merge**] rule moves binders based on the definition of **E** (in figure 13), it can move a ϱ out from the first sub-expression only. Thus, in order to preserve the binding structure of the expression as we reduce, we need only make sure that a ϱ that moves out of the first sub-expression of a `seq` does not capture a signal in the second sub-expression, which is precisely what the premise avoids.

The other rules all generally follow this reasoning process for their premises. The `suspend` rule's premise follows exactly that reasoning, as binder may be lifted out past the **S**. The `par` rule's second and third premises also follow exactly the same reasoning. The first premise of `par` is necessary to avoid the situation where the same signal is bound in both branches and then is lifted out from both. The fourth premise ensures that sequential variables are used properly.

The `loop` rule must ensure that the bound and free signals of its subexpression do not overlap, as it reduces by duplicating its first subexpression into a $\overline{\texttt{loop}}$, which acts like a `seq` expression (so the intuition for `seq` applies, but with both subexpressions being the same one). Similarly, because ($\overline{\texttt{loop}}$ **p** **q**) behaves like (`seq` **p** (`loop` **q**)), the premises of its rule are just the premises of the `seq` and `loop` rules, combined.

Theorem 3.1.

$$\forall\, \mathbf{p}, \mathbf{q}, \mathbf{C}.\ \ \vdash_{\text{CB}} \mathbf{C}[\mathbf{p}] \ \Rightarrow\ \mathbf{p} \rightharpoonup \mathbf{q} \ \Rightarrow\ \vdash_{\text{CB}} \mathbf{C}[\mathbf{q}]$$

This theorem states that, no matter which context an expression reduces in (with **C** as given in figure 19), if the expression had correct binding before reduction, it does afterwards, too. The proof is given as $\longrightarrow_1$`-preserve-CB` in the Agda code in the supplementary material.

It should also be noted that any Esterel program that uses its sequential variables correctly either already has correct binding or can be renamed into one that has correct binding (introducing new wires, of course) before reducing the program. Thus, the restriction that our calculus handles only

$$\boxed{\vdash_{\text{CB}}\ \mathbf{p}}$$

BV : **p** ⟶ (Setof «var»)
FV : **p** ⟶ (Setof «var»)

Computes the bound and free variables, respectively. The variables include signals, shared variables and sequential variables.

$$\frac{}{\vdash_{\text{CB}}\ \texttt{nothing}} \qquad \frac{}{\vdash_{\text{CB}}\ \texttt{pause}} \qquad \frac{}{\vdash_{\text{CB}}\ (\texttt{emit}\ \mathbf{S})}$$

$$\frac{\vdash_{\text{CB}}\ \mathbf{p}}{\vdash_{\text{CB}}\ (\texttt{signal}\ \mathbf{S}\ \mathbf{p})} \qquad \frac{\vdash_{\text{CB}}\ \mathbf{p} \qquad \vdash_{\text{CB}}\ \mathbf{q}}{\vdash_{\text{CB}}\ (\texttt{present}\ \mathbf{S}\ \mathbf{p}\ \mathbf{q})} \qquad \frac{\vdash_{\text{CB}}\ \mathbf{p}}{\vdash_{\text{CB}}\ (\varrho\ \boldsymbol{\theta}.\ \mathbf{p})}$$

$$\frac{\vdash_{\text{CB}}\ \mathbf{p}}{\vdash_{\text{CB}}\ (\texttt{var}\ \mathbf{x} \coloneqq \mathbf{e}\ \mathbf{p})} \qquad \frac{}{\vdash_{\text{CB}}\ (\coloneqq \mathbf{x}\ \mathbf{e})} \qquad \frac{\vdash_{\text{CB}}\ \mathbf{p} \qquad \vdash_{\text{CB}}\ \mathbf{q}}{\vdash_{\text{CB}}\ (\texttt{if}\ \mathbf{x}\ \mathbf{p}\ \mathbf{q})}$$

$$\frac{\text{BV}[\![\mathbf{p}]\!] \cap \text{FV}[\![\mathbf{q}]\!] = \emptyset \qquad \vdash_{\text{CB}}\ \mathbf{p} \qquad \vdash_{\text{CB}}\ \mathbf{q}}{\vdash_{\text{CB}}\ (\texttt{seq}\ \mathbf{p}\ \mathbf{q})} \qquad \frac{\{\,\mathbf{S}\,\} \cap \text{BV}[\![\mathbf{p}]\!] = \emptyset \qquad \vdash_{\text{CB}}\ \mathbf{p}}{\vdash_{\text{CB}}\ (\texttt{suspend}\ \mathbf{p}\ \mathbf{S})}$$

$$\frac{\begin{array}{ccc}\text{BV}[\![\mathbf{p}]\!] \cap \text{BV}[\![\mathbf{q}]\!] = \emptyset & \text{FV}[\![\mathbf{p}]\!] \cap \text{BV}[\![\mathbf{q}]\!] = \emptyset & \text{BV}[\![\mathbf{p}]\!] \cap \text{FV}[\![\mathbf{q}]\!] = \emptyset \\ \multicolumn{2}{c}{\{\,\mathbf{x} \mid \mathbf{x} \in \text{FV}[\![\mathbf{p}]\!]\,\} \cap \{\,\mathbf{x} \mid \mathbf{x} \in \text{FV}[\![\mathbf{q}]\!]\,\} = \emptyset} & \vdash_{\text{CB}}\ \mathbf{p} \qquad \vdash_{\text{CB}}\ \mathbf{q}\end{array}}{\vdash_{\text{CB}}\ (\texttt{par}\ \mathbf{p}\ \mathbf{q})}$$

$$\frac{\text{BV}[\![\mathbf{p}]\!] \cap \text{FV}[\![\mathbf{q}]\!] = \emptyset \qquad \text{BV}[\![\mathbf{q}]\!] \cap \text{FV}[\![\mathbf{q}]\!] = \emptyset \qquad \vdash_{\text{CB}}\ \mathbf{p} \qquad \vdash_{\text{CB}}\ \mathbf{q}}{\vdash_{\text{CB}}\ (\overline{\texttt{loop}}\ \mathbf{p}\ \mathbf{q})} \qquad \frac{\text{BV}[\![\mathbf{p}]\!] \cap \text{FV}[\![\mathbf{p}]\!] = \emptyset \qquad \vdash_{\text{CB}}\ \mathbf{p}}{\vdash_{\text{CB}}\ (\texttt{loop}\ \mathbf{p})}$$

$$\frac{\vdash_{\text{CB}}\ \mathbf{p}}{\vdash_{\text{CB}}\ (\texttt{trap}\ \mathbf{p})} \qquad \frac{}{\vdash_{\text{CB}}\ (\texttt{exit}\ \texttt{n})} \qquad \frac{\vdash_{\text{CB}}\ \mathbf{p}}{\vdash_{\text{CB}}\ (\texttt{shared}\ \mathbf{s} \coloneqq \mathbf{e}\ \mathbf{p})} \qquad \frac{}{\vdash_{\text{CB}}\ (\mathrel{+}= \mathbf{s}\ \mathbf{e})}$$

Fig. 18. Correct Binding

programs with correct binding is not severe, as any already correct program can be transformed into one which is well behaved in our calculus.

3.4 Evaluating Programs

Now that we have established the correct binding invariant and defined the primitive notions of reduction, we can turn to the definition of the evaluator. It is shown on the top-left of figure 19. It accepts a program and an initial environment (that captures what the host language sets the input signals to), and it returns the signals that were emitted at the end of the instant. The output of the evaluator ignores shared variables. However, values of shared variables can be indirectly returned by introducing new signals whose presence depends on the values of shared variables.

The $\equiv_\text{e}$ relation is the symmetric, transitive, reflexive closure of the $\longrightarrow$ relation, which is the compatible closure of the $\rightharpoonup$ reduction relation. The symmetric case has an additional premise $\vdash_{\text{CB}}\ \mathbf{p}$ to ensure that all of the intermediate terms used in $\equiv_\text{e}$ have correct binding.

The definition of `Eval` is written using a notation that assumes the central result of this paper, namely that `Eval` is a (partial) function:

$$\frac{(\rho\ \boldsymbol{\theta}.\ \mathbf{p}) \equiv\!\mathrm{e}\ (\rho\ \boldsymbol{\theta}^{c}.\ \mathbf{done})}{\texttt{Eval}(\mathbf{p}\ ,\ \boldsymbol{\theta}) = \{\ \mathsf{S} \in \mathrm{dom}(\boldsymbol{\theta}^{c})\ \mid \boldsymbol{\theta}^{c}(\mathbf{S}) = \texttt{present}\ \}}$$

$$\frac{\mathbf{p} \longrightarrow \mathbf{q}}{\mathbf{p} \equiv\!\mathrm{e}\ \ \mathbf{q}} \qquad \frac{\mathbf{p} \equiv\!\mathrm{e}\ \ \mathbf{q} \qquad \vdash_{\mathrm{CB}}\ \mathbf{p}}{\mathbf{q} \equiv\!\mathrm{e}\ \ \mathbf{p}}$$

$$\frac{}{\mathbf{p} \equiv\!\mathrm{e}\ \ \mathbf{p}} \qquad \frac{\mathbf{p}_{1} \equiv\!\mathrm{e}\ \ \mathbf{p}_{2} \qquad \mathbf{p}_{2} \equiv\!\mathrm{e}\ \ \mathbf{p}_{3}}{\mathbf{p}_{1} \equiv\!\mathrm{e}\ \ \mathbf{p}_{3}}$$

$$\frac{\mathbf{p} \rightharpoonup \mathbf{q}}{\mathbf{C}[\mathbf{p}] \longrightarrow \mathbf{C}[\mathbf{q}]}$$

$$\frac{\mathbf{p}_{1} \longrightarrow \mathbf{p}_{2} \qquad \mathbf{p}_{2} \longrightarrow^{*} \mathbf{p}_{3}}{\mathbf{p}_{1} \longrightarrow^{*} \mathbf{p}_{3}} \qquad \frac{}{\mathbf{p} \longrightarrow^{*} \mathbf{p}}$$

$$\begin{array}{rl}
\mathbf{C} ::= & (\texttt{signal}\ \mathbf{S}\ \mathbf{C}) \\
\mid & (\texttt{seq}\ \mathbf{C}\ \mathbf{q}) \\
\mid & (\texttt{seq}\ \mathbf{p}\ \mathbf{C}) \\
\mid & (\overline{\texttt{loop}}\ \mathbf{C}\ \mathbf{q}) \\
\mid & (\overline{\texttt{loop}}\ \mathbf{p}\ \mathbf{C}) \\
\mid & (\texttt{present}\ \mathbf{S}\ \mathbf{C}\ \mathbf{q}) \\
\mid & (\texttt{present}\ \mathbf{S}\ \mathbf{p}\ \mathbf{C}) \\
\mid & (\texttt{par}\ \mathbf{C}\ \mathbf{q}) \\
\mid & (\texttt{par}\ \mathbf{p}\ \mathbf{C}) \\
\mid & (\texttt{loop}\ \mathbf{C}) \\
\mid & (\texttt{suspend}\ \mathbf{C}\ \mathbf{S}) \\
\mid & (\texttt{trap}\ \mathbf{C}) \\
\mid & (\texttt{shared}\ \mathbf{s} \coloneqq \mathbf{e}\ \mathbf{C}) \\
\mid & (\texttt{var}\ \mathbf{x} \coloneqq \mathbf{e}\ \mathbf{C}) \\
\mid & (\texttt{if}\ \mathbf{x}\ \mathbf{C}\ \mathbf{q}) \\
\mid & (\texttt{if}\ \mathbf{x}\ \mathbf{p}\ \mathbf{C}) \\
\mid & (\rho\ \boldsymbol{\theta}.\ \mathbf{C}) \\
\mid & [\,]
\end{array}$$

Fig. 19. Eval

$$\begin{array}{lcl}
\looparrowright : \mathbf{complete} \longrightarrow \mathbf{p} & & \\
\looparrowright(\rho\ \boldsymbol{\theta}^{c}.\ \mathbf{p}) & = & (\rho\ \lfloor\boldsymbol{\theta}^{c}\rfloor.\ \looparrowright\mathbf{p}) \\
\looparrowright\texttt{pause} & = & \texttt{nothing} \\
\looparrowright\texttt{nothing} & = & \texttt{nothing} \\
\looparrowright(\overline{\texttt{loop}}\ \mathbf{p}\ \mathbf{q}) & = & (\texttt{seq}\ \looparrowright\mathbf{p}\ (\texttt{loop}\ \mathbf{q})) \\
\looparrowright(\texttt{seq}\ \mathbf{p}\ \mathbf{q}) & = & (\texttt{seq}\ \looparrowright\mathbf{p}\ \mathbf{q}) \\
\looparrowright(\texttt{par}\ \mathbf{p}\ \mathbf{q}) & = & (\texttt{par}\ \looparrowright\mathbf{p}\ \looparrowright\mathbf{q}) \\
\looparrowright(\texttt{suspend}\ \mathbf{p}\ \mathbf{S}) & = & (\texttt{suspend}\ (\texttt{seq}\ (\texttt{present}\ \mathbf{S}\ \texttt{pause}\ \texttt{nothing})\ \looparrowright\mathbf{p})\ \mathbf{S}) \\
\looparrowright(\texttt{trap}\ \mathbf{p}) & = & (\texttt{trap}\ \looparrowright\mathbf{p}) \\
\looparrowright(\texttt{exit}\ \mathrm{n}) & = & (\texttt{exit}\ \mathrm{n})
\end{array}$$

Fig. 20. Next Instant

Theorem 3.2.

$$\begin{array}{l}
\forall\ \mathbb{S}_{1}\ ,\ \mathbb{S}_{2}\ ,\ \boldsymbol{\theta}\ ,\ \mathbf{p}. \\
\vdash_{\mathrm{CB}}\ (\rho\ \boldsymbol{\theta}.\ \mathbf{p})\ \Rightarrow \\
\texttt{Eval}(\mathbf{p}\ ,\ \boldsymbol{\theta}) = \mathbb{S}_{1}\ \Rightarrow \\
\texttt{Eval}(\mathbf{p}\ ,\ \boldsymbol{\theta}) = \mathbb{S}_{2}\ \Rightarrow \\
\mathbb{S}_{1} = \mathbb{S}_{2}
\end{array}$$

The above theorem states that if $\mathbb{S}_1$ and $\mathbb{S}_2$ are both sets of signals satisfying the `Eval` judgment in figure 19, then $\mathbb{S}_1$ and $\mathbb{S}_2$ must be equal. The proof is given as `eval≡`$_e$`-consistent` in the Agda code in the supplementary material.

This theorem is a corollary of the consistency of ≡e, which states that if two expressions are ≡e, then there is an expression that both reduce to, under the transitive reflexive closure of the compatible closure of the reduction relation:

Theorem 3.3.

$$\forall\, \mathbf{p}\,,\mathbf{q}.\ \vdash_{\text{CB}}\ \mathbf{p}\ \Rightarrow\ \mathbf{p} \equiv\!\text{e}\ \mathbf{q}\ \Rightarrow$$
$$\exists\, \mathbf{r}.\ \mathbf{p} \longrightarrow^{*} \mathbf{r} \wedge \mathbf{q} \longrightarrow^{*} \mathbf{r}$$

The proof is given as `≡`$_e$`-consistent`, and it follows from the confluence of reduction.

Our semantics supports multiple instants via a transformation that prepares a complete expression for the next instant, $\looparrowright$, shown in figure 20. It makes four modifications to the expression. First, it resets all signals to unknown via $\lfloor \boldsymbol{\theta}^{c} \rfloor$ (also defined in figure 13). Second, it replaces the `pause` expressions where the program stopped with `nothing`. Third, it replaces each $\overline{\texttt{loop}}$ expression with a `loop` and `seq`. Finally, it adds a `present` expression to `suspend` expressions that have paused. The `present` serves to conditionally pause the body of the `suspend` in the next instant. The result is an expression suitable for use with `Eval` in the next instant.

4 ON CONSTRUCTIVENESS

The properties of logical correctness and constructiveness are key for any correct semantics of Esterel. For examples of these properties we refer the reader to section 2.4. We follow the definition of constructiveness given by the constructive operational semantics (COS) as referenced by Berry (2002) and described by Potop-Butucaru (2002). Constructiveness is defined by the COS evaluator: non-constructive programs reduce to stuck terms (that are not **complete**).

```
(signal S1
  (present S1
           (signal S2
             (seq (emit S2)
                  (present S2
                           nothing
                           (emit S1))))
           nothing))
```

Fig. 21: A Non-constructive Program

In our semantics, for many expressions, this is also the case. But, it is not the case for all of them because reductions that occur in arbitrary program contexts sometimes give `Can` more information than it “should” have (more precisely, more information that it would get by running the program directly). This extra information means that reductions in our calculus can transform a non-constructive program into a constructive one that can still reduce.

For an example, consider the expression in figure 21. If we restrict our attention to the outside part of the term (the way that the COS semantics does), it reduces only by replacing the outer `signal` form with a ϱ expression. At that point, the expression might appear to be stuck because `Can` is unable to prove that `S1` is not emitted (and thus the [**absence**] rule does not apply) and the `present` expression does not reduce (because `S1` is `unknown`).

There are reductions that can occur, however, at the inner `signal` expression, revealing information to `Can`, and enabling it to determine that `S1` is `absent`.

Specifically, the calculus can reduce in this context:

THEOREM 5.1.
```
∀ S₁ , S₂ , p.
⊢CB p ⇒
(signal S₁
  (signal S₂
    p)) ≡e
(signal S₂
  (signal S₁
    p))
```

THEOREM 5.2.
```
∀ S , p , q.
⊢CB p ⇒
(signal S
  (seq (emit S)
       (present
        S
        p
        q))) ≡e
(signal S
  (seq (emit S)
       p))
```

THEOREM 5.3.
```
∀ S , p , q.  ⊢CB q ⇒
(∀ status. S ∉ (Can p { S ↦ status }).S) ⇒
(∀ status. S ∉ (Can q { S ↦ status }).S) ⇒
(signal S
  (present S
           p
           q)) ≡e
(signal S
  q)
```

THEOREM 5.4.
```
∀ S , p , q.
⊢CB (par p q) ⇒
(signal S
  (par (seq
         (emit S)
         p)
       q)) ≡e
(signal S
  (seq (emit S)
       (par p
            q)))
```

THEOREM 5.5.
```
∀ n , p , q.
⊢CB p ⇒
q ∈ done ⇒
p ≡e q ⇒
(trap
 (par (exit n+1)
      p)) ≡e
(par (exit n)
     (trap p))
```

THEOREM 5.6.
```
∀ p , q , S.
⊢CB (seq (signal S p)
         q)          ⇒
(ρ {}. (seq (signal S p)
            q)) ≡e
(ρ {}. (signal S
         (seq p q)))
```

Fig. 23. Equivalences Provable in our Calculus

```
(ρ { S1 ↦ unknown }.
  (present S1
           []
           nothing))
```

and thus it can turn the signal form into a ρ and perform the emit, resulting in the expression in figure 22. Being able to reduce in that context is effectively "peeking" ahead into the future non-constructively.

Once those reductions happen, Can is able to determine that S1 cannot be emitted and now the [**absence**] rule can fire, eventually reducing the original expression to

```
(ρ { S1 ↦ absent }. nothing)
```

.

```
(ρ { S1 ↦ unknown }.
  (present
   S1
   (ρ { S2 ↦ present }.
     (seq nothing
          (present S2
                   nothing
                   (emit S1))))
   nothing))
```

Fig. 22: An expression equivalent to the expression in figure 21

In sum, our calculus equates some non-constructive programs to constructive programs with the same logical behavior. Although we are not satisfied with this aspect of our calculus and believe that it deserves further study, such a relaxation of constructiveness is not unprecedented (Tardieu 2007).

5 WHAT THE CALCULUS CAN AND CANNOT PROVE

Our semantics lends itself to establishing equivalences between program fragments because any two expressions that are `≡e` to each other always produce the same result in the evaluator:

Theorem 5.7.
$\forall\ \mathbf{p}, \boldsymbol{\theta}_1, \mathbb{S}_1, \mathbf{q}, \boldsymbol{\theta}_2, \mathbb{S}_2.$
$\vdash_{\text{CB}}\ (\varrho\ \boldsymbol{\theta}_1.\ \mathbf{p}) \Rightarrow$
$(\varrho\ \boldsymbol{\theta}_1.\ \mathbf{p}) \equiv\text{e}\ (\varrho\ \boldsymbol{\theta}_2.\ \mathbf{q}) \Rightarrow$
$\texttt{Eval}(\mathbf{p}, \boldsymbol{\theta}_1) = \mathbb{S}_1 \Rightarrow$
$\texttt{Eval}(\mathbf{q}, \boldsymbol{\theta}_2) = \mathbb{S}_2 \Rightarrow$
$\mathbb{S}_1 = \mathbb{S}_2$

This theorem is a straightforward consequence of `≡e` being consistent; the proof is given as $\equiv_e$`=>eval` in the Agda code in the supplementary material.

The remainder of this section explores various equivalences (shown in figure 23) as well as some limitations of the calculus. The proofs of the equivalences are all given in `calculus-examples.agda` in the supplementary material.

The first example, theorem 5.1, shows that we can rearrange signal forms. This example works well in our calculus. It requires only that the body expression has correct binding, allowing us to rearrange adjacent `signal` forms arbitrarily.

Next, theorem 5.2 shows that if an `emit` is followed by a `present`, the `present` can always be replaced by the taken branch. This example exposes a first limitation of the calculus. Although it is still true, our calculus cannot prove this equivalence without the `signal` form being visible in an evaluation context surrounding the `seq` form.

In a dual to theorem 5.2, theorem 5.3 shows that if we know that neither branch of the `present` expression can emit **S**, we can replace the `present` form with its second subexpression.

Theorem 5.4 lets us lift a `seq` expression that starts with an `emit` out of a `par` branch. Intuitively, this equivalence is a consequence of Esterel's deterministic parallelism. Because `emit` does not block, we can do it in parallel to **q** or before **q** starts, whichever is more convenient.

When a `trap` is outside a `par`, our calculus allows us to push the `trap` inside, in some situations. Theorem 5.5 is one such. This calculation requires **p** to be equivalent to some **done** expression **q**, but that is a weakness of our calculus. In fact, the two expressions are observably equivalent without any assumptions.

Theorem 5.6 further generalizes Theorem 5.1 to rearrange binding forms across other expressions. In this example, the `signal` form is pulled out of the `seq` expression. In general, these two expressions are observably equivalent even without the ϱ expression outside. Our calculus cannot prove it, however, because the calculus needs an outer ϱ expression in order to perform a [**merge**] in the middle of the proof.

We explored a calculus that includes a "lifting" rule that allows us to move a ϱ term up and down in an evaluation context. This rule makes it difficult to establish confluence of the calculus, however, as the would-be lifting rule and the [**merge**] rule interact with each other in complex ways. In particular, our evaluation contexts do not have unique decomposition, due to `par`. Accordingly, a use of the lifting rule from one side of a `par` expression can block a use of the [**merge**] rule from the other side. We conjecture that a lifting rule would be confluent, but have not proven it. If we did have such a rule, then we believe we would be able to prove theorem 5.6 without the need for an enclosing empty ϱ expression and even be able to relax one of the assumptions of theorem 5.5, assuming only that **q** is **complete**.

Our calculus also cannot reason effectively with expressions that span multiple instants. For example, the expression (`seq` (`loop pause`) **q**) is equivalent to (`loop pause`), but our calculus cannot prove it. Similarly, a common pattern is to emit a signal and pause in a loop, and also to run that loop in parallel with some code that looks at the signal. Our calculus would not be able to propagate the signal's presence because of the `pause`.

Another deep lack in our calculus is the ability to reason about input signals. In order for our calculus to work with a signal, it must be bound by `signal` so knowledge about it can be manipulated via the rules for ϱ expressions. Input signals, in contrast, might or might not be set by the environment and our calculus cannot perform the required conditional reasoning.

6 TESTING

As we are developing a new semantics for Esterel and Esterel is a well-established language, a natural concern is whether our semantics captures Esterel or some other, subtly different language. In order to mitigate this concern, we tested our semantics against two Esterel implementations: Esterel v5 (Berry 2000) and Hiphop.js (Berry et al. 2011), as well as an executable version of Potop-Butucaru et al. (2007)'s COS semantics. Perhaps unsurprisingly, we also discovered bugs in both of the implementations during this process (as random testing can be extremely effective (Yang et al. 2011)). The remainder of this section describes the testing process and the bugs discovered.

6.1 Testing for conformance

In order to test our model against the existing semantics and implementations, we had to build some software libraries:

- **Redex COS model** We built a model of the COS semantics in Redex (Felleisen et al. 2009). The semantics is a faithful model of the COS semantics because it is a rule-for-rule translation of the COS semantics; aside from a few syntactic differences (notably more parentheses), it mirrors Potop-Butucaru et al. (2007)'s model exactly, enabling us to simulate the behavior of the semantics on any example program.
- **Redex calculus model** Our calculus is implemented in Redex; the rules shown in all of the figures are generated automatically from the Redex source code, and the Redex model also enables us to explore the reduction of any example program.
- **Agda/Redex bridge** We built a library that can translate the reduction sequences generated by Redex into proofs in Agda, ensuring that the specific, concrete terms which reduce in Redex also reduce the same way in Agda. This process accepts a specific term and a reduction sequence. It produces a proof, which then is submitted to the Agda compiler for verification.
- **Redex/Hiphop.js bridge** We built a library that can translate Redex expressions into Hiphop.js programs and then evaluate them. We also built a translator for a subset of Hiphop.js programs that can translate them into Redex so they can be checked against the calculus and the COS model. This translator does not accept all Hiphop.js programs, because Hiphop.js programs embed JavaScript code and our model cannot evaluate the JavaScript.
- **Redex/Esterel v5 bridge** We also built a translator that produces Esterel v5 programs from Redex terms in the COS model and in our calculus.

Using these libraries we can test all four implementations of Esterel (the COS semantics, the Esterel v5 compiler, Hiphop.js, and our calculus) against each other.

There is one subtle point about testing our calculus. Because it is a calculus, we need an algorithm that can determine which of the many possible reductions we should take in order to find an effective path to a **complete** state (if one exists). To do this, we identified a subset of the possible reductions in a way that acts like an standard reduction, guaranteeing that we find a **complete** state if the program is constructive, and that reduction gets stuck if it is not. This reduction relation is given in figure 26 with some supplementary definitions given in figure 27; it is explained briefly in section 7. We use this reduction relation to guide the calculus, verifying that each step in the standard reduction is also possible in the calculus. There are no proofs about our standard reduction but we use it only to test our calculus against other implementations, as described in this section.

These libraries give us the ability to, given an Esterel program, determine if it produces the same signals across multiple instants. But we also need a source of Esterel programs to test. For that purpose, we used two approaches.

First, we took the Hiphop.js test suite, which consists of 130 Hiphop.js programs. Of those, four use `pre`, a construct that is not in Kernel Esterel, and were excluded from our tests. An additional 84 use JavaScript in some non-trivial way, and therefore could also not be run in our model. Our calculus produces the same results on the remaining 42 program as Hiphop.js.

The translation of the Hiphop.js tests into our model produces programs that have a large number of signals, which causes problems for the process that finds reductions in the calculus. In short, the problem is that the exponential behavior in `Can` triggers significant performance problems in the calculus, enough so that running these tests appears not to be feasible. To mitigate this issue, we disabled the exponential behavior of `Can` (by eliminating the first case in $\texttt{Can}_{\rho}$ and the first `signal` case) and then were able to use the standard reduction to simulate the calculus successfully, and got the same results as the Hiphop.js implementation.

Second, we used Redex's capability to generate random Esterel expressions and run them in all of the implementations to see if they agree. We have discovered (and fixed) errors in our calculus using this method, and we currently have no known bugs. We have run over 1,800,000 random tests and they still do periodically find counterexamples, but they find only known bugs in the implementations.

This random testing process proved invaluable in debugging the calculus, catching several errors that cannot be found via the proofs in Agda. For example, late in the development process, the random tester found that an old version of the [**shared**] rule was incorrect. The old version initialized the shared variables status to `new`, but the COS specifies that the initial status is `old`. This bug does not invalidate any of the theorems in Agda, but it does violate the property that our calculus and the other implementations agree. That is, the properties we can effectively check via random testing are stronger than those we can check via proof (in practice).

6.2 Bugs Discovered

During the process of validating our calculus, we discovered four bugs in Hiphop.js and one bug in the Esterel v5 compiler. All of the bugs have been confirmed by the authors of the systems. All but one of the Hiphop.js bugs have been fixed.

```
(shared s-outer := e₀
  (seq (shared s-inner := e_outer
          nothing)
        (+= s-outer e₁)))
```

Fig. 24: A Bug Found in the Esterel v5 Compiler

The bug in Esterel v5 is exhibited by a translation of the program in figure 24, where $\mathbf{e}_0$ evaluates to 0, $\mathbf{e}_1$ evaluates to 1, and $\mathbf{e}_{outer}$ refers to `s-outer`.

This program is non-constructive and it gets stuck in our both our calculus and the COS semantics; it cannot reduce the inner `shared` because the initialization of the signal depends on `s-outer`, but `s-outer` cannot be read because there is a write pending. The Esterel v5 compiler runs the program, incorrectly setting `s-inner` to 0.

This program also demonstrates one of the bugs we found in Hiphop.js. Of the other three bugs, one of them was an internal error, crashing Hiphop.js on this program (`trap` (`suspend` (`exit` 0) `S1`)).

The next bug was triggered by this expression (`suspend nothing S1`), and produced an error in terms of the undefined value from Hiphop.js's host language, Javascript.

The final bug is exhibited by the program in figure 25. Both in our calculus and in the COS semantics, the `Can` function can determine that `S-inner` cannot be emitted, and that therefore `S-outer` cannot be emitted. Therefore the program is constructive, both signals are `absent`, and the program reduces to `nothing`. However this program appeared to be non-constructive to Hiphop.js.

```
(signal S-outer
  (signal S-inner
    (seq
      (present S-outer nothing nothing)
      (present S-inner
               (emit S-outer)
               nothing))))
```

Fig. 25: A Bug Found in Hiphop.js

7 STANDARD REDUCTION

Our standard reduction exists only in Redex (not in Agda, unlike the rest of the semantics). We use it to help with our testing process, as described in section 6.

Figure 26 shows the reduction rules. There are four differences between the rules of the calculus and the rules of the standard reduction. First, expressions reduce only if they have an outer ρ. Second, the [**absence**] and [**readyness**] rules set as many signals or variables as they can in a single step. Third, the [**absence**] and [**readyness**] rules use Can_ρ in the calculus and `Can` in the standard reduction. In the standard reduction, the extra analysis that Can_ρ performs is not necessary. Finally, the rules are oriented so that at most one applies at each step. There are two pieces to this orientation: restricting the context in which the rules may apply and restricting the [**absence**] and [**readyness**] rules so they apply only when no other rule applies.

To understand how the rules are oriented, consider the [**absence**] and [**readyness**] rules. They require the body to be either done or blocked, where blocked is given in figure 27. It captures when an expression cannot reduce because it needs the value of a signal or shared variable that is not known or `ready`.

The context restriction is captured by the $\boldsymbol{\theta} \vdash \mathbf{E}$ det judgment. The judgment is designed to restrict the choice of sub-expression in `par` terms so only one side is considered for reduction. There are three `par` rules: one that always allows reductions in the left-hand side, and two others that allow reduction on the right, but only when the left is either done or blocked.

Otherwise, the reduction rules in the standard reduction parallel those in calculus. Of course, we do not take the compatible closure; instead we just reduce in evaluation contexts.

There is one subtle point of this standard reduction: it is not the standard reduction relation for our calculus, but rather the one for a slightly smaller calculus. In particular, it does not bypass constructiveness, unlike the calculus (as discussed in section 4). It reaches a stuck state instead.

8 RELATED WORK

Three decades of work on Esterel have resulted in a diversity of semantic models. A fundamental difference of our semantics is that ours is a calculus rather than merely a reduction system—it has an equational theory. In general, the nature of our semantics makes it better for discussing transformations to Esterel programs and, more specifically, it lets us prove equivalences in arbitrary contexts, like those discussed in section 5. By the same token, our semantics is not particularly well-suited to implementing Esterel.

Prior semantics of Esterel can be broadly categorized as follows:

- Macrostep operational semantics compute the result of an instant in big-step style, where evaluation relates the state of a program at the beginning of an instant to the state at the end.

signals

$$(\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\texttt{signal}\ \mathbf{S}\ \mathbf{p})]) \longrightarrow (\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\rho\ \{\ \mathbf{S} \mapsto \texttt{unknown}\ \}.\ \mathbf{p})])\ \text{ where } \boldsymbol{\theta} \vdash \mathbf{E}\ \text{det} \qquad \textbf{[signal]}$$

$$(\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\texttt{emit}\ \mathbf{S})]) \longrightarrow (\rho\ (\boldsymbol{\theta} \leftarrow \{\ \mathbf{S} \mapsto \texttt{present}\ \}).\ \mathbf{E}[\texttt{nothing}]) \qquad \textbf{[emit]}$$
$$\text{where } \boldsymbol{\theta} \vdash \mathbf{E}\ \text{det},\ \boldsymbol{\theta}(\mathbf{S}) \in \{\ \texttt{present}\ ,\ \texttt{unknown}\ \}$$

$$(\rho\ \boldsymbol{\theta}.\ \mathbf{p}) \longrightarrow (\rho\ \text{set-absent}(\boldsymbol{\theta}\ ,\ \mathbb{S}).\ \mathbf{p}) \qquad \textbf{[absence]}$$
$$\text{where } (\boldsymbol{\theta} \vdash \mathbf{p}\ \text{blocked or}\ \mathbf{p} \in \mathbf{done}),$$
$$\mathbb{S} = \{\ \mathbf{S} \in \text{dom}(\boldsymbol{\theta}\ )\ |\ \boldsymbol{\theta}^{c}(\mathbf{S}) = \texttt{unknown}\ \} \setminus \texttt{Can}[\![\mathbf{p},\ \boldsymbol{\theta}]\!].\text{S},\ \mathbb{S} \neq \emptyset$$

$$(\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\texttt{present}\ \mathbf{S}\ \mathbf{p}\ \mathbf{q})]) \longrightarrow (\rho\ \boldsymbol{\theta}.\ \mathbf{E}[\mathbf{p}])\ \text{ where } \boldsymbol{\theta} \vdash \mathbf{E}\ \text{det},\ \boldsymbol{\theta}(\mathbf{S}) = \texttt{present} \qquad \textbf{[is-present]}$$

$$(\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\texttt{present}\ \mathbf{S}\ \mathbf{p}\ \mathbf{q})]) \longrightarrow (\rho\ \boldsymbol{\theta}.\ \mathbf{E}[\mathbf{q}])\ \text{ where } \boldsymbol{\theta} \vdash \mathbf{E}\ \text{det},\ \boldsymbol{\theta}(\mathbf{S}) = \texttt{absent} \qquad \textbf{[is-absent]}$$

shared variables

$$(\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\texttt{shared}\ \mathbf{s} \coloneqq \mathbf{e}\ \mathbf{p})]) \longrightarrow (\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\rho\ \{\ \mathbf{s} \mapsto \langle \text{n}\ ,\ \texttt{old}\rangle\ \}.\ \mathbf{p})]) \qquad \textbf{[shared]}$$
$$\text{where } \boldsymbol{\theta} \vdash \mathbf{E}\ \text{det},\ \text{FV}(\mathbf{e}) \subset \text{dom}(\boldsymbol{\theta}),\ \forall\ \text{s} \in \text{FV}(\mathbf{e}).\ \boldsymbol{\theta}(\text{s}) = \langle\_\ ,\ \texttt{ready}\rangle,\ \text{n} = \mathcal{E}val[\![\mathbf{e}\ ,\ \boldsymbol{\theta}]\!]$$

$$(\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\texttt{+=}\ \mathbf{s}\ \mathbf{e})]) \longrightarrow (\rho\ (\boldsymbol{\theta} \leftarrow \{\ \mathbf{s} \mapsto \langle \text{n} + \mathcal{E}val[\![\mathbf{e}\ ,\ \boldsymbol{\theta}]\!]\ ,\ \texttt{new}\rangle\ \}).\ \mathbf{E}[\texttt{nothing}]) \qquad \textbf{[set-new]}$$
$$\text{where } \boldsymbol{\theta} \vdash \mathbf{E}\ \text{det},\ \boldsymbol{\theta}(\mathbf{s}) = \langle \text{n}\ ,\ \texttt{new}\rangle,\ \text{FV}(\mathbf{e}) \subset \text{dom}(\boldsymbol{\theta}),\ \forall\ \text{s} \in \text{FV}(\mathbf{e}).\ \boldsymbol{\theta}(\text{s}) = \langle\_\ ,\ \texttt{ready}\rangle$$

$$(\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\texttt{+=}\ \mathbf{s}\ \mathbf{e})]) \longrightarrow (\rho\ (\boldsymbol{\theta} \leftarrow \{\ \mathbf{s} \mapsto \langle\mathcal{E}val[\![\mathbf{e}\ ,\ \boldsymbol{\theta}]\!]\ ,\ \texttt{new}\rangle\ \}).\ \mathbf{E}[\texttt{nothing}]) \qquad \textbf{[set-old]}$$
$$\text{where } \boldsymbol{\theta} \vdash \mathbf{E}\ \text{det},\ \boldsymbol{\theta}(\mathbf{s}) = \langle\_\ ,\ \texttt{old}\rangle,\ \text{FV}(\mathbf{e}) \subset \text{dom}(\boldsymbol{\theta}),\ \forall\ \text{s} \in \text{FV}(\mathbf{e}).\ \boldsymbol{\theta}(\text{s}) = \langle\_\ ,\ \texttt{ready}\rangle$$

$$(\rho\ \boldsymbol{\theta}.\ \mathbf{p}) \longrightarrow (\rho\ \text{set-ready}(\boldsymbol{\theta}\ ,\ \mathbb{S}_{2}).\ \mathbf{p}) \qquad \textbf{[readyness]}$$
$$\text{where } (\boldsymbol{\theta} \vdash \mathbf{p}\ \text{blocked or}\ \mathbf{p} \in \mathbf{done}),$$
$$\{\ \mathbf{S} \in \text{dom}(\boldsymbol{\theta}\ )\ |\ \boldsymbol{\theta}^{c}(\mathbf{S}) = \texttt{unknown}\ \} \setminus \texttt{Can}[\![\mathbf{p},\ \boldsymbol{\theta}]\!].\text{S} = \emptyset,$$
$$\mathbb{S}_{1} = \{\ \mathbf{s} \in \text{dom}(\boldsymbol{\theta}\ )\ |\ \boldsymbol{\theta}^{c}(\mathbf{s}) = \langle \mathbf{ev}\ ,\ \textbf{shared-status}\rangle\ \},\ \textbf{shared-status} \in \{\texttt{new}\ ,\ \texttt{old}\},$$
$$\mathbb{S}_{2} = \mathbb{S}_{1} \setminus \texttt{Can}[\![\mathbf{p},\ \boldsymbol{\theta}]\!].\text{sh},\ \mathbb{S}_{2} \neq \emptyset$$

sequential variables

$$(\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\texttt{var}\ \mathbf{x} \coloneqq \mathbf{e}\ \mathbf{p})]) \longrightarrow (\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\rho\ \{\ \mathbf{x} \mapsto \mathcal{E}val[\![\mathbf{e}\ ,\ \boldsymbol{\theta}]\!]\ \}.\ \mathbf{p})]) \qquad \textbf{[var]}$$
$$\text{where } \boldsymbol{\theta} \vdash \mathbf{E}\ \text{det},\ \text{FV}(\mathbf{e}) \subset \text{dom}(\boldsymbol{\theta}),\ \forall\ \text{s} \in \text{FV}(\mathbf{e}).\ \boldsymbol{\theta}(\text{s}) = \langle\_\ ,\ \texttt{ready}\rangle$$

$$(\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\coloneqq\ \mathbf{x}\ \mathbf{e})]) \longrightarrow (\rho\ (\boldsymbol{\theta} \leftarrow \{\ \mathbf{x} \mapsto \mathcal{E}val[\![\mathbf{e}\ ,\ \boldsymbol{\theta}]\!]\ \}).\ \mathbf{E}[\texttt{nothing}]) \qquad \textbf{[set-var]}$$
$$\text{where } \boldsymbol{\theta} \vdash \mathbf{E}\ \text{det},\ \mathbf{x} \in \text{dom}(\boldsymbol{\theta}),\ \text{FV}(\mathbf{e}) \subset \text{dom}(\boldsymbol{\theta}),\ \forall\ \text{s} \in \text{FV}(\mathbf{e}).\ \boldsymbol{\theta}(\text{s}) = \langle\_\ ,\ \texttt{ready}\rangle$$

$$(\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\texttt{if}\ \mathbf{x}\ \mathbf{p}\ \mathbf{q})]) \longrightarrow (\rho\ \boldsymbol{\theta}.\ \mathbf{E}[\mathbf{p}])\ \text{ where } \boldsymbol{\theta} \vdash \mathbf{E}\ \text{det},\ \mathbf{x} \in \text{dom}(\boldsymbol{\theta}),\ \boldsymbol{\theta}(\mathbf{x}) \neq 0 \qquad \textbf{[if-true]}$$

$$(\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\texttt{if}\ \mathbf{x}\ \mathbf{p}\ \mathbf{q})]) \longrightarrow (\rho\ \boldsymbol{\theta}.\ \mathbf{E}[\mathbf{q}])\ \text{ where } \boldsymbol{\theta} \vdash \mathbf{E}\ \text{det},\ \boldsymbol{\theta}(\mathbf{x}) = 0 \qquad \textbf{[if-false]}$$

ρ

$$(\rho\ \boldsymbol{\theta}_{1}.\ \mathbf{E}[(\rho\ \boldsymbol{\theta}_{2}.\ \mathbf{p})]) \longrightarrow (\rho\ (\boldsymbol{\theta}_{1} \leftarrow \boldsymbol{\theta}_{2}).\ \mathbf{E}[\mathbf{p}])\ \text{ where } \boldsymbol{\theta}_{1} \vdash \mathbf{E}\ \text{det} \qquad \textbf{[merge]}$$

seq

$$(\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\texttt{seq}\ \texttt{nothing}\ \mathbf{q})]) \longrightarrow (\rho\ \boldsymbol{\theta}.\ \mathbf{E}[\mathbf{q}])\ \text{ where } \boldsymbol{\theta} \vdash \mathbf{E}\ \text{det} \qquad \textbf{[seq-done]}$$

$$(\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\texttt{seq}\ (\texttt{exit}\ \text{n})\ \mathbf{q})]) \longrightarrow (\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\texttt{exit}\ \text{n})])\ \text{ where } \boldsymbol{\theta} \vdash \mathbf{E}\ \text{det} \qquad \textbf{[seq-exit]}$$

trap

$$(\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\texttt{trap}\ \mathbf{stopped})]) \longrightarrow (\rho\ \boldsymbol{\theta}.\ \mathbf{E}[\downarrow^{\text{p}}\ \mathbf{stopped}])\ \text{ where } \boldsymbol{\theta} \vdash \mathbf{E}\ \text{det} \qquad \textbf{[trap]}$$

par

$$(\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\texttt{par}\ \mathbf{stopped}\ \mathbf{done})]) \longrightarrow (\rho\ \boldsymbol{\theta}.\ \mathbf{E}[\mathbf{stopped} \sqcap_{\|} \mathbf{done}])\ \text{ where } \boldsymbol{\theta} \vdash \mathbf{E}\ \text{det} \qquad \textbf{[parr]}$$

$$(\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\texttt{par}\ \mathbf{paused}\ \mathbf{stopped})]) \longrightarrow (\rho\ \boldsymbol{\theta}.\ \mathbf{E}[\mathbf{paused} \sqcap_{\|} \mathbf{stopped}])\ \text{ where } \boldsymbol{\theta} \vdash \mathbf{E}\ \text{det} \qquad \textbf{[parl]}$$

$$(\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\texttt{suspend}\ \mathbf{stopped}\ \mathbf{S})]) \longrightarrow (\rho\ \boldsymbol{\theta}.\ \mathbf{E}[\mathbf{stopped}])\ \text{ where } \boldsymbol{\theta} \vdash \mathbf{E}\ \text{det} \qquad \textbf{[suspend]}$$

$$(\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\texttt{loop}\ \mathbf{p})]) \longrightarrow (\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\overline{\texttt{loop}}\ \mathbf{p}\ \mathbf{p})])\ \text{ where } \boldsymbol{\theta} \vdash \mathbf{E}\ \text{det} \qquad \textbf{[loop]}$$

$$(\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\overline{\texttt{loop}}\ (\texttt{exit}\ \text{n})\ \mathbf{q})]) \longrightarrow (\rho\ \boldsymbol{\theta}.\ \mathbf{E}[(\texttt{exit}\ \text{n})])\ \text{ where } \boldsymbol{\theta} \vdash \mathbf{E}\ \text{det} \qquad \textbf{[loop\^{}stop-exit]}$$

Fig. 26. Standard Reduction Rules

$$\frac{\boldsymbol{\theta} \vdash \mathbf{e}\ \text{blocked}}{\boldsymbol{\theta} \vdash (\texttt{var}\ \mathbf{x} \coloneqq \mathbf{e}\ \mathbf{p})\ \text{blocked}} \qquad \frac{\boldsymbol{\theta} \vdash \mathbf{e}\ \text{blocked}}{\boldsymbol{\theta} \vdash (\coloneqq \mathbf{x}\ \mathbf{e})\ \text{blocked}} \qquad \boxed{\boldsymbol{\theta} \vdash \mathbf{p}\ \text{blocked}}$$

$$\frac{\boldsymbol{\theta}(\mathbf{S}) = \texttt{unknown}}{\boldsymbol{\theta} \vdash (\texttt{present}\ \mathbf{S}\ \mathbf{p}\ \mathbf{q})\ \text{blocked}} \qquad \frac{\boldsymbol{\theta} \vdash \mathbf{p}\ \text{blocked}}{\boldsymbol{\theta} \vdash (\texttt{suspend}\ \mathbf{p}\ \mathbf{S})\ \text{blocked}} \qquad \frac{\boldsymbol{\theta} \vdash \mathbf{p}\ \text{blocked}}{\boldsymbol{\theta} \vdash (\texttt{trap}\ \mathbf{p})\ \text{blocked}}$$

$$\frac{\boldsymbol{\theta} \vdash \mathbf{p}\ \text{blocked} \qquad \boldsymbol{\theta} \vdash \mathbf{q}\ \text{blocked}}{\boldsymbol{\theta} \vdash (\texttt{par}\ \mathbf{p}\ \mathbf{q})\ \text{blocked}} \qquad \frac{\boldsymbol{\theta} \vdash \mathbf{p}\ \text{blocked}}{\boldsymbol{\theta} \vdash (\texttt{par}\ \mathbf{p}\ \mathbf{done})\ \text{blocked}} \qquad \frac{\boldsymbol{\theta} \vdash \mathbf{q}\ \text{blocked}}{\boldsymbol{\theta} \vdash (\texttt{par}\ \mathbf{done}\ \mathbf{q})\ \text{blocked}}$$

$$\frac{\boldsymbol{\theta} \vdash \mathbf{p}\ \text{blocked}}{\boldsymbol{\theta} \vdash (\texttt{seq}\ \mathbf{p}\ \mathbf{q})\ \text{blocked}} \qquad \frac{\boldsymbol{\theta} \vdash \mathbf{p}\ \text{blocked}}{\boldsymbol{\theta} \vdash (\texttt{l}\overline{\texttt{oo}}\texttt{p}\ \mathbf{p}\ \mathbf{q})\ \text{blocked}}$$

$$\frac{\boldsymbol{\theta} \vdash \mathbf{e}\ \text{blocked}}{\boldsymbol{\theta} \vdash (\texttt{shared}\ \mathbf{s} \coloneqq \mathbf{e}\ \mathbf{p})\ \text{blocked}} \qquad \frac{\boldsymbol{\theta} \vdash \mathbf{e}\ \text{blocked}}{\boldsymbol{\theta} \vdash (\mathrel{+}= \mathbf{s}\ \mathbf{e})\ \text{blocked}}$$

$$\frac{\mathbf{s} \in \mathrm{FV}(\mathbf{e}) \qquad \boldsymbol{\theta}(\mathbf{s}) = \langle \mathrm{n}\ , \texttt{old} \rangle}{\boldsymbol{\theta} \vdash \mathbf{e}\ \text{blocked}} \qquad \frac{\mathbf{s} \in \mathrm{FV}(\mathbf{e}) \qquad \boldsymbol{\theta}(\mathbf{s}) = \langle \mathrm{n}\ , \texttt{new} \rangle}{\boldsymbol{\theta} \vdash \mathbf{e}\ \text{blocked}} \qquad \boxed{\boldsymbol{\theta} \vdash \mathbf{e}\ \text{blocked}}$$

$$\boxed{\boldsymbol{\theta} \vdash \mathbf{E}\ \text{det}}$$

$$\frac{\boldsymbol{\theta} \vdash \mathbf{E}\ \text{det}}{\boldsymbol{\theta} \vdash (\texttt{seq}\ \mathbf{E}\ \mathbf{q})\ \text{det}} \qquad \frac{\boldsymbol{\theta} \vdash \mathbf{E}\ \text{det}}{\boldsymbol{\theta} \vdash (\texttt{l}\overline{\texttt{oo}}\texttt{p}\ \mathbf{E}\ \mathbf{q})\ \text{det}} \qquad \frac{}{\boldsymbol{\theta} \vdash \texttt{[]}\ \text{det}}$$

$$\frac{\boldsymbol{\theta} \vdash \mathbf{E}\ \text{det}}{\boldsymbol{\theta} \vdash (\texttt{suspend}\ \mathbf{E}\ \mathbf{S})\ \text{det}} \qquad \frac{\boldsymbol{\theta} \vdash \mathbf{E}\ \text{det}}{\boldsymbol{\theta} \vdash (\texttt{trap}\ \mathbf{E})\ \text{det}}$$

$$\frac{\boldsymbol{\theta} \vdash \mathbf{E}\ \text{det}}{\boldsymbol{\theta} \vdash (\texttt{par}\ \mathbf{E}\ \mathbf{q})\ \text{det}} \qquad \frac{\boldsymbol{\theta} \vdash \mathbf{E}\ \text{det}}{\boldsymbol{\theta} \vdash (\texttt{par}\ \mathbf{done}\ \mathbf{E})\ \text{det}} \qquad \frac{\boldsymbol{\theta} \vdash \mathbf{E}\ \text{det} \qquad \boldsymbol{\theta} \vdash \mathbf{p}\ \text{blocked}}{\boldsymbol{\theta} \vdash (\texttt{par}\ \mathbf{p}\ \mathbf{E})\ \text{det}}$$

Fig. 27. Standard Reduction Auxiliary Judgments

- Microstep operational semantics compute the result of an instant as a series of small-step style transitions until the instant is considered terminated. Our semantics is in this style.
- Circuit semantics gives meaning to Esterel programs by translation to Boolean circuits.

Semantics of Esterel are also classified as logical or constructive. Logical semantics is simpler, but gives meaning to programs that are logically correct but non-constructive. Constructive semantics use constructive information propagation to enforce both (Berry 2002).

$\text{set-absent} : \boldsymbol{\theta}\ (\text{Setof}\ \mathbf{S}) \longrightarrow \boldsymbol{\theta}$
$\text{set-absent}(\boldsymbol{\theta}\ ,\ \mathbb{S}) =$
$\text{set-absent}(\boldsymbol{\theta} \leftarrow \{\ \mathbf{S} \mapsto \texttt{absent}\ \}\ ,\ \mathbb{S} \setminus \{\ \mathbf{S}\ \})$
$\quad \text{where } \mathbf{S} \in \mathbb{S}$
$\text{set-absent}(\boldsymbol{\theta}\ ,\ \mathbb{S}) =$
$\boldsymbol{\theta}$

$\text{set-ready} : \boldsymbol{\theta}\ (\text{Setof}\ \mathbf{s}) \longrightarrow \boldsymbol{\theta}$
$\text{set-ready}(\boldsymbol{\theta}\ ,\ \mathbb{S}) =$
$\text{set-ready}(\boldsymbol{\theta} \leftarrow \{\ \mathbf{s} \mapsto \langle \text{n}\ ,\ \texttt{ready} \rangle\ \}\ ,\ \mathbb{S} \setminus \{\ \mathbf{s}\ \})$
$\quad \text{where } \mathbf{s} \in \mathbb{S},\ \boldsymbol{\theta}(\mathbf{s}) = \langle \text{n}\ ,\ \textbf{shared-status} \rangle$
$\text{set-ready}(\boldsymbol{\theta}\ ,\ \mathbb{S}) =$
$\boldsymbol{\theta}$

$$
\begin{aligned}
&\sqcap_{\|} : \textbf{done}\ \textbf{done} \not\rightarrow \textbf{done} \\
&\texttt{nothing} \sqcap_{\|} \textbf{done} &&= \textbf{done} \\
&\textbf{done} \sqcap_{\|} \texttt{nothing} &&= \textbf{done} \\
&(\texttt{exit}\ \text{n}_1) \sqcap_{\|} (\texttt{exit}\ \text{n}_2) &&= (\texttt{exit}\ \max(\text{n}_1\ ,\ \text{n}_2)) \\
&(\texttt{exit}\ \text{n}) \sqcap_{\|} \textbf{paused} &&= (\texttt{exit}\ \text{n}) \\
&\textbf{paused} \sqcap_{\|} (\texttt{exit}\ \text{n}) &&= (\texttt{exit}\ \text{n})
\end{aligned}
$$

Fig. 28. Standard Reduction Auxiliary Metafunctions

Finally, semantics of Esterel are distinguished by how much of the language they cover. Some cover all of Kernel Esterel, while others cover only Pure Esterel, which omits shared and sequential variables.

Berry and Gonthier (1992) give two operational semantics of Esterel, a macrostep logical semantics called the behavioral semantics and a microstep logical semantics called the execution semantics. They have proved these equivalent and, for the latter, proved a confluence theorem.

Berry (2002) gives an update to the logical behavioral (macrostep) semantics to make it constructive. The logical behavioral semantics requires existence and uniqueness of a behavior, without explaining how it could be computed, while the constructive semantics introduces Can to do it in an effective but restricted way. Berry (2002) also gives the state behavioral semantics, another macrostep semantics.

The constructive operational semantics (COS) first appears in Potop-Butucaru (2002)'s thesis. It uses program decorations track control flow and avoid rewriting the program. The COS model, like the constructive behavioral semantics, avoids giving meaning to logically incorrect programs, but unlike the behavioral semantics, it provides a guide toward efficient implementation.

Like some of those semantics, our semantics handles the larger language (Kernel Esterel) and accounts for constructiveness. Unlike all of those semantics, our semantics works by term rewriting, substituting equals for equals and simplifying programs, which makes it a good basis for proving program fragment equivalences.

Circuit semantics, such at those that appear in Berry (2002) and Potop-Butucaru et al. (2007), are the semantics generally used by Esterel implementations like Hiphop.js and Esterel v5. These implementations use circuits as an intermediate representation during compilation.

These semantics can be used for program optimization (as ours can too), but in a different way than ours. Because the translation to circuits is complex, it is difficult to connect transformations done at the circuit level back to the source level, so these semantics would not form a good basis for, say, refactoring tools, or other more human-centered tasks. Another contrast is that these semantics are much better suited to whole-program optimizations (e.g., using existing CAD tools to simplify the circuit) whereas our semantics is better suited to local transformations.

Additional semantics include Tardieu (2007)'s, who uses a different technique than constructiveness to eliminate logically incorrect programs. It handles only Pure Esterel.

Our semantics of Esterel closely follows the work of Felleisen and Hieb (1992) on semantics for programs with state. In particular, we borrow the idea of internalizing state into terms using a ϱ term that binds a partial store embedded at any level in a term. However, unlike Felleisen and Hieb, because of Esterel's `par` construct, we do not have unique decomposition into an evaluation context and redex. This means that the [**merge**] rule can merge from both sides of a `par` into the same position, which complicates our proofs. Our semantics also has to handle many Esterel-specific notions, which are not a concern in Felleisen and Hieb (1992)'s work.

ACKNOWLEDGMENTS

Thanks to Daniel Feltey, Christos Dimoulas, and Vincent St-Amour for their feedback on drafts of this paper. Thanks to Colin Vidal, Gérard Berry, and Manuel Serrano for their help with Hiphop.js, Esterel v5 and, more generally, understanding Esterel's semantics and feedback on this work. Thanks to Matthias Felleisen for his help with how to structure the proofs and help understanding the semantics of state. A special thanks to Gérard Berry for suggesting that we use Newman's lemma; without it we would probably never have finished the proof of confluence. This work was supported by NSF proposal CCF-1526109.

BIBLIOGRAPHY

H. Barendregt. The Lambda Calculus: its Syntax and Semantics. North Holland, 1984.

Albert Benveniste and Gérard Berry. The Synchronous Approach to Reactive and Real-Time Systems. *Proceedings of the IEEE* 79(9), 1991.

Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The Synchronous Languages 12 Years Later. *Proceedings of the IEEE* 91(1), 2002.

Gérard Berry. The Esterel v5 Language Primer Version v5_91. École des Mines de Paris, CMA and INRIA, Sophia-Antipolis, France, 2000.

Gérard Berry. The Constructive Semantics of Pure Esterel (Draft Version 3). 2002.

Gérard Berry, Amar Bouali, Xavier Fornari, Emmanuel Ledinot, Eric Nassor, and Robert de Simone. ESTEREL: A Formal Method Applied to Avionic Software Development. *Science of Computer Programming* 36(1), pp. 5–25, 2000.

Gérard Berry and Georges Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming* 19(2), pp. 87–152, 1992.

Gérard Berry, Cyprien Nicolas, and Manuel Serrano. HipHop: A Synchronous Reactive Extension for Hop. In *Proc. PLASTIC*, 2011.

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. Semantics Engineering with PLT Redex. MIT Press, 2009.

Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science* 103(2), pp. 235–271, 1992.

Matthew Flatt and PLT. Reference: Racket. PLT Inc., PLT-TR-2010-1, 2010. http://racket-lang.org/tr1/

Spencer P. Florence, Burke Fetscher, Matthew Flatt, William H. Temps, Tina Kiguradze, Dennis P. West, Charlotte Niznik, Paul R. Yarnold, Robert Bruce Findler, and Steven M. Belknap. POP-PL: A Patient-Oriented Prescription Programming Language. In *Proc. GPCE*, 2015.

Lindsey Kuper and Ryan R. Newton. LVars: lattice-based data structures for deterministic parallelism. In *Proc. Workshop on Functional High-performance Computing (FHPC)*, 2013.

Michael Mendler, Thomas R. Shiple, and Gérard Berry. Constructive Boolean circuits and the exactness of timed ternary simulation. *Formal Methods in System Design* 40, pp. 283–329, 2012.

Benjamin Pierce. Types and Programming Languages. MIT Press, 2002.

Gordon Plotkin. Call-by-name, Call-by-value, and the λ-Calculus. *Theoretical Computer Science* 1(2), pp. 125–159, 1975.

Dumitru Potop-Butucaru. Optimizations for Faster Simulation of Esterel Programs. PhD dissertation, Ecole des Mines de Paris, 2002.

Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. Compiling Esterel. Springer, 2007.

Klaus Schneider and Michael Wenz. A New Method for Compiling Schizophrenic Synchronous Programs. In *Proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2001.

Manuel Serrano and Pierre Weis. Bigloo: a portable and optimizing compiler for strict functional languages. In *Proc. Static Analysis Symposium*, 1995.

Olivier Tardieu. A Deterministic Logical Semantics for Pure Esterel. *ACM Transactions on Programming Languages and Systems* 29(2), 2007.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *Proc. Programming Language Design and Implementation (PLDI)*, 2011.