The Functional, the Imperative, and the Sudoku: Getting Good, Bad, and Ugly to Get Along (Functional Pearl)

MANUEL SERRANO, Inria, France and Université Côte d'Azur, France
ROBERT BRUCE FINDLER, Northwestern University, United States of America

Conventional wisdom suggests that the benefits of functional programming no longer apply in the presence of even a small amount of imperative code, as if the addition of imperative code effectively subtracts. And yet, as we show in this paper, combining functional programming with the special imperative language Esterel provides a multiplicative improvement to the benefits of functional programming.

The key to the benefit of both Esterel and functional programming stems from a restriction that both share. As in functional programming, where only the inputs to a function determine its outputs, the state of an Esterel computation is fully determined by the program's input and the state that the computation had in the previous time step, where the notion of a time step is explicit in the language. Esterel's guarantee holds even though Esterel programs feature concurrent threads, mutable state, and the ability to create, suspend, and even terminate threads as the computation proceeds. This similarity is the root of the benefits that programmers accrue as they informally reason about their program's behavior.

To illustrate these benefits, the bulk of this paper consists of an in-depth exploration of HipHop code (a mashup of JavaScript and Esterel) that implements a Sudoku solver, showing how it is possible to write code that is as easy to understand as if it were written in a pure functional programming style, even though it uses multiple threads, mutable state, thread preemption, and even thread abortion. Even better, concurrent composition and task canceling provide significant program structuring benefits that allow a clean decomposition and task separation in the solver.

CCS Concepts: • **Software and its engineering → Functional languages**; **Imperative languages**.

Additional Key Words and Phrases: HipHop, Esterel, JavaScript, Sudoku

ACM Reference Format:
Manuel Serrano and Robert Bruce Findler. 2024. The Functional, the Imperative, and the Sudoku: Getting Good, Bad, and Ugly to Get Along (Functional Pearl). *Proc. ACM Program. Lang.* 8, ICFP, Article 242 (August 2024), 26 pages. https://doi.org/10.1145/3674631

1 Introduction

Expressing computations as the composition of functions that depend only on their inputs has significant advantages for reasoning about programs, both for people and machines. If the input to a function is all that's required to determine its output, then the reasoning about an entire program can be broken down into the simpler tasks of reasoning about each of the functions in the program and then, separately, reasoning about how they compose. Beyond supporting proofs about properties of programs, programs structured in this manner offer many more practical benefits; they are easy to test and amenable to a host of other techniques for gaining confidence in the correctness of the program.

Authors' Contact Information: Manuel Serrano, Inria, France and Université Côte d'Azur, France, Manuel.Serrano@inria.org; Robert Bruce Findler, Northwestern University, United States of America, robby@cs.northwestern.edu.

ACM 2475-1421/2024/8-ART242
https://doi.org/10.1145/3674631

Still, if the value of functional programming is in the ability to reason about parts independently and compose them successfully, perhaps the focus on functions themselves is too limiting. We use this paper to make the case that structuring programs in modular parts that can be reasoned about separately (and precisely) generalizes to what might seem like the opposite of functional programming, namely imperative programming. We make the case through a specific imperative programming language, Esterel (Berry 2002; Berry and Gonthier 1992; Potop-Butucaru et al. 2007).

Esterel has all the hallmarks of imperative programming, allowing computations to be expressed using state change in concurrent threads. Nevertheless, central to Esterel is the old idea that a program should realize a mathematical function (Böhm 1954; McCarthy 1960).

Normally, when two different threads read and write the same variable in an imperative language, chaos ensues. In Java, we have merely the possibility of race conditions that lead to non-determinism. Although non-determinism of this kind is a terrible consequence, in other languages the consequences are even worse. In C++, race conditions are undefined behavior.[1] Going down closer to the machine, we find that specifications of memory models include many specific strange behaviors, even including the possibility of variables taking on values that have never appeared anywhere else in the program.

In Esterel, however, the behavior of programs that access shared state concurrently is entirely deterministic. Sometimes, an error occurs (errors that can often be detected with static analysis) but even the criteria for these errors are independent of the scheduling of concurrent threads and, in general, well specified for all Esterel programs. Even better, Esterel's guarantees scale up beyond concurrent readers and writers and hold when threads create each other, pause each other, or even abort each other.

The power of Esterel's determinism coupled with shared mutable state brings with it an important form of program composition, namely concurrent composition. As we shall see through the exploration of code, the ability to compose loosely coupled tasks concurrently, and optionally cancel those tasks when their results are no longer needed, is a powerful structuring tool for organizing programs. And, thanks to the determinism of Esterel, reasoning about the code is no more complex than reasoning about code in a functional programming language.

Beyond the benefits that Esterel's determinism brings to imperative programming, the combination of Esterel-style imperative programming and functional programming in a single program is even more striking and powerful than each is in isolation.

We make all of these ideas concrete through the modern instantiation of Esterel in JavaScript called HipHop (Berry and Serrano 2014, 2020).[2] We introduce HipHop and Esterel using a conventional traffic light example and then show how we can use these same ideas to effectively structure a Sudoku solver, an unconventional application of these programming techniques.

2 Introduction to HipHop

Although HipHop combines Esterel and JavaScript into a single programming system, the two languages have dramatically different semantics. Accordingly, HipHop uses staging to combine them. That is, a HipHop program is much like a JavaScript program but with extra syntax that makes it convenient to build Esterel abstract syntax trees as JavaScript values. These trees can then, at JavaScript runtime, be compiled and executed to evaluate Esterel programs.

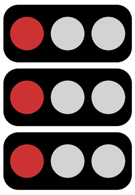
To see how this plays out and to introduce the essential aspects of the semantics of Esterel programs, we turn to an example program that controls a conventional traffic light signal with three lights: red, green, and orange, as you might encounter while driving a car.

[1]See `https://blog.regehr.org/archives/213` for an introduction to the horrific subtleties of undefined behavior.
[2]Not to be confused with HHVM, the HipHop Virtual Machine

Traffic light controllers are, at their hearts, state machines and programming such state machines is precisely where Esterel shines. In a conventional, imperative programming language, one can also program such state machines, but the transitions between the states are codified via programming patterns or documentation about how to use various APIs. In Esterel, in contrast, the semantics of the programming language directly captures the idea of a state transition. Specifically, every Esterel program executes via a series of *instants*, each of which must terminate after some finite amount of time. Each instant corresponds to the computation of the next state from the current state and they are explicit in the syntax of a program, via the `yield` keyword. Accordingly, every active thread in every Esterel program must evaluate a `yield` statement eventually and, when all threads have done so, the instant ends. The next instant picks up right where the previous one left off, with control resuming at the point right after the `yield` statement.

As we are focused on safety, let us begin with a traffic light that disallows any crossings of the intersection at all, by being permanently red. Here is the code:

```
const TL1 = hiphop module() {
   inout light = new Set() combine (x,y) => x.union(y);
   loop {
      emit light(new Set(["red"]));
      yield;
   }
}
```



The first line declares that the JavaScript variable `TL1` is bound to an Esterel module. The keyword `hiphop` is a quotation operator that builds a syntax tree for an Esterel fragment. Here it is building a `module`, a top-level Esterel program. To aid readability, we color the keywords in three colors: purple for JavaScript, olive for Esterel, and blue for the keywords that shift between the two languages.

The body of the module starts by declaring the signal `light`, using the `inout` keyword. In Esterel, signals play the role of imperative variables, whose values change over time. Because of Esterel's determinism, however, the way signals' values are allowed to change is more tightly controlled. Specifically, every signal can have only one (possibly compound) value in each instant, although the values may change across instants. Additionally, a signal's value is set via the `emit` statement, but `emit` does not directly set the value of the signal. Instead, each signal is associated with an initial value and an associative, commutative combining function. All of the values supplied at each emission in a particular instant are collected using that combining function and that becomes the value of the signal in that instant. The declaration of `light` says that if there are no emissions, the value of the signal is the empty set and all of the emissions are collected by unioning the sets.

The remaining three lines of this Esterel program are an infinite loop that repeatedly emits the singleton set containing `"red"` and then yields, ending the instant. Accordingly, this program's traffic light is always red. The traffic lights along the right edge of the code illustrate the behavior of this program, with one traffic light picture for each instant that was run, starting from the top and going downwards. The specific lights that appear in this box (and the other boxes in this section) are calculated by running the program in the box.

To run the program, we first compile it into a reactive machine:

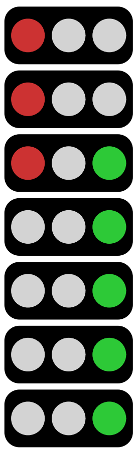
```
const light1 = new hh.ReactiveMachine(TL1);
```

After that, the JavaScript invocation `light1.react({})` runs the first instant, and each subsequent call to `react` runs a subsequent instant. The argument to `react` is an object whose keys are signal names that have been declared with `inout` and whose values are the values on those signals. Passing signal names and values to `react` has the same effect as emitting those values on those signals before doing anything else in the instant. The result of `react` is also an object whose keys are similarly signal names and whose values are the signals' values at the end of the instant, allowing JavaScript to read off the result of the Esterel computation.

Our first traffic light example uses the pattern of an infinite loop that emits a particular value on a particular signal, and it is a common pattern in Esterel, dubbed sustaining the signal. Because it is so common, Esterel supports the `sustain` construct, which uses the same syntax as `emit` but, internally, there is a `loop` and a `yield`.

At first blush, hiding the infinite loop inside another construct seems strange. After all, what if you want the traffic light to turn green at some point; wouldn't that work by adding a parameter to the looping construct that might count down? That is not the Esterel way. An Esterel programmer would, instead, take advantage of concurrent composition and thread abortion to transition the light from one phase to the next. Consider this next version of the traffic light code.

```
 1 const TL2 = hiphop module() {
 2    inout light = new Set() combine (x,y) => x.union(y);
 3    done: {
 4       fork {
 5          sustain light(new Set(["red"]));
 6       } par {
 7          yield;
 8          yield;
 9          break done;
10       }
11    }
12    sustain light(new Set(["green"]));
13 }
```



The `sustain` on line 5 is equivalent to the entire `loop` in the previous program. It is wrapped, however, inside two constructs. The first, `done:`, is simply a label, as in JavaScript, and the braces indicate where control transfers when the program uses `break` to go to the label, in this case down to line 11. The other is the Esterel `fork`...`par` construct, which runs its subexpressions concurrently and, when all of them either complete, break to a label outside the `par`, or `yield`, then the `fork`...`par` itself is done for the current instant.

Although each of these two constructs in isolation might not seem particularly special, their combination, together with the guaranteed determinism of Esterel makes them special indeed. In this instance, we can see that the second thread waits for two instants to pass and then exits to the label. But, because the label is outside the `par`, exiting cancels the infinite loop in the first thread, meaning that control transfers to line 12 and green is emitted.

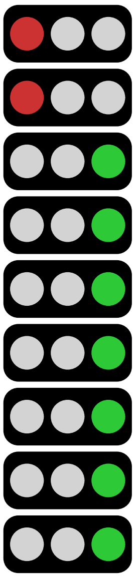
Unfortunately, the way concurrent composition works in the presence of non-local control like `break` means that the emission of the red light also happens in the same instant, resulting in both lights being emitted. Roughly, the semantics of the concurrent composition is that both branches run until both of them either yield, terminate, or break to a label. Once all of the concurrent arms have finished, then fork combines their results into its overall result. If both yield, then the fork itself remains active but the instant is over. If one of the arms breaks to an enclosing label, then the break supersedes the other arms and control transfers to the first point after the label, but in the same instant. Thus, in our example, the first par arm emits red and yields, and only at that point does the fork itself finish, by exiting to the done label, which leads to the second emission.

To get the correct behavior, we need to use one more Esterel construct, suspend. Syntactically, suspend is followed by a parenthesized expression and then some number of statements, enclosed in {}, much like an if statement (without an else). Semantically, a suspend is relevant only when control picks up from a yield in the body of the suspend. In that situation, the expression inside the parentheses is evaluated and, if it is a true value, the yield simply does not wake up and whatever code in the body of the suspend simply does not run. We can solve our problem with the incorrect light transition with suspend, using it to stop the emission of the red signal in the instant that the traffic light transitions to green.

In this version of the traffic light example, in addition to wrapping the sustain in a par, we wrap it in a suspend guarded by the value of a new signal, s, which we declare with signal; the signal keyword indicates that the signal is not visible externally, unlike inout, which declares signals that are visible via react. This is a boolean valued signal, whose default value is false and whose combination function is logical or. The expression s.nowval is the value of the signal, in the current instant.

To fully make the transition from red to green, then, instead of simply breaking after two instants, we also emit a true on s, triggering the suspend and avoiding the red emission.

```
const TL3 = hiphop module() {
   inout light = new Set() combine (x,y) => x.union(y);
   signal s = false combine (x,y) => x || y;
   done: {
      fork {
         suspend (s.nowval) {
            sustain light(new Set(["red"]));
         }
      } par {
         yield;
         yield;
         emit s(true);
         break done;
      }
   }
   sustain light(new Set(["green"]));
}
```



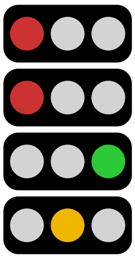
Of course, this is a large amount of code and we need three copies of it, as we will need the same combination of parallelism, labels, and sustain to make all three transitions of the traffic light. We can take advantage of HipHop's JavaScript level to abstract over the pattern, by writing a JavaScript function that accepts the color, the name of the signal to emit, and the number of instants to yield and then produces the Esterel code from the previous example.

```
 1 const phase = (color, light, count) => hiphop {
 2    signal s = false combine (x,y) => x || y;
 3    done: {
 4       fork {
 5          suspend (s.nowval) {
 6             sustain ${light}(new Set([color]));}
 7       } par {
 8          ${Array.from({length: count}, _ => hiphop yield)}
 9          emit s(true);
10          break done;
11       }}}
```

Just as hiphop is the quoting operator, ${...} is the unquoting operator. Its use on line 6 puts the name of the signal into the argument to sustain. On line 8, we use it in a more sophisticated way: when the JavaScript value is an array, the contents of the array are spliced into the code at that point. Since the Array.from expression produces an array whose elements are all hiphop yield, we end up with count copies of yield spliced into the second part of the par.

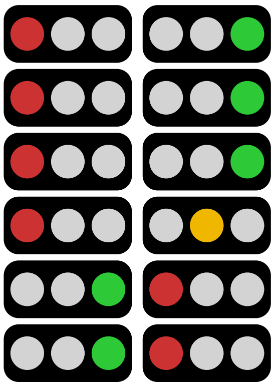
With phase defined, we can simply call it three times in the body of loop to make our traffic light cycle through its three phases, staying red for 2 instants, then green for 1 and orange for 1.

```
const TL4 = hiphop module () {
   inout light = new Set() combine (x,y) => x.union(y);
   loop {
      ${phase("red", "light", 2)}
      ${phase("green", "light", 1)}
      ${phase("orange", "light", 1)}
   }
}
```

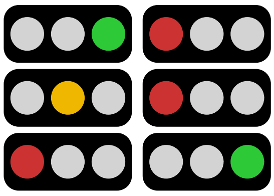


With the phase abstraction built, we can even build two traffic lights to control a simple intersection, being careful to ensure that one of the directions always has a red light.

```
const TL5 = hiphop module() {
   inout ns = new Set() combine (x,y) => x.union(y);
   inout ew = new Set() combine (x,y) => x.union(y);
   fork {
      loop {
         ${phase("green", "ns", 3)}
         ${phase("orange", "ns", 1)}
         ${phase("red", "ns", 4)}
      }
   } par {
      loop {
         ${phase("red", "ew", 4)}
```

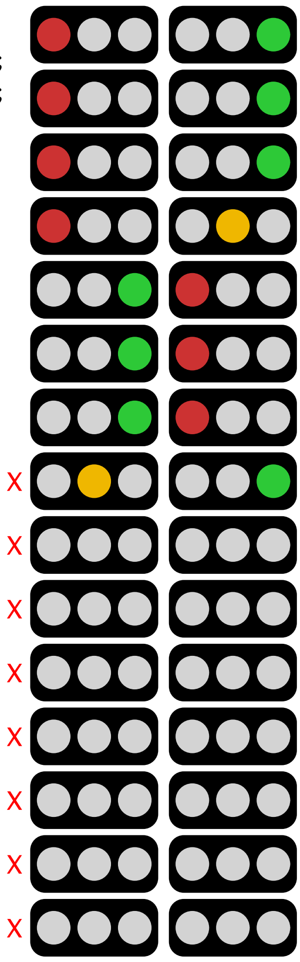


```
            ${phase("green", "ew", 3)}
            ${phase("orange", "ew", 1)}
        }
    }
}
```



Combining the two traffic lights in this manner is fairly subtle and requires some careful calculation to be sure that we always have a red light in one direction or the other. To help ensure the invariant, we can add another concurrent thread that simply checks the invariant and uses another signal that tells the traffic system to switch to a fail-safe mode (commonly, all directions blink red in such a situation). Here is the code; it is localized to a new `par` block that runs concurrently with the traffic lights themselves, again demonstrating the power of Esterel's concurrent composition. The new thread runs an infinite loop, checking the safety condition continuously.

```
const TL6 = hiphop module() {
   inout ns = new Set() combine (x,y) => x.union(y);
   inout ew = new Set() combine (x,y) => x.union(y);
   inout failed = false combine (x,y) => x || y;
   Lfailed: {
      fork {
         loop {
            ${phase("green", "ns", 3)}
            ${phase("orange", "ns", 1)}
            ${phase("red", "ns", 3)}
         }
      } par {
         loop {
            ${phase("red", "ew", 4)}
            ${phase("green", "ew", 3)}
            ${phase("orange", "ew", 1)}
         }
      } par {
         loop {
            if (!(ns.nowval.has("red") ||
                  ew.nowval.has("red"))) {
               break Lfailed;
            }
            yield;
         }
      }
   }
   sustain failed(true)
}
```



Note that the arithmetic in this code is actually wrong so the program goes wrong in the 8th instant. The red "X" next to the traffic lights shows when the `failed` signal is `true`.

To implement the checking, we need to observe the value of signals from within Esterel, which is done with `ns.nowval` and `es.nowval`. Since each signal's value is a set, we can use the standard JavaScript `has` method to test membership and combine the two tests with `||`. In fact, this code is using a different way for Esterel and JavaScript to interact. Specifically, `if` is a bridge between the two languages; it provides a way for Esterel to make callbacks back into JavaScript, during an instant. More precisely, the JavaScript expression in the test position of the `if` expression is wrapped in a thunk and saved as part of the Esterel AST. Then, later, during a reaction, the Esterel machine invokes the thunk to call back into JavaScript to determine which branch of the `if` to take.

There are a few other places where Esterel calls back into JavaScript during an instant that we have seen already: the parenthesized arguments to `emit` and `sustain` also allow arbitrary JavaScript expressions and call back to JavaScript to find the value to emit. Such positions are sprinkled throughout the Esterel code; another pair are the initial value for a signal and the signal's combining function.

The careful reader may have noticed that Esterel's determinism and the ability to have control flow depend on a signal's value during an instant leads to some interesting and subtle questions about the semantics of Esterel. For example, consider this encoding of the Liar's paradox in Esterel:

```
const LiarParadox = hiphop module() {
   signal whatIsThisSignal = false combine (x,y) => x || y;

   if (whatIsThisSignal.nowval) {
      ;
   } else {
      emit whatIsThisSignal(true);
   }
}
```

We start with a boolean-valued signal like `failed` above, whose initial value is `false` and with logical or as the combining function. As you can see, if the value of the signal is true, then the signal is not emitted (and thus the value should not have been true) and, if the value of the signal is false, then the signal is emitted (and thus the value should not have been false).

This program is an error in Esterel, and so are many like it. This class of errors are called *causality errors* and they occur whenever the value of a signal is used where there are, downstream of the use, emitters. For example, if some code checks the value of a signal but, sequentially past that point, the signal is emitted, the code also is a causality error. If a signal's value is checked in parallel to the signal's emission, however, there is no problem. Intuitively, Esterel must be able to find a way to evaluate a program such that all causes (emissions) come before any effects (using `nowval`). If it cannot, the program is considered to have a causality error and is rejected.

Overall, causality is a topic that has seen significant study in the Esterel literature. It does not play an outsize role in our Sudoku solver but, for the interested reader, the first stop for a deeper understanding of causality errors is Berry (2002)'s *The Constructive Semantics of Pure Esterel.* Shiple et al. (1996), Sentovich (1997), and Schneider et al. (2005) have developed analyses to detect causality errors in the context of Esterel. Lustre, a language that complements Esterel well, has also seen significant study of causality errors (Halbwachs and Maraninchi 1995; Halbwachs et al. 1991). An alternative semantics for Esterel by Boussinot and Simone (1996) avoids causality errors by construction and has been explored in the context of ML (Mandel et al. 2015; Mandel and Pouzet 2005) and C (Boussinot 1991). More recently, Krishnamurthy and Serrano (2021) offer improved debugging of causality errors in HipHop.

3 Introduction to Sudoku and Its Terminology

Sudoku is a logic puzzle that dates back to 1979 (Garns 1979) but was popularized by Nikoli in Japan. Each puzzle consists of a nine by nine grid that must be filled in with numbers between 1 and 9 such that each column, each row, and each of the 9 non-overlapping 3x3 squares all have all nine digits with no repeats. Each of these separate groups of 9 cells is called a “house”, and there are 27 of them: 9 rows, 9 columns, and 9 boxes. Each house consists of three “chutes”, three contiguous cells either horizontally or vertically. Unlike row and column houses, box houses can be seen as a combination of three chutes two different ways, horizontally or vertically.

The puzzles are seeded with some given numbers in such a way that there is exactly one way to fill in the blank squares to satisfy the constraints. Additionally, the given numbers should be arranged in a symmetric pattern, an improvement pioneered by Nikoli. Just to the right is an example from the Puzzle Bank (McLean 2020).

			8		9			
	4	3				8	7	
	2						1	
3				2				9
			3		8			
	1						2	
6		7				4		2
	8	4	2		5	6	9	
2			7		4			1

A first step in solving this puzzle is to notice the 9 in the topmost row. It rules out any 9s appearing in the upper-right box house’s topmost horizontal chute. In a similar manner, the 9 in the right-most column rules out any 9s appearing in that same house’s right-most vertical chute. The givens in that house (the 8, 7, and 1) also rule out 9 from appearing in those cells. Therefore, the one remaining cell, namely the upper-right house’s lower-left cell, must be a 9.

Typically, solving a Sudoku puzzle consists of making such logical deductions and Sudoku enthusiasts collect strategies to find and justify these logical deductions. The Sudoku community has even given them names; the deduction strategy used in the previous paragraph is called “Hidden Single”. Although there are many such deduction strategies available, some Sudoku puzzles eventually require guessing and backtracking search.

4 Solving Sudoku

Solving Sudoku offers a programming challenge that plays to both the strengths of imperative programming in Esterel and functional programming in JavaScript. Guessing and backtracking search are simply and naturally expressed as a recursive process, the bread and butter of functional programming. This recursive traversal would be a poor fit for Esterel, as it does not even have functions, let alone recursive ones.

The management of the state of knowledge of the puzzle, in contrast, is a natural fit for imperative programming. We have some facts and, as we learn more facts, we want to update the set of facts we have, allowing all parts of the program to consult a single place when making queries. Managing this state fits neatly into imperatively updated variables in Esterel, as we can organize the separate tasks that update the state as parallel workers, each independently firing to offer new information without worrying about the order in which the updates occur (except to avoid causality errors). Even better, thread abortion is a boon that simplifies our programming task. Specifically, we can recover from a wrong guess by aborting the threads that make deductions and restarting them in a fresh state. Managing the state in this way is a nightmare in conventional programming languages. Indeed, Java famously deprecated `Thread.stop` as it was deemed too difficult to program with.[3]

[3]https://docs.oracle.com/javase/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html

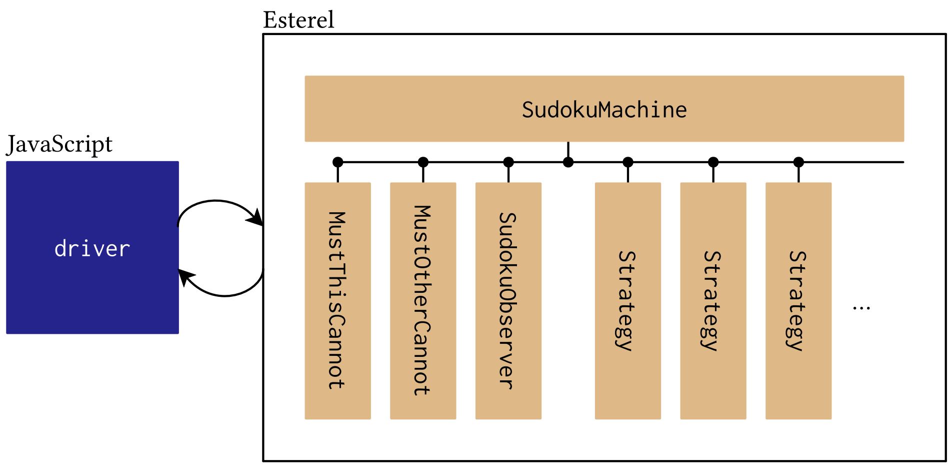


Fig. 1. A Diagrammatic Overview of the Solver

The remainder of this section explores the code that makes up the basic structure of our solver, showing how it plays to the recursive strengths of functional programming and the state management strengths of imperative programming. An overview of the software architecture of the server is shown in figure 1. In blue on the left is the JavaScript driver loop, which we discuss in section 4.1. The tan boxes on the right of the diagram are Esterel code. The Sudoku Machine controls all the others parts of the Esterel code, combining them in parallel, but also aborting them when a bad guess is discovered. We discuss the Sudoku Machine in section 4.2. The first two taller tan boxes to the left, `MustThisCannot` and `MustOtherCannot`, propagate information about the numbers in the puzzle according to the basic rules of Sudoku, and we discuss them in section 4.3. And finally, in section 4.4, we discuss `SudokuObserver`. It monitors the state of the puzzle detecting, among other things, when the puzzle is solved. The strategies and how they interact with the solver are postponed until section 5.

The interface between the JavaScript and Esterel parts of the program are the signals consumed and produced by the Esterel machine. The machine produces two signals: `status`, whose value is either the string `"solved"`, meaning that the puzzle has been solved, `"progress"`, meaning that a logical deduction was made about the puzzle in the instant that just completed, `"stall"`, meaning that it was unable to make any logical deductions (and the puzzle isn't solved yet), or `"reject"`, meaning that an inconsistency was discovered (because of an earlier bad guess). The JavaScript driver should repeatedly run instants in the machine as long as it receives `"progress"`. If it receives `"stall"`, then it should make a guess and if it receives `"reject"`, it should revisit earlier guesses, as one of them was wrong.

The second signal that the machine produces, `unsolved`, contains information to help the JavaScript driver decide which cell to make a guess in. Concretely, `unsolved`'s value is a set of objects, one for each cell in the puzzle whose digit is not yet known. Each object has the fields `i`, `j`, and `digits`. The `i` and `j` fields indicate which cell the information is about and `digits` is a set of digits that the cell might contain. Importantly, if `digits` does not contain a particular digit, then that digit is not possible in that cell.

The input to our Sudoku-solving machine is the givens (although they may also include some guesses made by the JavaScript driver), encoded via values on some signals, as well as a boolean-valued signal `reset` that is true when an inconsistency was discovered. The machine has code that will, when it sees the `reset` signal, clear out internal state to keep the incorrect guesses from polluting the deduction process.

As both the JavaScript code and the Esterel code can discover inconsistencies, we let the JavaScript code make the decision to reset the state of the machine. It may reset the machine when the Esterel code's deductions produce an inconsistency, which is communicated via a `"reject"` value of the `status` signal, but it may also reset the machine when it has exhausted all of the guesses for a particular cell and the puzzle is not solved.

4.1 The Recursive JavaScript Driver

The `driver` function accepts two arguments. The first is the machine itself. The second is `givens`, an object whose keys are signal names and the corresponding values are the values on those signals. The `driver` function's result is an object with at least one key, `status`, whose value is either `"reject"` or `"solved"`. If the value is `"solved"`, then the object is one that came from the Esterel machine, so it also contains all of the signal values from the machine (which encode the solution to the puzzle, as we shall see). Here is the driver code.

```
const driver = (mach, givens) => {
   while (true) {
      const signals = mach.react(givens);
      switch (signals.status) {
         case "progress":
            break;
         case "stall":
            const {i, j, digits} = signals.unsolved.first();
            const newGivens = Object.assign({}, givens);

            for (let guess of digits) {
               newGivens[`must${i}${j}`] = new Set([guess]);
               const newSignals = driver(mach, newGivens);
               if (newSignals.status === "solved") {
                  return newSignals;
               } else {
                  mach.react({reset: true});
               }
            }
            return {status: "reject"};

         case "solved":
         case "reject":
            return signals;
      }
   }
}
```

Each iteration of the outer while loop (line 3) runs an instant of the machine. As long as the status signal's value is "progress", the code keeps running instants of the machine.

If the machine stalls, then we prepare for a recursive invocation, starting on line 7. Line 8 selects one of the cells from the unsolved set in preparation for a loop that iterates over all of the digits that are possible in that cell. Line 9 contains a JavaScript idiom that makes a copy of the givens object, which will be mutated on each iteration of the loop, filling in a new digit to guess.

Then, on line 11, we start the loop. Each iteration of the loop makes a recursive call to driver, updating the new givens with the next guess. If the recursive invocation is successful and the puzzle is solved, then we can return the signals from the machine. If it not, then we continue with the for loop and try a new guess for the selected cell.

Unfortunately, once we have discovered an inconsistency, the internal state of the machine is corrupted because of logical deductions that were based on incorrect information. So, the machine's state needs to be reset so that it contains only valid information. To do so, we run an extra instant with the reset signal set, as shown on line 17. The machine, in response, erases all of its knowledge of the state of the puzzle so all future deductions start from an empty, uncorrupted state.[4]

Once the for loop completes, we know that none of the guesses worked, so on line 20 we make up an object that looks like the value of the signals that the machine would produce when it discovers an inconsistency and we return that.

Finally, if the machine either solves the puzzle or discovers that no solution is possible, then we simply return the set of signals (lines 22-24).

4.2 The Esterel Machine

Now that we have established a driver loop for the Esterel machine, let us turn to the machine's code and see how it computes the values of the status and unsolved signals from the givens.

The machine works with two global signals per cell in the Sudoku board, a "must" signal and a "cannot" signal. All of these signals are set-valued signals with union as the combining operation. Each must signal's value should either be the empty set, indicating that we do not yet know the cell's value, or a singleton set, indicating that we know the cell's value and it should be the element of the set. Each cannot signal tracks information from elsewhere in the puzzle, localizing information about which digits are not possible to be in the cell.

As there are many places in the Sudoku implementation where we need to iterate nine times, we define iota to be an array with the numbers 0 to 8, based on BOARD_SIZE, which is 9.

```
const iota = Array.from({length: BOARD_SIZE}, (_, i) => i);
const digits = new Set(iota.map(v => v + 1));
```

We use iota with various array methods in order to construct pieces of our HipHop machine. A common use of iota is to create 9 pieces of Esterel code, by using the map method, which accepts a function and calls it 9 times, with the numbers 0 through 8. In the definition of digits, shown above, we use that same map method, but we use it to build the set of valid digits in the puzzle, as that will also be useful in the code that follows.

The construction of our Sudoku machine starts in the JavaScript function SudokuMachine. It accepts an array of strategies that is, for the time being, always empty. It returns the HipHop machine that the JavaScript code in the previous subsection invokes.

[4]Although it is possible to design a more elaborate interaction where the Esterel machine checkpoints its knowledge of the puzzle at each step and the JavaScript driver loop records and restores that old state, we opt for a simpler implementation and simply reset to a state where the machine knows only the givens after uncovering a bad guess.

```
 1 const SudokuMachine = strategies => hiphop module() {
 2    inout ... ${iota.flatMap(i => iota.map(j => `must${i}${j}`))} =
 3       new Set() combine (x, y) => x.union(y);
 4    inout ... ${iota.flatMap(i => iota.map(j => `cannot${i}${j}`))} =
 5       new Set() combine (x, y) => x.union(y);
 6    inout reset = false combine (x, y) => x || y;
 7    inout status;
 8    inout unsolved = new Set() combine (x, y) => x.union(y);
 9
10    loop {
11       abort immediate (reset.nowval) {
12          fork {
13             ${MustThisCannot()}
14          } par {
15             ${ForkHouseMap(MustOtherCannot)}
16          } par {
17             ${SudokuObserver()};
18          } par {
19             fork ${strategies}
20          }
21       }
22
23       ${iota.map(i => hiphop ${iota.map(j => hiphop {
24          emit ${`must${i}${j}`}(new Set());
25          emit ${`cannot${i}${j}`}(new Set());
26       })})}
27
28       yield;
29       emit reset(false);
30    }
31 }
```

Lines 2-5 declare the cannot and must signals, using JavaScript to build the actual names of the signals. Specifically the map method of `iota` iterates over the numbers from `0` to 8 and returns a new array with the results of its function, as before. The `flatMap` method is similar, except instead of combining the results of each iteration directly into a list, it flattens one layer of nested lists which, overall, results in a list of 81 names, i.e., `must00`, `must01`, `must02`, etc. The ellipsis preceding the `${}` on lines 2 and 4 is syntax that tells HipHop to expect a sequence of names and to bind them all to the same kind of signal.

Line 7 declares the signal `status`, but it does not have an initial value nor a combining function. It is an error to emit such signals more than once in an instant. The `fork` from 12 to 20 corresponds to the vertically oriented Esterel boxes in figure 1, collecting all of the parallel tasks together.

Focus for a moment on line 11. The `abort` construct packages up the same constructs that we used to move from one traffic light phase to the next, as this is actually a common idiom in Esterel programming. It is simply a syntactic shorthand that accepts a boolean condition (in this case `reset.nowval`) and expands into a use of `break` and a `suspend`. Because of the `immediate` modifier to the abort, it also guards its entire body in a conditional based on the boolean condition, before entering the suspend.

Thus, when `reset` is true, we abort the computations happening on lines 12-21 and go to line 23. At that point, the code uses the `iota.map` idiom again. Because it is nested, we create 81 pieces of Esterel code at that point, each of them copies of lines 24 and 25, but each copy with a different pair of `i` and `j`. Each one uses `$` to create two identifiers, just the same as were created on lines 2 and 4. This time, however, the identifiers are placed into `emit` statements. All together, this code has the effect of resetting each must and cannot signal back to the empty set.

Thus far, we have glossed over one subtle point of signal values, in particular how they carry their values forward from instant to instant. If a signal is never emitted in an instant, its value will be the same as it was in the previous instant (or the initial value, if it has never been emitted at all). But, if a signal is emitted in the current instant, the value of the signal in the previous instant is ignored; it is not passed to the combining function. Accordingly, since the `emit`s on lines 24 and 25 are the only emissions of the musts and cannot when they happen (thanks to the `abort` on line 11), they have the effect of resetting the must and cannots to the empty set for future instants, ensuring that no value is carried forward from the previous instant. Similarly, but after the `yield`, we also emit `false` on `reset` (line 29) to avoid carrying the `true` forward.

4.3 Esterel Workers

This section describes `MustThisCannot` and `MustOtherCannot` and how they are used. Overall, their job is to propagate information based on the rules of Sudoku and to provide a baseline of deduction that ensures that bad guesses are eventually caught.

First, `MustThisCannot` sets the cannot signal for any cell where the must signal is already set.

```
 1 const MustThisCannot = () => hiphop {
 2    fork ${iota.map(i => hiphop fork ${iota.map (j => hiphop {
 3       loop {
 4          let m = this[`must${i}${j}`].nowval;
 5          if (m.size === 1) {
 6             emit ${`cannot${i}${j}`}(digits.difference(m));
 7          }
 8          yield;
 9       }
10    })})}
11 }
```

Line 2 uses the idiom `fork ${iota.map … }` to, as before, create 9 pieces of Esterel code but, because the `fork` has the `$` directly after it, without enclosing `{}`, the 9 pieces of Esterel code are put in parallel with each other, instead of in sequence as we have seen previously. Accordingly, we have 81 copies of lines 3 through 9 running in parallel with each other, each with their own pair of `i` and `j`. Line 4 is also worth a closer look. Like several other places in the embedded Esterel code, the right-hand side of a `let` expression is actually a JavaScript expression. It uses the notation ``this[`must${i}${j}`]``, which combines the use of `this`, to which HipHop binds to an object whose keys are the signals in the current instant, and the use of a JavaScript idiom for field selection. That is, in JavaScript, `o.x` is the same operation as `o["x"]`, both extract the field named `x` from the object `o`. The use of backquotes constructs a JavaScript string, but allows interpolation. Putting all of these pieces together, `m` is bound to the value of must signal of the (i,j)th cell in the Sudoku board. Since the `difference` method computes the set difference, we end up emitting the correct set for the corresponding cannot signal in every instant.

Next, we turn to the code that handles updating the cannot signals based on the must signals that are on other spots in the board. As you recall from the definition of `SudokuMachine`, the expression put in parallel is `ForkHouseMap(MustOtherCannot)`. The `ForkHouseMap` function is a JavaScript function that calls its `proc` argument 27 times, once for each house on the Sudoku board. Each time it passes an array of objects with `i` and `j` fields that give the coordinates of the cells in the house. It combines the results of these 27 calls together in parallel. Here is the code.

```
const ForkHouseMap = proc => hiphop {
   fork {
      fork ${iota.map(i => proc(iota.map(j => {return {i, j}})))}
   } par {
      fork ${iota.map(j => proc(iota.map(i => {return {i, j}})))}
   } par {
      fork ${iota.map(i => {
         const chute_len = Math.sqrt(iota.length);
         const i0 = chute_len * (i % chute_len);
         const j0 = chute_len * Math.floor(i / chute_len);
         return proc(iota.map(j => {
            return {i : i0 + j % chute_len,
                    j : j0 + Math.floor(j * chute_len / iota.length)};
         }))})}
   }
}
```

Implementing `ForkHouseMap` is straightforward with the ideas seen so far, but requires some fiddly calculations to get the box house coordinates right.

`MustOtherCannot` gets the value of the must for each cell in a given house and then, for every cell in the house that is not itself, it emits that same set, but on the cannot signal. Each cell either has an empty set on its must, meaning that the emit on the cannot signal has no effect (thanks to the union combination function), or it has a singleton set, meaning that that value cannot appear anywhere else in the house. All of these emissions are `union`ed together (again thanks to the signal combining function) to calculate the correct cannot set for each cell. Here is the code:

```
const MustOtherCannot = coords => hiphop {
   fork ${coords.map(c => hiphop loop {
      let c_must = this[`must${c.i}${c.j}`].nowval;
      ${coords
         .filter(d => c.i !== d.i || c.j !== d.j)
         .map(d => hiphop {
            emit ${`cannot${d.i}${d.j}`}(c_must);
         })}
      yield;
   })}
}
```

One highlight is the use of the `filter` and map methods from `iota`, first removing the cell itself (c) from consideration and then generating the emits for all the other cells in the house (d).

4.4 The Observer

As the strategies remain empty for now, the only remaining code is the observer, which tracks the state of the knowledge about the board, summarizing it for the recursive JavaScript `driver` function. As we saw in the beginning of this section, the values of the signals `status` and `unsolved` are the outputs of the Esterel code; it is the observer that computes their values. Here is the code.

```
const SudokuObserver = () => hiphop {
   signal progress = false combine (x, y) => x || y;

   loop {
      emit unsolved(new Set());
      emit progress(false);
      cont: {
         ${iota.map(i => hiphop ${iota.map(j => hiphop {
            if (inconsistent(this[`must${i}${j}`].nowval,
                             this[`cannot${i}${j}`].nowval)) {
               emit status("reject");
               break cont;
            }
            if (this[`must${i}${j}`].nowval.size === 0) {
               emit unsolved(new Set([{
                  i: i, j: j,
                  digits: digits.difference(
                     this[`cannot${i}${j}`].nowval)}]));
            }
            if ((this[`must${i}${j}`].nowval.size >
                 this[`must${i}${j}`].preval.size) ||
                (this[`cannot${i}${j}`].nowval.size >
                 this[`cannot${i}${j}`].preval.size)) {
               emit progress(true);
            }
         })})}
         if (unsolved.nowval.size === 0) {
            emit status("solved");
         } else if (progress.nowval) {
            emit status("progress");
         } else {
            emit status("stall");
         }
      } // end cont:
      yield;
   } // end loop
}
```

The observer first declares a local boolean-valued signal, `progress`, that helps to determine if `status`'s result is going to be `"progress"` or not. The main observer loop (lines 4-34) runs one complete iteration on each instant. It starts by initializing the `unsolved` and `progress` signals so their values are determined only by the calculations in this instant. Then, the loop's body asks these three questions of each cell:

- Are this cell's can and must signal values inconsistent (lines 9 and 10)? If so, we know that the status should be `"reject"`, so line 11 emits that and then we break to the `cont:` label, sending control down to line 35 where the instant ends.
- Is this cell's must the empty set (line 14)? If so, this cell is unsolved, so we add it to the `unsolved` signal. This time, we do not break, so control continues on to the next cell.
- Is the set in this cell's must or cannot a larger one than it was in the previous instant (lines 20-23)? This uses a new construct, `preval`, the value of the signal in the previous instant. If the condition holds, then we know that progress has been made, so we emit `true` on the `progress` signal.

Once all three of these `if` expressions execute for all 81 cells and if we do not discover an inconsistency, control arrives at the `if` block on line 27, and we have enough information to determine the value of `status`. If none of the cells are unsolved, then we are done and we can emit `"solved"` on `status`. If not, the value of `progress` determines if we should emit `"progress"` or `"stall"`. Whatever happens, we reach the `yield` on line 35 and the instant is over.

And finally, here is the helper function that determines inconsistency, given the must and cannot sets for a single cell.

```
const inconsistent = (must, cannot) => {
   return must.size > 1 ||
          must.intersection(cannot).size > 0;
}
```

5 Strategies

The Sudoku community has developed many strategies to help people solve Sudoku puzzles and the HoDoKu site (Hobiger 2008) has a comprehensive inventory. In this section, we'll pick three of them and show their implementation in HipHop. Thanks to the structure of the solver, these strategies can be developed independently but still can collaborate with each other.

As we have prepared for strategies back in the second Esterel code block in section 4.2, we simply need to build code that examines the values of the must and cannot signals and contributes the results of logical deductions by emitting additional information. Although we glossed over it when we first discussed `SudokuMachine`, the unquoting `$` for the strategies is directly under a `par`, meaning that all of the strategies are executed in parallel. Furthermore, thanks to Esterel's determinism and support for abortion, the strategies do not need to worry about guessing; they are simply aborted and then restarted in a fresh state without needing to do anything special.

Still, there are rules that a strategy must abide by in order for the puzzle solving process to be successful. Of course, it must not contribute incorrect information. But beyond that, there are two temporal constraints. First, in order to avoid causality errors, the strategies may not consult the value of any of the cannot signals' current values, but instead must use `.preval`, the value in the previous instant. Accessing the value of the must signals, however, is okay. This restriction means that we do not end up with cycles of the kind illustrated by the Liar's Paradox example at the end of section 2. Beyond that safety constraint, there is also a liveness constraint to attend to. Since the JavaScript driver checks for progress at each instant to determine if it will make a new guess, strategies must emit information in the first instant possible, instead of waiting for a future instant.

The liveness constraint deserves a little more explanation. Specifically, imagine a strategy that, instead of looking at the `preval` of a cannot signal looks at its value in the current instant. Then, in order to avoid a causality violation, the strategy `yield`s and emits some information on a must signal in the next instant. While such code would successfully avoid the causality errors, it may defeat the checking that the observer does. That is, even though the strategy knows something about one of the cells, the observer does not observe this knowledge and thus may declare that the solving process is stalled, leading to a restart of the strategies in a fresh state with different guesses.

The strategies we consider in this subsection each reduce the number of guesses required to solve the example puzzle from section 3. Without any strategies, every (non-given) cell must be solved by guessing, meaning we need 52 successful guesses. Of course, depending on the specific guesses made and how long it takes to discover incorrect guesses, there may be considerably more total guesses needed. Indeed, without any of the strategies described here, the code from section 4 makes more than 15,000 total guesses for the example puzzle.

5.1 The Naked Single Strategy

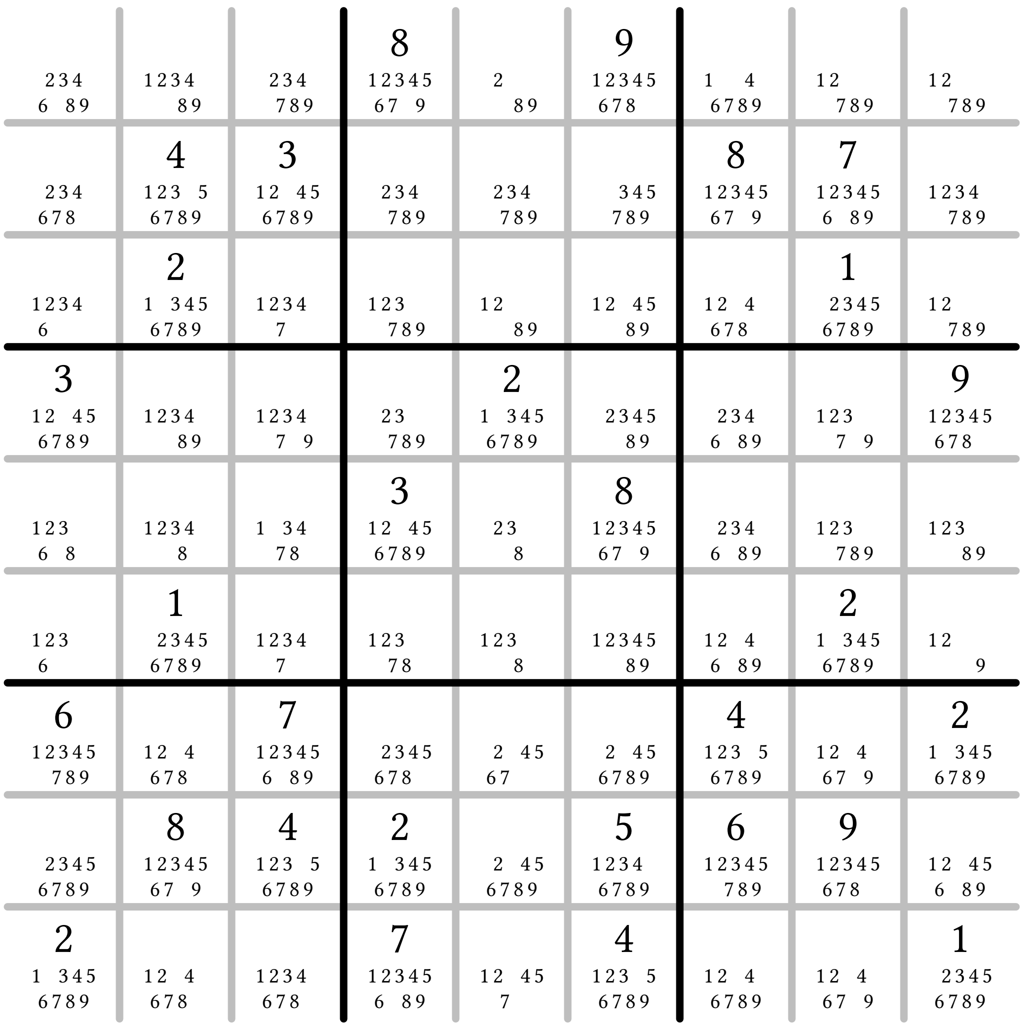
The simplest strategy to implement is called Naked Single. It waits until a cell's cannot has 8 digits and then declares that the cell's digit must be the one remaining digit. Here is the code.

```
 1 const SudokuNakedSingle = hiphop {
 2    fork ${iota.map(i => hiphop fork ${iota.map(j => hiphop {
 3       done: {
 4          loop {
 5             if (this[`cannot${i}${j}`].preval.size === digits.size - 1)
 6                break done
 7             yield;}}
 8       let must = digits.difference(this[`cannot${i}${j}`].preval);
 9       sustain ${`must${i}${j}`}(must);
10    })})}}
```

Line 2 sets up 81 parallel threads, one for each cell. Each thread then starts a loop with the potential of breaking out of it via the label `done:` on lines 3 and 4. Line 5 tests to see if there is only one digit that is not in the cannot set and, if so, escapes out of the loop. After escaping the loop, the code sustains that one remaining digit on the must signal for the cell on lines 9 and 10.

The use of `preval` on lines 5 and 8 avoids a causality error, following the temporal constraints discussed at the beginning of this section. It simply gives the value of the signal in the previous instant, instead of in the current instant, which introduces no causality constraint. If we had used `nowval`, it would cause a dependency where, in this thread we need to know the cannots to compute the musts but, as discussed in section 4.3, `MustThisCannot` and `MustOtherCannot` need to know the musts to compute the cannots. That is, both sides introduce an emission downstream of a use, one from musts to cannots and the other from cannots to musts. Either way individually is okay, but both together is not.

Naked Single brings the number of guesses we need to solve the puzzle from section 3 down from more than 15,000 to 17 guesses.



234 6 89	1234 89	234 789	**8** 12345 67 9	2 89	**9** 12345 678	1 4 6789	12 789	12 789
234 678	**4** 123 5 6789	**3** 12 45 6789	234 789	234 789	345 789	**8** 12345 67 9	**7** 12345 6 89	1234 789
1234 6	**2** 1 345 6789	1234 7	123 789	12 89	12 45 89	12 4 678	**1** 2345 6789	12 789
3 12 45 6789	1234 89	1234 7 9	23 789	**2** 1 345 6789	2345 89	234 6 89	123 7 9	**9** 12345 678
123 6 8	1234 8	1 34 78	**3** 12 45 6789	23 8	**8** 12345 67 9	234 6 89	123 789	123 89
123 6	**1** 2345 6789	1234 7	123 78	123 8	12345 89	12 4 6 89	**2** 1 345 6789	12 9
6 12345 789	12 4 678	**7** 12345 6 89	2345 678	2 45 67	2 45 6789	**4** 123 5 6789	12 4 67 9	**2** 1 345 6789
2345 6789	**8** 12345 67 9	**4** 123 5 6789	**2** 1 345 6789	2 45 6789	**5** 1234 6789	**6** 12345 789	**9** 12345 678	12 45 6 89
2 1 345 6789	12 4 678	1234 678	**7** 12345 6 89	12 45 7	**4** 123 5 6789	12 4 6789	12 4 67 9	**1** 2345 6789

Fig. 2. Example Sudoku Board, Showing Must (in a large font) and Cannot (in a small font)

5.2 The Hidden Single Strategy

The most common strategy used by people to solve Sudoku and the strategy discussed in the introduction to Sudoku from section 3 is called Hidden Single. It requires non-local information but it is an easier strategy for humans when solving the puzzle, because the non-local information does not need to be computed in its entirety. Instead, just looking at a few houses here and there can often be enough to make a successful deduction.

In its more general form, however, Hidden Single considers every house to see if there are any digits that can appear in only one cell in that house. The implementation, just below, looks at each house 9 different times, each time ignoring one cell in the house. Once the house and the specific cell to ignore are fixed, the implementation starts with the set of all of the digits and removes all of the digits that cannot appear in each cell, without considering the ignored cell. If that set of digits at the end of that process is a singleton, then that digit must appear in the ignored cell.

As this reasoning is pretty complex, let's return to the example deduction on the Sudoku board from section 3. Figure 2 shows that same Sudoku puzzle, but with the cannots in a small font below the givens. Consider all of the cells in the upper right box house, but ignore its lower-left cell. If we take the intersection of all of those cannots, we are left with the singleton set containing the digit 9, as 9 appears in every cannot except the lower-left cell, and no other digit also appears in every other cell's cannot. Therefore, the lower-left cell must contain a 9. Here is the code.

```
 1 const SudokuHiddenSingle = hiphop ${ForkHouseMap(coords => hiphop {
 2    loop {
 3       ${coords.map(ignored => hiphop {
 4          signal cans = digits combine (x, y) => x.intersection(y);
 5
 6          ${coords
 7             .filter(c => ignored.i !== c.i || ignored.j !== c.j)
 8             .map(c => hiphop {
 9                emit cans(this[`cannot${c.i}${c.j}`].preval);
10             })};
11
12          if (cans.nowval.size === 1) {
13             emit ${`must${ignored.i}${ignored.j}`}(cans.nowval);
14          }
15       })};
16       yield;
17    }})}
```

The code starts with a loop for every house on the board and then, on line 3, generates code for each cell in the house, binding the variable `ignored` to the cell that is to be ignored. The signal `cans` is a set-valued signal, as we have seen before, but this time the initial value is the entire set of digits and the combining function takes the intersection. Then, the generated code on lines 6 through 9 emits the value of cannot for every cell except the ignored cell and, if this process leaves a single value in `cans`, then that must be the value of the must signal for the ignored cell, so it is emitted on line 13. As with the previous section, if we use `nowval` (this time on line 9) we would have a causality error, so we again use `preval` to avoid the problem.

When using both Naked Single and Hidden Single, we solve our running example puzzle with only 2 guesses.

5.3 The Naked Pair Strategy

The final strategy we implement here is called Naked Pair. It looks at pairs of cells that are within a single house. If any such pair of cells have only two remaining possibilities, according to the cannots, then those two possibilities cannot be in any other cell in the same house. In other words, if the first two cells can only be either 1 or 2, then, even though we do not know which cell has a 1 and which cell has a 2, we know that no other cell in the house can be either a 1 or 2, as 1 and 2 must be placed in those two cells.

This example is interesting because it deduces cannots from other cannots. It never actually concludes which digit belongs in a cell but it does augment other strategies, strengthening ones that, without the help of Naked Pair, would not be able to deduce a must. Here is the code, which is also interesting because of the way it uses staging, generating many nested Esterel expressions, accounting for information that is known when generating the code (i.e. which cells are in which houses) and information that is not known when generating the code (i.e. the values of the cannots and the musts).

```
 1 const SudokuNakedPair = hiphop ${ForkHouseMap(coords => hiphop {
 2    fork ${coords.map(c => hiphop {
 3       loop {
 4          let c_cannot = this[`cannot${c.i}${c.j}`].preval;
 5          if (c_cannot.size === digits.size - 2) {
 6             ${coords
 7                .filter(d => c.i !== d.i || c.j !== d.j)
 8                .map(d => hiphop {
 9                   let d_cannot = this[`cannot${d.i}${d.j}`].preval;
10                   if (c_cannot.equal(d_cannot)) {
11                      fork ${coords
12                         .filter(e => (e.i !== c.i || e.j !== c.j)
13                                   && (e.i !== d.i || e.j !== d.j))
14                         .map(e => hiphop {
15                            sustain ${`cannot${e.i}${e.j}`}(
16                               digits.difference(c_cannot));
17                         })}}})}}
18          yield;
19       }})}}})}
```

It starts by creating parallel threads for each house and, within those, a second set of parallel threads for each cell in the house, calling the specific house we are in h and the cell c. Line 5 restricts our attention to the cells that have exactly two possible digits and then lines 6 through 9 create eight HipHop expressions that bind d to a cell in the house h that is not d. Thus, starting from line 9, we have a pair of cells that is a candidate to be a naked pair. On 9 we bind d_cannot to the value of the cannot set for d and then, on line 10 we compare it to the cannot set for c. If they are the same, we have found a naked pair.

Next, now that we know we have a naked pair, lines 12 through 14 generates another 7 pieces of code, one for each cell in the original house that is neither c nor d, binding each of these cells to e. Finally, since the code on lines 15 and 16 is specific to just this particular naked pair, we can simply sustain the information on the cannot for cell e, and all the other parallel threads will continue the loop started on line 3.

And finally, when using all three strategies, no guesses are needed to solve our example puzzle.

6 Running the HipHop Solver

Everything is now almost ready to solve Sudoku puzzles. We have at our disposal a HipHop machine ready to be fed with new puzzles (section 4.2 and section 4.3) and a JavaScript driver that controls it to solve them (section 4.1). We are only missing the main JavaScript program that will parse a puzzle, create the HipHop machine, and start the driver.

A puzzle is implemented as JavaScript string where empty cells are denoted with the "." character and givens by their actual values, as shown on the left. On the right the parseBoard JavaScript function that parses a string of that form into an array of HipHop signals.

```
const expert = `
  4..6.....
  ..2.3....
  .....9827
  8..41....
  9.......5
  .6.....7.
  .3....4.6
  ....962..
  .9.....5.`;
```

```
const parseBoard = (board) => {
   const rows = board.split("\n")
      .filter(l => !l.match(/^[ \t]*$/))
      .map(s => s.trim());
   const givens = {};
   iota.forEach(i => iota.forEach(j => {
      if (rows[j][i] !== ".") {
         givens[`must${i}${j}`] =
            new Set([parseInt(rows[j][i])]);
      }}));
   return givens;
}
```

All that remains is to assemble our components, the `driver` (section 4.1), the `SudokuMachine` (section 4.2), the three strategies in section 5, and the board parser `parseBoard` to solve a puzzle.[5]

```
function runExpertToCompletion() {
   const prog = SudokuMachine([ // create a HipHop program solving
      SudokuNakedPair,          // Sudoku problems, using three
      SudokuNakedSingle,        // strategies, each executing
      SudokuHiddenSingle        // in parallel.
   ]);
   const mach = new hh.ReactiveMachine(prog, {sweep: false});
   return driver(mach, parseBoard(boards9x9.expert));
}
```

After 15 reactions and without any guess this produces the solution to the puzzle.

7 Do We *Really* Need Esterel, After All?

Now that we have shown how to take advantage of Esterel's parallel composition and thread abortion to combine the implementations of three different deduction strategies into a single solver, one might wonder if we can use the same structure in a purely JavaScript-based Sudoku solver. After all, could we not implement the code that performs deduction in JavaScript, and simply adapt it to return its deductions instead of imperatively updating variables with the results of the deductions? If we could make that adjustment, the solver could iterate the strategies until they yield no new information and, as before, make guesses when that happens.

The answer is no, at least not without significantly adjusting the way the strategies interact with their context. Concretely, look back at the `SudokuNakedSingle` strategy in section 5.1. This strategy never terminates but, even worse, important state in the strategy is encoded in its control. Specifically, each of the 81 concurrent threads might be in the `loop` on line 4 or might be in the `sustain` on line 9. This amounts to 81 booleans that have to be explicitly reified into a data structure and then managed, if the strategy were to completely yield control at the end of each round of deduction. And, of course, each new strategy comes with its own data structure to manage.

Another approach would be to use JavaScript generators or to CPS convert the strategies, but this is an inversion of control which boils down to a Sudoku-specific green threading strategy.

[5] Passing `{sweep: false}` disables an optimization that, for our Sudoku solver, turns out to be a pessimization.

Of course, beyond the trouble of programming with JavaScript generators or in CPS, structuring the code in that manner carries with it all of the trouble of programming with concurrency and mutable state that Esterel lets us avoid, bringing us full circle back to the introduction.

8 Conclusion

In this paper we make the argument that the essential principles that underlie functional programming are much more general than simply programming with pure functions might suggest. Hughes (1998)'s seminal essay *Why Functional Programming Matters* tackles a deeper question that is also relevant here, offering an argument about what is important about these essential principles. Hughes argues that the value in functional programming is not merely the power to reason about individual program fragments, but it comes from powerful glue. That is, the ability to divide a computation up into smaller pieces is a well-understood, central aspect to every well-structured program and supported by every well-designed programming language—functions are just one way to do that. But, as Hughes argues, powerful forms of glue that combine the pieces back into a complete program are also essential, and powerful glue is where functional programming shines.

As we have shown in this paper, Esterel too supports powerful glue: parallel composition. Just as in real life where a team of people can accomplish a task more accurately by dividing up the work (although Esterel compilers generate efficient code, even circuits, we have kept our focus on understanding and organizing code in this paper), an Esterel programmer can divide up different aspects of a program into separate pieces that can be reasoned about separately and then composed by simply putting them in parallel with each other. The reason Esterel's version of this glue is so powerful is precisely its determinism. No matter what happens in the two parallel threads, the program either aborts with a causality error or produces one single result. There are no racy executions, no deadlocks, and certainly no out-of-thin-air reads.

More importantly than the individual forms of glue supported by functional programming and Esterel-style imperative programming separately, however, this pearl demonstrates that they combine into a synergistic whole. The example of the Sudoku solver shows that imperative Esterel programming and functional JavaScript programming combine nicely, enabling us to achieve well-structured, maintainable code in a way that neither could on its own.

Acknowledgments

A special thanks to Timothy Bourke who offered an insightful question one evening at Synchron 2023; without that question and the discussions that followed, this paper would not exist. Thanks also to Christos Dimoulas, Matthias Felleisen, Alan Schmitt, Shu-Hung You, and the ICFP reviewers for thoughtful and helpful comments on earlier drafts.

Data Availability

To experiment with the code from this article, please see the artifact (Serrano and Findler 2024). It includes all of the code from the paper as well as a few exercises that serve as an active introduction to programming in HipHop.

A Two Sudoku Puzzles

We invite the reader to play with the strategies from section 5 via the puzzles in figures 3 and 4.

		5				4		
			1		3			
4								9
	6						2	
			9	2	5			
3								1
		2		5		9		
			3		4			
	7			6			3	

Fig. 3. This puzzle requires only deductions like those described in section 5.2 to solve. It is from the Puzzle Bank (McLean 2020) and has a difficulty ranking of 1.2, on a scale from 1 to 10.

12345 6789	12345 6789	12345 6789	9 12345 6789	5 12345 6789	2 12345 6789	12345 6789	12345 6789	12345 6789
12345 6789	12345 6789	12345 6789	12345 6789	12345 6789	12345 6789	12345 6789	12345 6789	12345 6789
12345 6789	6 12345 6789	1 12345 6789	12345 6789	4 12345 6789	12345 6789	5 12345 6789	2 12345 6789	12345 6789
12345 6789	12345 6789	12345 6789	1 12345 6789	3 12345 6789	6 12345 6789	12345 6789	12345 6789	12345 6789
1 12345 6789	12345 6789	12345 6789	12345 6789	2 12345 6789	12345 6789	12345 6789	12345 6789	7 12345 6789
12345 6789	8 12345 6789	12345 6789	12345 6789	12345 6789	12345 6789	12345 6789	6 12345 6789	12345 6789
2 12345 6789	12345 6789	3 12345 6789	8 12345 6789	12345 6789	7 12345 6789	4 12345 6789	12345 6789	1 12345 6789
5 12345 6789	12345 6789	12345 6789	12345 6789	12345 6789	12345 6789	12345 6789	12345 6789	3 12345 6789
12345 6789	12345 6789	12345 6789	12345 6789	9 12345 6789	12345 6789	12345 6789	12345 6789	12345 6789

Fig. 4. Solving this puzzle requires at least one deduction from each the strategies in the subsections of section 5, but no guessing. It is also from the Puzzle Bank, where it has a difficulty rating of 2.0. The small numbers in the bottom of each cell are meant to help you track the value of the cannots, crossing out numbers as you learn they cannot appear.

Bibliography

Gérard Berry. The Constructive Semantics of Pure Esterel Draft Version 3. https://www-sop.inria.fr/members/Gerard.Berry/Papers/EsterelConstructiveBook.pdf (accessed June 22, 2024), 2002.

Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming* 19(2), 1992. doi:10.1016/0167-6423(92)90005-V

Gérard Berry and Manuel Serrano. Hop and HipHop: Multitier Web Orchestration. In *Proc. International Conference on Distributed Computing and Internet Technology*, 2014. doi:10.1007/978-3-319-04483-5_1

Gérard Berry and Manuel Serrano. HipHop.js: (A)Synchronous reactive web programming. In *Proc. ACM Conference on Programming Language Design and Implementation*, 2020. doi:10.1145/3385412.3385984

Frédéric Boussinot. Reactive C: An extension of C to program reactive systems. *Software: Practice and Experience* 21(4), 1991. doi:10.1002/spe.4380210406

Frédéric Boussinot and Robert de Simone. The SL Synchronous Language. *IEEE Transactions on Software Engineering* 22(4), 1996. doi:10.1109/32.491649

Corrado Böhm. DIGITAL COMPUTERS: On encoding logical-mathematical formulas using the machine itself during program conception. http://www.itu.dk/~sestoft/boehmthesis/, 1954.

Howard Garns. Dell Pencil Puzzles & Word Games. https://imgur.com/a/IH8mL, 1979.

N. Halbwachs and F. Maraninchi. On the symbolic analysis of combinational loops in circuits and synchronous programs. In *Proc. Euromicro'95*, 1995.

Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE* 79(9), 1991. doi:10.1109/5.97300

Bernhard Hobiger. HoDoKu. https://hodoku.sourceforge.net/en/techniques.php, 2008.

John Hughes. Why functional programming matters. *The Computer Journal* 32(2), 1998. doi:10.1093/comjnl/32.2.98

Jayanth Krishnamurthy and Manuel Serrano. Causality Error Tracing in HipHop.Js. In *Proc. 23rd International Symposium on Principles and Practice of Declarative Programming (PPDP)*, 2021. doi:10.1145/3479394.3479408

L Mandel, C Pasteur, and M Pouzet. ReactiveML, Ten Years Later. In *Proc. of the 17th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP'15)*, 2015. doi:10.1145/2790449.2790509

L Mandel and M Pouzet. ReactiveML, a reactive extension to ML. In *Proc. of the 7th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, 2005. doi:10.1145/1069774.1069782

John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM* 3(4), 1960. doi:10.1145/367177.367199

Grant McLean. Sudoku Exchange "Puzzle Bank". https://sudokuexchange.com/puzzle-bank, 2020.

Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. Compiling Esterel. Springer, 2007. doi:10.1007/978-0-387-70628-3

Klaus Schneider, Jens Brandt, Tobias Schuele, and Thomas Tuerk. Maximal causality analysis. In *Proc. Fifth International Conference on Application of Concurrency to System Design (ACSD'05)*, 2005. doi:10.1109/ACSD.2005.24

Ellen M. Sentovich. Quick conservative causality analysis. In *Proc. Tenth International Symposium on System Synthesis (Cat. No. 97TB100114)*, 1997. doi:10.1109/ISSS.1997.621669

Manuel Serrano and Robert Bruce Findler. The Functional, the Imperative, and the Sudoku: Getting Good, Bad, and Ugly to Get Along (Artifact). https://github.com/manuel-serrano/icfp2024-sudoku, 2024. doi:10.5281/zenodo.12792675

Thomas R. Shiple, Gérard Berry, and Herve Touati. Constructive analysis of cyclic circuits. In *Proc. ED&TC European Design and Test Conference*, 1996. https://dl.acm.org/doi/10.5555/787259.787602

Received 2024-02-28; accepted 2024-06-18