Optimization of Synthesized Functional Reactive Programs
A Senior Project in Computer Science and Mathematics

Haohang Xu
Advised by Professor Ruzica Piskac
May 4th, 2017

Abstract

Reactive systems are applicable in many useful fields. However, reactive systems are difficult to design and implement correctly by hand. A tool to synthesize functional reactive programs from a specification describing the behavior of the program has previously been implemented. However, the programs produced by this tool have been shown to run much more slowly than equivalent handwritten programs. This project examines the reasons for the slow runtime and seeks to optimize the synthesized programs.

1 Background and Motivation

1.1 Reactive Systems

Reactive systems are a broad class of computer systems whose defining element is the continued interaction between the system and its environment. Consider an example of the computation

\[ A = B + C \]

In a non-reactive system, \( B \) and \( C \) would be constant values. The computation would be evaluated once to obtain the value of \( A \), and that value does not change.
However, in a reactive system, rather than being single values, $B$ and $C$ might be streams of time-varying values. Then, in a reactive system which evaluates the computation $A = B + C$, $A$ would take on different values at different times, and be automatically updated when the values of $B$ and $C$ change.

### 1.2 Functional Reactive Programming

Functional Reactive Programming (FRP), is a programming paradigm for expressing reactive, streaming computations in a functional language. It creates a clean modularization between the control structures and the data transformations in a reactive system[2].

The fundamental idea of FRP is to extend the classic building blocks of functional programming with the abstraction of a *signal* to describe time-varying values:

$$\text{Signal } a :: \text{Time} \rightarrow a$$

$$\text{SF Input Output} :: \text{Signal Input} \rightarrow \text{Signal Output}$$

which produces values of some arbitrary type $a$ over time. The type $a$ can be an input from the world, such as the current position of the mouse, or an output type, such as some text that should be rendered to the screen. Signals are also used internally to manipulate values over time, such as when the position of the mouse should be rendered to the screen.

FRP programs can be exceptionally efficient. For example, a network controller recently implemented as an FRP program on a multicore processor outperforms any other such controller existing today [10]. FRP programs also inherit many desirable advantages of functional programming in building large and complex systems. First-class functions in FRP allow for easy composability and scalability. Immutability and referential transparency ensure focused, side-effect free functions. These qualities allow for more correct, responsive, and scalable code.

### 1.3 Program Synthesis

Reactive systems are important because of their many applications, such as in embedded devices [3], robots [4], hardware circuits [5], GUIs [1], and interactive multimedia [9]. Reactive systems are also considered among the most difficult types of systems to design correctly [7], and writing a reactive program is a highly error-prone process. In an ideal world,
we would only describe the desired properties of the program in a standard specification language, such as a temporal logic, and let the computer generate a correct-by-construction program from that specification.

The immediate advantage of synthesis over manual programming is that if the synthesis succeeds, there is a guarantee that the constructed program satisfies the specification. In this way, synthesis can also serve as a check on manually written specifications, where a failed synthesis may indicate an incorrectly written specification. Finally, synthesis is desirable as a way to make programs easier to inspect and understand, even by readers who may not be familiar with the particular FRP libraries or syntax.

Previous tools for reactive systems has been focused on finite-state implementations. This focus is a serious limitation because even simple programming tasks, such as implementing a counter, quickly become exceedingly expensive, because each counter value is treated as a separate state.

Previously, Professor Piskac’s group developed a new framework for synthesis of functional reactive systems by defining a new logic, called Temporal Stream Logic (TSL), that can easily specify control flow properties over arbitrarily complex data transformations. TSL is based on linear-time temporal logic (LTL), the most commonly used specification language for reactive programs. TSL extends LTL with a specification of how output signals are computed from input signals. This specification allows easy specification of the behavior of reactive systems [6].

In the current system for FRP synthesis from TSL specifications, the user provides a TSL specification over a set of predicate and function terms. This specification is then synthesized into a Term Annotated Mealy Machine (TAM), which is then translated into an FRP using the Haskell Yampa library\(^1\).

While the current framework successfully synthesizes FRP programs, the synthesized programs exhibit a much slower runtime than equivalent, handwritten programs. In this project, I examine this problem and seek to optimize the synthesized FRP programs.

\(^{1}\)https://wiki.haskell.org/Yampa
2 AIGER Translation

The current framework for synthesizing FRP programs takes a specification in a TSL specification, reduces it to LTL, uses an LTL to TAM synthesis tool to create the TAM, and finally translates the TAM into an FRP. Semantically, the TAM represents the program as a series of state transitions in the form of a graph. We hypothesized that the intermediate representation of the program as a graph posed a bottleneck in the runtime of the final program. To test this hypothesis, we created a new synthesis framework that instead synthesizes an intermediate representation in the form of a boolean circuit.

2.1 AIGER Specification

AIGER is a format, library and set of utilities for And-Inverter Graphs (AIGs). AIGER specifies the logic of a program as a boolean circuit. The AIGER format can be used to formulate structural SAT and model checking problems of both sequential and combinational circuits.

2.2 Implementation

We implemented a tool to translate a program in the AIGER format into an FRP program using the Haskell Yampa library. To obtain the AIGER program from the original TSL specification, we used the acacia4aiger tool along with our current TSL to LTL reduction tool. We combined these three separate steps into a complete tool which executes the entire synthesis procedure, taking a TSL specification into an runnable FRP program. Like the TAM-synthesized FRP, this program imports the Terms used by the program in a separate file. The implementation of these Terms is not needed until the final compilation.

2.3 Benchmarking

We benchmarked the AIGER-synthesized programs against the TAM-synthesized programs and the handwritten programs. While the LTL AIGER synthesis times were much faster than the TAM synthesis time, the unaltered FRP programs synthesized using the AIGER intermediate ran significantly slower than the unaltered FRP programs synthesized

\footnote{http://fmv.jku.at/aiger/FORMAT}

\footnote{https://github.com/gaperez64/acacia4aiger}
using the TAM intermediate by an order of magnitude.

Part of this inefficiency was immediately obvious from the redundancy in the synthesized AIGER circuit. This redundancy appeared in the form of literals with equivalent inputs as well as unnecessarily complicated circuit elements. These redundancies appear as acacia4aiger is written to find a solution in a short time, even if the solution is not the most optimal.

In order to obtain a better benchmark for the potential performance of an optimized AIGER program, we applied circuit optimization rules by hand to simplify the AIGER intermediate before synthesizing the FRP program. However, even with the optimizations, the AIGER-synthesized programs ran significantly slower than the TAM-synthesized programs and the handwritten programs. Detailed results are listed in Figure 1.

<table>
<thead>
<tr>
<th>specification</th>
<th>steps</th>
<th>TAM</th>
<th>AIGER</th>
<th>AIGER (opt)</th>
<th>handwritten</th>
</tr>
</thead>
<tbody>
<tr>
<td>escalator2</td>
<td>10000</td>
<td>0.028</td>
<td>0.088</td>
<td>0.028</td>
<td>0.020</td>
</tr>
<tr>
<td></td>
<td>100000</td>
<td>0.272</td>
<td>1.212</td>
<td>0.348</td>
<td>0.196</td>
</tr>
<tr>
<td></td>
<td>1000000</td>
<td>2.884</td>
<td>11.408</td>
<td>3.492</td>
<td>2.096</td>
</tr>
<tr>
<td>escalator3</td>
<td>10000</td>
<td>0.044</td>
<td>0.304</td>
<td>-</td>
<td>0.028</td>
</tr>
<tr>
<td></td>
<td>100000</td>
<td>0.452</td>
<td>2.556</td>
<td>-</td>
<td>0.276</td>
</tr>
<tr>
<td></td>
<td>1000000</td>
<td>4.536</td>
<td>31.516</td>
<td>-</td>
<td>2.804</td>
</tr>
</tbody>
</table>

Figure 1: Runtime of AIGER programs

3 TAM-Synthesized Program Optimization

Turning back to the original TAM-synthesized program, we examined the synthesized programs in comparison to the handwritten programs in order to determine reasons for the relatively slow runtime performance of the synthesized programs.

The TAM that is synthesized from the TSL specification contains a behavior for each combination of predicate terms. Thus, if the number of predicates in our program is \( N \), we need \( 2^N \) cases just to cover every combination of the predicates. The handwritten programs are able to avoid such extensive conditions due to semantic understanding of the interaction
between the predicates. The optimizations in this section seek to reduce the number of computations necessary to evaluate the next state by applying similar principles.

### 3.1 Computation Redundancy

The original synthesized program is comprised of a list of case statements against which the state of the environment is evaluated. Each case will be evaluated until one case is found to be true, upon which the new state will be computed. While straightforward, this implementation results in many unnecessary computations. In particular, instead of writing the cases as a list, we can rewrite them as a series of nested conditions.

```plaintext
stateTrans ((bottom, top), (steps), state) = if
  | (exitevent top) && (not $ enterevent bottom)
    -> ((Stop), 0)
  | ((exitevent top) && (enterevent bottom))
    | ((not $ exitevent top) && (not $ enterevent bottom))
      -> (steps, 0)
    | (not $ exitevent top) && (enterevent bottom)
      -> ((Up), 0)
then
```

Figure 2:

Figure 3:

Figure 2: Original synthesized state transition code for smart escalator.

Figure 3: State transition of smart escalator expanded into nested conditions.

Figure 2 depicts the original synthesized code for a smart escalator which stops when a user gets off the escalator. Figure 3 depicts the expanded version of the same code.

We can think of the latter structure as a binary tree. Structuring the code in a tree not only guarantees no redundancy in computations, but it also guarantees a much fewer number of computations needed to compute the next state. In particular, while the original structure could require $N \times 2^N$ computations for an environment that satisfied the very last case in the list, the modified structure guarantees that the next state will be found after computing $N$ predicates.
3.2 Dependent Clauses

Another way to reduce the number of computations is to eliminate redundancy that arises to do dependent predicates. For example, if we had two predicates such that

\[ A \implies B \]

then

\[ \text{if } A \text{ and } B \iff \text{if } A \]

Hence, if we append our synthesis to include more information about the relationships between the terms, we can eliminate redundancies and improve efficiency.

To implement this, we encode the conditions

\[ A \implies B \]
\[ A \implies \neg B \]

where \( A \) and \( B \) can be single conditions or conjunctions and disjunctions of conditions. We encode these conditions by hand based on what we know about the input signals and how the program should work. Once encoded, we traverse our program tree and prune the subtrees that are guaranteed to never be traversed due to conditions that are impossible to fulfill. This occurs when the conditions before a particular condition fully determine what that condition must be.

3.3 Tree Optimization

In addition to pruning based on dependent clauses, we can take advantage of the tree structure of our code to make further optimizations. The principle we apply in all of these optimizations is to try to reduce the heights of path to more common state to reduce the average number of computations needed to compute the next state.

Subtree Reduction

Pruning often results in subtrees where all leaves have the same next state. In this case, we can reduce the subtree to a single leaf, thereby also reducing the number of computations we need to reach the next state for environments that satisfy cases in that subtree. For example:
Extending this idea, we can also prune out any internal node whose two children are identical. For example:

Figure 4: Demonstration of subtree reduction.

In both these examples, we are able to eliminate conditions and reduce the height of leaves in the tree, thus reducing the number of computations necessary.

Predicate Decisiveness

Without deeper knowledge about how often we can expect to hit each case, we can make observations about how decisive a particular predicate is. In particular, if a single condition determines the next state in many cases, we want to prioritize that condition to improve the average number of computations. For example:
We implement reordering based on predicate decisiveness using the ID3 algorithm [8]. With this algorithm, we start out with a set of all of the leaves of our program tree as well as all of the predicates that are used in our tree. At each step, we greedily select the predicate that results in the greatest information gain, where we calculate information gain by subtracting the weighted sum of the entropies of the two sets partitioned by the predicate from the entropy of the entire set. Figures 7 and 8 show the change in the code after applying this algorithm. From this example, it is clear that not only does predicate reordering help with the number of computations, but it also makes the code much more readable and manageable.

### 3.4 State Prediction

We can also improve our decision tree by looking at the edges between states in our code. We model each state as a node in a directed graph, with edges representing the relationship between the current state and the next state. Assuming a uniform distribution of cases, state nodes that have more edges directed towards them will occur more frequently. Based on the frequencies, we can apply a weight to the leaves in our tree to improve our determination of the decisiveness of a condition.

### 3.5 Comparison with Handwritten Programs

Beyond optimizations, we can identify some differences between the handwritten and synthesized programs.
Figure 7: Synthesized FRP before applying reordering by predicate decisiveness.

One difference is that the handwritten programs do not exhibit the same separation of data and control that we see in the synthesized programs which is desirable of FRP programs. Part of the cause of this difference is that the handwritten programs use different predicate and update function terms than the synthesized programs, because while the synthesized program must use the terms in the specification, we can handwrite a program based on our understanding of the semantics of the program without necessarily enforcing the usage of isolated terms.

For example, for our escalator 2 benchmark, which runs in one direction until there are no users left on the escalator, Figure 9 shows the original handwritten program, compared with the same program with the same terms as the synthesized program, shown in Figure 10.
Figure 9: Handwritten FRP for escalator that handles multiple people in one direction.

Figure 10: Handwritten FRP for escalator that handles multiple people in one direction, using the same terms as the synthesis specification.

Comparing these handwritten programs to our optimized synthesized program in Figure 8. It is clear that the programs employ different structures, even when changing the handwritten program to use the same terms as the synthesized programs. We use the handwritten program with the same terms as the synthesized program to gain more information from benchmarking. Eliminating the difference in function terms allows us to better isolate and study the effect of the structure of the FRP on the runtime of the program. Additional examples may be found in the Appendix.

3.6 Benchmarking

Figure 11 shows the total number of computations that were evaluated in running each of two benchmark programs. We can see that expanding the conditions into a nested structure alone decreased the number of computations significantly (as expected). The number of computations eliminated as a result of optimizing through pruning was more significant in the more complex escalator3 benchmark, but was able to reduce the number of computations in both benchmark programs.

Figure 12 shows the runtime of the TAM synthesized programs after being optimized
through pruning, expanding into a nested form, and both pruning and expanding. We compared this optimized program against the handwritten program with the same terms. As expected, the expansion significantly decreased the runtime. However, the added effect of pruning after expanding was very small, suggesting that the bottleneck in the runtime may be elsewhere.

<table>
<thead>
<tr>
<th>specification</th>
<th>steps</th>
<th>original</th>
<th>handwritten</th>
<th>opt</th>
<th>expanded</th>
<th>expanded + opt</th>
</tr>
</thead>
<tbody>
<tr>
<td>escalator2</td>
<td>10000</td>
<td>62171</td>
<td>53907</td>
<td>34969</td>
<td>29979</td>
<td></td>
</tr>
<tr>
<td></td>
<td>100000</td>
<td>624429</td>
<td>541509</td>
<td>350181</td>
<td>300228</td>
<td></td>
</tr>
<tr>
<td>escalator3</td>
<td>10000</td>
<td>1315487</td>
<td>511603</td>
<td>74994</td>
<td>54590</td>
<td></td>
</tr>
<tr>
<td></td>
<td>100000</td>
<td>13157997</td>
<td>5116603</td>
<td>749994</td>
<td>545840</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 11:** Effect of optimizations on total number of computations

These benchmark times show that for some programs, our synthesized programs with optimizations run faster than the handwritten programs, while for some of the other programs, our synthesized programs run slower.

<table>
<thead>
<tr>
<th>specification</th>
<th>steps</th>
<th>original</th>
<th>handwritten</th>
<th>opt</th>
<th>expanded</th>
<th>expanded + opt</th>
</tr>
</thead>
<tbody>
<tr>
<td>escalator2</td>
<td>10000</td>
<td>0.028</td>
<td>0.020</td>
<td>0.024</td>
<td>0.028</td>
<td>0.028</td>
</tr>
<tr>
<td></td>
<td>100000</td>
<td>0.348</td>
<td>0.196</td>
<td>0.264</td>
<td>0.272</td>
<td>0.260</td>
</tr>
<tr>
<td></td>
<td>1000000</td>
<td>2.884</td>
<td>2.096</td>
<td>2.860</td>
<td>2.720</td>
<td>2.668</td>
</tr>
<tr>
<td>escalator3</td>
<td>10000</td>
<td>0.048</td>
<td>0.028</td>
<td>0.040</td>
<td>0.020</td>
<td>0.028</td>
</tr>
<tr>
<td></td>
<td>100000</td>
<td>0.464</td>
<td>0.284</td>
<td>0.356</td>
<td>0.252</td>
<td>0.248</td>
</tr>
<tr>
<td></td>
<td>1000000</td>
<td>4.580</td>
<td>2.808</td>
<td>3.524</td>
<td>2.560</td>
<td>2.516</td>
</tr>
</tbody>
</table>

**Figure 12:** Effect of TAM optimizations on runtime

4 Conclusions and Future Work

From our benchmarks, it is clear that our optimizations do result in a significant speedup in the runtime of the synthesized programs. Additionally, it seems that our optimizations on the Mealy machine synthesis at a code level resulted in a program that looks like a Binary Decision Diagram (BDD). While current synthesis tools favor synthesizing BDDs in less time at the cost of a larger program, future work may be done to investigate the possibility of
synthesizing directly to a BDD and its effects on runtime and program structure.

Our implementation of FRP synthesis treats time as discrete. Further research can be done to investigate the possibility of extending to programs with continuous time domains. Our current synthesis also targets a restricted set of arrows called Causal Commutative Arrows (CCA). It may be interesting to investigate the possibility of lifting the commutativity condition for greater expressiveness of synthesized programs.

Given a particular TSL specification, the question of whether or not this specification is realizable is undecidable. Hence, the synthesis framework is a nondeterministic process. Analyzing the specifications for which synthesis does not succeed may offer interesting results. Furthermore, we may be able to optimize on the decidability problem by giving additional parameters in the specification.

5 Acknowledgements

Immeasurable thanks to Mark Santolucito for guiding me during every step of my senior project. Thanks also to my adviser, Prof. Ruzica Piskac, for her help and support.
References


A Appendix

A.1 Handwritten Programs

```
esc3 :: SF (Top, Bottom) (Int, Dir)
esc3 = proc (b, t) -> do
  rec
    userChange <- arr f -< (t, b)
    u <- init 0 -< userChange u
    d <- init Stop -< nextd
    nextd <- arr g -< (u, d, t, b)
    returnA -< (u, d)
  where
    g (u, d, t, b) = if
      | enterevent t -> movedown
      | enterevent b -> moveup
      | eqzero u -> stop
      | True -> d
    f (t, b) = if
      | enterevent t -> if
      | | True -> inc
      | | False -> dec
      | enterevent b -> inc
      | | True -> dec
      | | False -> id
```

**Figure 13:** Handwritten FRP for escalator that handles multiple people in both directions.

```
esc3 :: SF (Top, Bottom) (Int, Dir)
esc3 = proc (b, t) -> do
  rec
    userChange <- arr f -< (t, b)
    u <- init 0 -< userChange u
    d <- init Stop -< nextd
    nextd <- arr g -< (u, d, t, b)
    returnA -< (u, d)
  where
    g (u, d, t, b) = if
      | t == Event Enter -> Down
      | b == Event Enter -> Up
      | u == 0 -> Stop
      | True -> d
    f (t, b) = if
      | t == Event Enter -> if
      | | b == Event Exit -> id
      | | True -> inc
      | | False -> dec
      | t == Event Exit -> if
      | | b == Event Enter -> id
      | | True -> dec
      | | False -> id
```

**Figure 14:** Handwritten FRP for escalator that handles multiple people in both directions, using the same terms as the synthesis specification.
Figure 15: Optimized Synthesized FRP for escalator handling multiple people in both directions.