

Lowering: A Static Optimization Technique for Transparent Functional Reactivity^{*}

Kimberley Burchett Gregory H. Cooper Shriram Krishnamurthi

Department of Computer Science
Brown University
{kburchet, greg, sk}@cs.brown.edu

Abstract

Functional Reactive Programming (FRP) extends traditional functional programming with dataflow evaluation, making it possible to write interactive programs in a declarative style. An FRP language creates a dynamic graph of data dependencies and reacts to changes by propagating updates through the graph. In a *transparent* FRP language, the primitive operators are implicitly *lifted*, so they construct graph nodes when they are applied to time-varying values. This model has some attractive properties, but it tends to produce a large graph that is costly to maintain. In this paper, we develop a transformation we call *lowering*, which improves performance by reducing the size of the graph. We present a static analysis that guides the sound application of this optimization, and we present benchmark results that demonstrate dramatic improvements in both speed and memory usage for real programs.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—dataflow languages

General Terms Languages, performance

Keywords Reactive programming, FRP, FrTime, lifting, lowering, functional programming, Scheme, static analysis, optimization

1. Introduction

Functional Reactive Programming (FRP) [8, 15, 20] is a modern extension of traditional dataflow concepts to a dynamic and higher order setting. FRP programs appear to be stateless functional specifications, but they compute over values that may change over time. The language reacts to changes by automatically recomputing and propagating values through the program, according to its data dependencies. As a result, FRP offers the expressive power of functional programming in a reactive setting, taking care of the onerous task of propagating changes and ensuring the consistency of values.

^{*}Work partially supported by NSF grant CCF-0447509.

For several years now, we have been developing FrTime [5], an embedding of FRP in DrScheme [9]. FrTime differs in several ways from, and offers certain advantages over, previous FRP implementations. Evaluating an expression in FrTime results in the construction of a fragment of *dataflow graph* that precisely captures the expressions’s data dependencies. A *dataflow engine* reacts to events in the environment by traversing the graph and recomputing the values of affected nodes.

FrTime induces construction of the dataflow graph by redefining operations through an implicit *lifting* transformation. Lifting takes a function that operates on constant values and produces a new function that performs the same operation on time-varying values. Each time the program applies a lifted function to time-varying arguments, it builds a new node and connects it to the nodes representing the arguments. Core Scheme syntactic forms are redefined to extend the graph when used with time-varying values.

Dynamic dataflow graph construction permits incremental development of reactive programs in, for instance, a read-eval-print loop (REPL). The implicit lifting allows programmers to write in exactly the same syntax as a purely functional subset of Scheme. Because lifting is conservative, FrTime programs can reuse Scheme code without any syntactic changes, a process we call *transparent reactivity*.

Unfortunately, this implicit graph construction can be very inefficient. Every application of a lifted function may create a new dataflow node, whose construction and maintenance consume significant amounts of time and space. As a result, large legacy libraries imported into FrTime may be slowed down by two orders of magnitude or more. In one experiment, for example, we attempted to reuse an image library from PLT Slideshow [10], but the result was unusably slow.

This paper presents an optimization technique designed to eliminate some of the inefficiency associated with FrTime’s evaluation model, while still giving programmers the same notion of transparent reactivity. The technique works by collapsing regions of the dataflow graph into individual nodes. This moves computation from the dataflow model back to traditional call-by-value, which the runtime system executes much more efficiently. Because this technique undoes the process of lifting, we call it *lowering*. Of course, lowering must not alter the semantics of the original program or sacrifice the advantages of FrTime’s evaluation strategy. We present a static analysis that determines when the optimizer can safely lower an expression. Our lowering analysis and its implementation yield a significant reduction in time and space usage for real programs.

```

(define (planet dist speed)
  (make-point
   (* dist (cos (/ speed (current-seconds))))
   (* dist (sin (/ speed (current-seconds))))))
(define persephone (planet 97 0.0166))
(define whitefall (planet 148 0.0273))
(define dist-between-planets
  (distance persephone whitefall))
(define updater
  (new timer
   [interval 10]
   [callback
    (lambda ()
     (set! persephone (planet 97 0.0166))
     (set! whitefall (planet 148 0.0273))
     (set! dist-between-planets
      (distance persephone whitefall))))))

```

Figure 1. A program to calculate distance between two planets.

2. Background

2.1 Functional Reactive Programming

Programs written in a stateless, declarative style are often easier to understand and more likely to be correct than programs written in an imperative style. However, substantial difficulties arise when trying to integrate a declarative program with the arrival of asynchronous I/O events from the outside world. Similarly, the use of mutable state internal to a program tends to undermine the advantages of declarative programming. We view these problems as special cases of a more general problem: that of programming over *values that change over time*.

For example, consider a simple program that calculates the distance between two planets orbiting a sun along circular orbits at different rates; we show such a program in Fig. 1. The first six lines compute this distance for a given point in time. The complexity increases, however, when we try to keep this value up-to-date as time changes. In traditional languages, programmers register callbacks that respond to change by re-evaluating dependent expressions and imperatively updating variables. This explicit mutation, required by the callback mechanism, destroys the functional model and introduces subtle opportunities for error.

The problem is more difficult than it first appears. For example, if the call to *current-seconds* in Figure 1 merely returns the current system time, then the program may compute a planet’s *x* and *y* coordinates with respect to *different times*. Such a glitch can be fixed by changing the *planet* function so that it calls *current-seconds* only once and stores the result in a temporary local variable. However, even that does not ensure that the relative positions of *different* planets will be consistent, since their computation occurs in different function calls. Fixing such interprocedural inconsistencies requires more invasive changes.

Functional Reactive Programming (FRP) [8, 15, 20] is an attempt to bridge the gap between declarative programming and time-varying values. In FRP each output value is defined as a function of its inputs, but the inputs (and hence the outputs) may change dynamically. Whenever an input value

```

(define (planet dist speed)
  (make-point
   (* dist (cos (/ speed seconds))))
   (* dist (sin (/ speed seconds))))))
(define persephone (planet 97 0.0166))
(define whitefall (planet 148 0.0273))
(define dist-between-planets
  (distance persephone whitefall))

```

Figure 2. Functional reactive program to calculate distance between two planets.

changes, the language automatically updates all dependent output values. These time-varying values are called *signals*.

To redo the preceding example with signals, we reuse the definition of the distance function almost verbatim (Fig. 2). The main difference is that we replace the call to *current-seconds* with a reference to the signal *seconds* (boxed). As *seconds* changes, the language automatically recomputes the result. We no longer need callbacks because the language responds to changes and updates values automatically.

The FRP model is not new; spreadsheets have been using a similar computation model for decades. Indeed, the massive popularity of spreadsheets shows that the concept of FRP is both intuitive and useful for solving real-world problems.

2.2 A Brief History of FrTime

FrTime (pronounced “father time”) is an implementation of FRP in the language of DrScheme [9]. FrTime differs from earlier FRP systems by upholding the following design goals:

1. It supports incremental program construction in an interactive read-eval-print loop (REPL), so the programmer can interleave program construction, evaluation, and observation.
2. It reuses the Scheme evaluator and provides a conservative extension of Scheme. It permits reuse of legacy (purely functional) Scheme library code, and expressions that do not use its reactive features evaluate exactly as they would in Scheme.

FrTime is implemented in Scheme as a collection of syntactic abstractions (“macros”) [13] and value definitions. The standard Scheme primitives are replaced with versions that support being called with signals as well as plain Scheme values. It thus presents the programmer with the seamless illusion of a Scheme-like language where values that change over time automatically propagate through the program.

Evaluation in FrTime proceeds in two stages: initialization and update. During initialization, the FrTime program runs under a standard Scheme evaluator in a carefully defined environment, which we describe below.

For example, suppose the programmer enters the expression $(+ 3 \text{ seconds})$ at the FrTime REPL. Scheme primitives such as $+$ only operate on ordinary numbers, so evaluating this expression in Scheme would ordinarily yield a type mismatch instead of the desired dataflow semantics.

FrTime is nonetheless able to reuse Scheme’s evaluator because it redefines all such primitive operators with *lifted* versions. Lifting wraps a function with code that checks for signal arguments and, if there are any, returns a new sig-

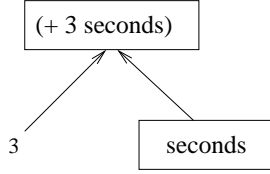


Figure 3. Dataflow graph for $(+ 3 \text{ seconds})$.

nal that depends on those arguments. The lifted primitives thus cause the program’s execution to build a graph of its dataflow dependencies. Nodes correspond to the values of expressions, and arcs indicate the flow of values.

For example, consider FrTime’s implementation of $+$, which could be defined as follows in Scheme:¹

```

(define frtime:+
  (lambda (a b)
    (if (or (signal? a) (signal? b))
        (make-signal (lambda () (+ (project a) (project b)))
                     a b)
        (+ a b))))
  
```

Within the FrTime language, all references to $+$ are automatically redirected to this extended implementation. In $(+ 3 \text{ seconds})$, the second argument is a signal, so the above conditional selects the first branch. This calls *make-signal*, which constructs a new signal from a thunk (nullary procedure) and any number of dependencies; Figure 3 shows the resulting dataflow graph. FrTime calls the thunk to compute the signal’s value, first at creation-time and again each time any of the dependencies changes. The thunk projects the current values of the summands and applies the original Scheme $+$ to them.

This evaluation strategy applies to all expressions, even those free of signals. For example, in $(+ 3 4)$ neither of the arguments is a signal, so the conditional takes the second branch and computes the constant 7 without constructing any dataflow nodes.

FrTime programs can arbitrarily nest and mix computations involving constants and signals. For example, if we write $(+ (+ 1 2) \text{seconds})$, the $(+ 1 2)$ evaluates as in Scheme, reducing to the constant 3, after which evaluation proceeds exactly as above for $(+ 3 \text{seconds})$. Only one signal is created, and the resulting dataflow graph is identical to the one shown in Fig. 3.

After graph construction, the FrTime engine makes the program react to events by propagating changes through the dataflow graph. For example, once every second, a timer triggers a change in *seconds*, which triggers recomputation of every signal that depends on *seconds*, such as $(+ 3 \text{seconds})$ in our example. To ensure that updates always compute over up-to-date values, the algorithm processes the graph in topological order. This of course requires an acyclic graph, so programs with cyclic dependencies must use special combinators (e.g., delays) to break direct dependency cycles.

In addition to supporting syntactic reuse of legacy Scheme code, FrTime’s evaluation model has the benefit of working naturally with an interactive read-eval-print loop (REPL).

¹For simplicity, we treat $+$ as a binary operator. The actual version supports a variable number of arguments.

Because it constructs a new signal object for each primitive operation, it supports dynamic creation of new signals, allowing for incremental program construction.

The signals we’re concerned with in this paper have a well-defined value at every moment during the program’s execution. FRP systems can also model transient phenomena, such as keystrokes and button clicks. In FrTime (as in other FRP systems), the programmer processes these *events* through a set of special operators. Since these do not involve implicit lifting, we do not consider optimizing them in this paper. While the ideas may generalize to events, we have yet to explore such an extension.

2.3 Lifting and Projection

We have already mentioned that FrTime works by replacing Scheme primitives with *lifted* versions. We have also mentioned the *project* operation, which takes a signal and retrieves its current raw Scheme value. Formally, these two operations have the following types:

$$\begin{aligned} \text{lift} & : (t_1 \dots t_n \rightarrow u) \rightarrow (\text{sig}(t_1) \dots \text{sig}(t_n) \rightarrow \text{sig}(u)) \\ \text{project} & : \text{sig}(t) \rightarrow t \end{aligned}$$

In these definitions, t and u are type variables that can stand for any base (non-signal) type, and $\text{sig}(t)$ is either a base type t , or a signal of base type t . That is, t is a subtype of $\text{sig}(t)$. This means that lifted functions are polymorphic with respect to the time-variance of their arguments, so they can consume an arbitrary combination of constants and signals. Likewise, *projecting* the current value of a constant simply yields that constant.

Lift and *project* are related through the following identity:

$$(\text{project } ((\text{lift } f) s \dots)) \equiv (f (\text{project } s) \dots)$$

In words, at any point in time, the current value of the application of a lifted function is equal to the result of applying the original, unlifted function to the projections of the arguments.

Throughout the rest of the paper, we will refer to constants and signals as inhabiting separate *layers*. Specifically, we will talk about constants as belonging to a *lower* layer, and we will underline the names of lower functions, which can only operate on constants. In contrast, we will say that signals belong to an *upper* layer, and we will put a hat over the names of upper functions, which can operate on signals.

Since lifting generalizes the behavior of raw Scheme functions, it is always safe to substitute a lifted function for its lower counterpart. FrTime does exactly this, so programmers rarely need to worry about accidentally applying lower functions to signals. (The exception is when they import raw Scheme libraries, whose procedures must be explicitly lifted.) In the next section, we shall see that this extreme conservatism takes a toll on performance, and we shall explore ways of avoiding it when possible.

3. Static Optimization

Every application of a lifted function to time-varying arguments results in a new dataflow graph node. For example, Fig. 5 (left) shows the dataflow graph for the relatively simple function in Fig. 4. To evaluate this function, six signal objects must be allocated on the heap and connected together: one for each $-$, $+$, *sqr* (square), and *sqrt* (square

```
(define  $\widehat{distance}$ 
  (λ (x1 y1 x2 y2)
    (sqrt (⊕ (sq̄r (⌢ x1 x2))
             (sq̄r (⌢ y1 y2))))))
```

Figure 4. Definition of distance function.

root) in the expression. Each signal object requires nearly one hundred bytes of memory on the heap.

Whenever one of the inputs to the $\widehat{distance}$ function changes, FrTime has to update the four signals along the path from that input to the root. (If multiple inputs change simultaneously, then it must update everything along the union of their paths.) Each update requires 1) extracting the node from a priority queue, 2) retrieving the current value of its input signals, 3) invoking a closure to produce an updated value, 4) storing the new value in the signal object, and 5) iterating through a list of dependent signals and enqueueing them for update. Thus every invocation of the $\widehat{distance}$ function introduces a significant cost in three different areas: the time required to initially *construct* the dataflow graph, the amount of memory required to *store* the dataflow graph, and the time required to *propagate changes* along the dataflow graph.

Figure 6 shows another definition of the $\widehat{distance}$ function, this time with the upper and lower layers made explicit. Note that each of the functions called by $\widehat{distance}$ is actually a lifted version of the lower function by the same name. In other words, they are just lower functions that FrTime has wrapped (like how *frtime:+* wrapped the primitive \oplus function, above). When lifted functions are composed to form expressions, every intermediate value is lifted to the upper layer, only to be immediately projected back to the lower layer by the next function in line.

Our goal is to reduce the use of the expensive dataflow evaluator by eliminating some of the intermediate nodes from the dataflow graph. The key observation is that in many cases it is unnecessary to use the dataflow mechanism for every step of the computation. For example, if a large expression only applies lifted primitives to a collection of signals, then we can replace its graph with a single node that projects the inputs once, performs the whole computation under call-by-value, and lifts the result. We call this transformation *lowering*, since it undoes the effect of intermediate lifting.

By moving computation from the dataflow model back into a call-by-value regime, lowering eliminates the overhead of repeatedly transferring values between the upper and lower layers. It also allows the use of the call stack to transfer control and data, which is much more efficient than using the dataflow graph for the same purpose.

In the $\widehat{distance}$ example above, lowering can collapse the entire graph into a single node, yielding an order of magnitude improvement in both speed and memory usage. Section 4 shows experimental results on substantial programs.

3.1 Dipping and Lowering

We introduce a new syntactic form called **dip**. **Dip** is like *lift* and *project* in that it bridges the two layers, but it does so in a different way.

```
(define  $\widehat{sqrt}$  (lift sqrt))
(define  $\widehat{sq̄r}$  (lift sq̄r))
(define  $\widehat{\oplus}$  (lift ⊕))
(define  $\widehat{\ominus}$  (lift ⊖))
(define  $\widehat{distance}$ 
  (λ (x1 y1 x2 y2)
    (sqrt (⊕ (sq̄r (⌢ x1 x2))
             (sq̄r (⌢ y1 y2))))))
```

Figure 6. Definition of the distance function with upper and lower layers made explicit.

Dip operates on two syntactic entities: a list of variables whose values are assumed to be signals, and an expression which is assumed to be lower code. **Dip** expands into an expression that, at runtime, projects the variables, evaluates the code, and lifts the resulting value. In this way **dip** allows an entire subexpression of lower code to be embedded inside a section of upper code; whereas *lift* operates on functions, **dip** operates on expressions.

$(\mathbf{dip} (x \dots) e) \stackrel{\text{def}}{=} ((\mathit{lift} (\lambda (x \dots) e)) x \dots)$

Each time a **dip** expression is evaluated, it adds a single node to the dataflow graph that depends on all the variables. Note that the list of variables is a co-environment for the **dip**'s body; it contains all the free variables to which the expression actually refers.

In order to optimize a program, we dip as many subexpressions as possible. To dip a subexpression, we extract its set of free variables and replace the code with its lower counterpart. To perform this translation, the optimizer needs to know the lower counterpart of each function it calls.

The lower counterpart of each lifted primitive is simply the original (unlifted) primitive. Initially, primitives are the only functions with known lower counterparts, but as the optimizer processes the program, it may discover user-defined functions that also have lower counterparts. In general, an expression has a lower counterpart if the operation is purely combinatorial and all of its subexpressions have lower counterparts. Our optimizer maintains an explicit mapping between functions and their lower counterparts; we denote entries in this mapping by $\langle \widehat{func}, \underline{func} \rangle$.

Not all functions have lower counterparts. For example, the function \widehat{delay} , which time-shifts a signal's value, needs to remember the history of its changing input signal. It cannot do anything useful if it is called afresh with each new value the signal takes. In general, any function that depends on history has no meaning in the lower layer of constant values. For expressions that involve such functions, it is critical that the optimizer not erroneously dip them, as the resulting program would behave incorrectly.

In the following sections, we will distinguish between lowering, which *replaces* an upper expression with a corresponding lower expression, and dipping, which takes values from the upper layer to the lower layer and back, with some computation in between. The following summarizes the three varieties of code that result from these transformations:

Lower code consists entirely of pure Scheme expressions.

All the functions it calls are lower versions, so it cannot operate on time-varying values.

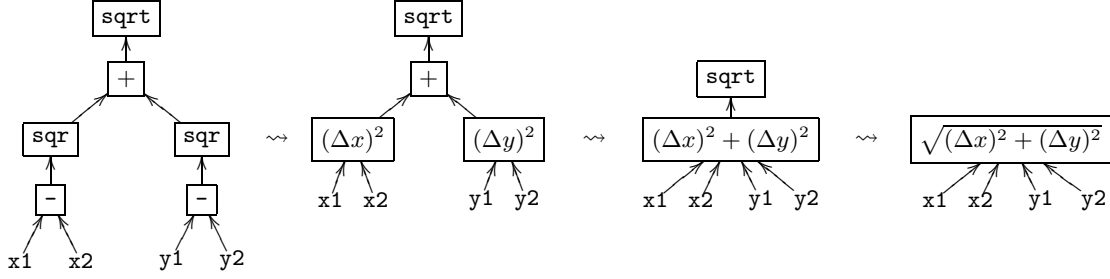


Figure 5. Left: Unoptimized dataflow graph for the distance function. Right: optimized equivalent. Various stages of optimization are shown in-between. Inter-procedural optimization can improve the result even further. Each box is a heap-allocated signal object.

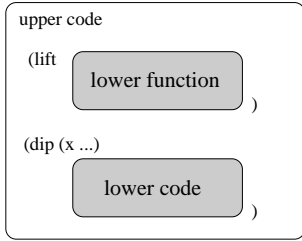


Figure 7. Allowed containment relationships for code.

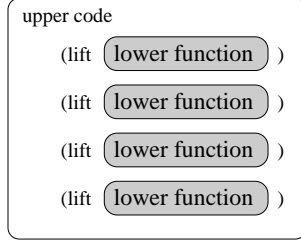


Figure 8. Unoptimized FrTime code.

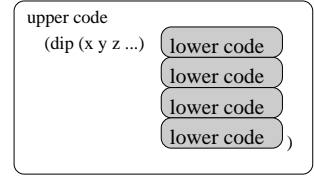


Figure 9. Optimized FrTime code.

Upper code is standard FrTime. Each primitive operation constructs a dataflow node that recomputes its value whenever its input values change.

Dipped code is observationally equivalent to upper code, but operates very differently. Instead of producing *many* dataflow nodes, each of which performs *one* primitive operation, dipped code produces *one* dataflow node that evaluates a complex expression involving *many* primitive operations.

Figure 7 shows the allowed containment relationships for these different varieties of code. At the top-level, the program consists of upper code (we assume that it actually involves signals). This code can refer to lifted functions and dipped expressions, but not to bare lower code. The lifts and dips wrap lower code with logic that protects it from time-varying values. In contrast, lower code never contains upper code of any form (including lifted functions or dipped expressions), since it has no need to process signals.

Figures 8 and 9 illustrate the goal of optimization. Figure 8 represents unoptimized FrTime code. In it, the upper program refers to a large number of small fragments of lifted² code. In comparison, Figure 9 represents code of the sort that we would like the optimizer to produce. The fragments of dipped code have been combined into a small number of larger blocks, reducing the overhead associated with constructing and maintaining a signal for each atomic operation.

² Because the application of a lifted primitive yields the same result as dipping, we could just express everything in terms of **dip**. However, lifting is an established term within the FRP community, so we use it for clarity.

3.2 The Lowering Algorithm

The optimization algorithm works by dipping subexpressions in a bottom-up fashion. It begins with variables and constants, then proceeds to their parent expressions, their grandparent expressions, and so on.

Formally, the algorithm is guided by a set of rewrite rules. We write $\Gamma \vdash e \rightsquigarrow (\mathbf{dip}(\vec{x}) e')$ to indicate that e' is the dipped version of e , where the environment Γ associates function names with the names of their lower counterparts, and \vec{x} is the set of all signals on which the value of e may depend. For example, dipping of literals c simply involves wrapping them in a **dip** expression. Since the value of a literal is always a constant, its dipped equivalent does not depend on anything:

$$\vdash c \rightsquigarrow (\mathbf{dip} () c)$$

We treat identifiers similarly, but since they may refer to signals, we include them in the list of dependencies:

$$\vdash id \rightsquigarrow (\mathbf{dip} (id) id)$$

For example, in the case of the $\widehat{distance}$ function, the optimizer arrives at the identifiers $x1$ and $x2$ and applies this rule, resulting in the following expression:

$$\begin{aligned}
 &(\mathbf{define} \widehat{distance} \\
 & \quad (\lambda (x1\ y1\ x2\ y2) \\
 & \quad \quad (\widehat{sqr} (\widehat{+} (\widehat{sqr} (\widehat{-} (\mathbf{dip} (x1) x1) \\
 & \quad \quad \quad (\mathbf{dip} (x2) x2)))))) \\
 & \quad \quad (\widehat{sqr} (\widehat{-} y1\ y2))))))
 \end{aligned}$$

The optimizer proceeds by combining dipped subexpressions into larger code fragments. In the case of function applications, it computes the union of the arguments' dependencies and replaces the lifted function with its lower counterpart:

$$\frac{\widehat{f, \underline{f}} \in \Gamma \quad \Gamma \vdash e_i \rightsquigarrow (\mathbf{dip} (\vec{x}_i) e'_i)}{\Gamma \vdash (\widehat{f} e_i \dots) \rightsquigarrow (\mathbf{dip} (\vec{x}_i \dots) (f e'_i \dots))}$$

Continuing the $\widehat{distance}$ example, one application of this rule produces the following result:

$$\begin{aligned} &(\mathbf{define} \widehat{distance} \\ &(\lambda (x1 \ y1 \ x2 \ y2) \\ &(\widehat{sqr} (\widehat{+} (\widehat{sqr} (\mathbf{dip} (x1 \ x2) (\underline{=} x1 \ x2))) \\ &(\widehat{sqr} (\widehat{-} y1 \ y2)))))) \end{aligned}$$

Applying this rule once more produces:

$$\begin{aligned} &(\mathbf{define} \widehat{distance} \\ &(\lambda (x1 \ y1 \ x2 \ y2) \\ &(\widehat{sqr} (\widehat{+} (\mathbf{dip} (x1 \ x2) (\underline{sqr} (\underline{=} x1 \ x2))) \\ &(\widehat{sqr} (\widehat{-} y1 \ y2)))))) \end{aligned}$$

Next, the optimizer dips the second argument to $\widehat{+}$, which is transformed identically to the left branch:

$$\begin{aligned} &(\mathbf{define} \widehat{distance} \\ &(\lambda (x1 \ y1 \ x2 \ y2) \\ &(\widehat{sqr} (\widehat{+} (\mathbf{dip} (x1 \ x2) (\underline{sqr} (\underline{=} x1 \ x2))) \\ &(\mathbf{dip} (y1 \ y2) (\underline{sqr} (\underline{=} y1 \ y2)))))) \end{aligned}$$

Since dipping does not change the observable semantics of an expression, it is safe to stop optimizing at any time. In this case the bottom-up traversal will continue until it reaches the λ , at which point it must stop because of subtleties involved with lambda abstractions (explained below).

The final optimized result contains only a single **dip** expression, which means that when evaluated, it creates only a single dataflow graph node instead of the six nodes required for the original function. We show the final dataflow graph, along with some intermediate graphs, in Fig. 5. The final code is as follows:

$$\begin{aligned} &(\mathbf{define} \widehat{distance} \\ &(\lambda (x1 \ y1 \ x2 \ y2) \\ &(\mathbf{dip} (x1 \ x2 \ y1 \ y2) \\ &(\underline{sqr} (\underline{+} (\underline{sqr} (\underline{=} x1 \ x2))) \\ &(\underline{sqr} (\underline{-} y1 \ y2)))))) \end{aligned}$$

Though the above example does not contain any **let** expressions, dipping them is also straightforward. The newly-introduced binding (*id*) is excluded from the body's dependency list (\vec{x}_e) because it is guaranteed to be subsumed by the bound value's dependency list (\vec{x}_v).

$$\frac{\Gamma \vdash v \rightsquigarrow (\mathbf{dip} (\vec{x}_v) v') \quad \Gamma \vdash e \rightsquigarrow (\mathbf{dip} (\vec{x}_e) e')}{\Gamma \vdash (\mathbf{let} ((id \ v)) e) \rightsquigarrow (\mathbf{dip} (\vec{x}_v \cup (\vec{x}_e \setminus id)) (\mathbf{let} ((id \ v')) e'))}$$

The following subsections describe the details of optimizing several different constructs.

3.3 Lambda Abstractions

Dipping a λ expression is somewhat subtle. For example, suppose the optimizer encounters the following expression:

$$(\lambda (x) (\widehat{+} x \ 3))$$

So far, we have dipped expressions by wrapping their lowered counterparts in the **dip** form. If we do that, we get:

$$(\mathbf{dip} () (\lambda (x) (\underline{+} x \ 3)))$$

This is clearly unsafe, because if the resulting closure were applied to a signal, the lowered $+$ operator would cause a type error. To prevent such errors, we dip only the *body* instead of the whole λ expression. In this case, the result is:

$$(\lambda (x) (\mathbf{dip} (x) (\underline{+} x \ 3)))$$

In general, the rule is as follows:

$$\frac{\Gamma \vdash e \rightsquigarrow (\mathbf{dip} (\vec{x}) e')}{\Gamma \vdash (\lambda (\vec{v}) e) \rightsquigarrow (\lambda (\vec{v}) (\mathbf{dip} (\vec{x}) e'))}$$

If the optimizer never lowered function bodies, then user-defined functions could never be assigned lower counterparts. This would make the analysis purely intraprocedural, greatly reducing the opportunities for optimization.

To allow interprocedural optimization, we take advantage of the fact that a **dip** expression's body is the original expression's lower counterpart. Therefore, if the optimizer successfully dips a function, then it knows that function's lower counterpart. We write $\Gamma \vdash e \rightsquigarrow (\mathbf{dip} (\vec{x}) e')$ to indicate not only that e' is the dipped version of e , but that in addition e is a lambda expression whose body can be lowered:

$$\frac{\Gamma \vdash e \rightsquigarrow (\mathbf{dip} (\vec{x}) e')}{\Gamma \vdash (\lambda (\vec{v}) e) \rightsquigarrow (\mathbf{dip} (\vec{x} \setminus \vec{v}) (\lambda (\vec{v}) e'))}$$

References to variables bound by the lambda's argument list are removed from the list of dependencies, since in a lower context they cannot be signals.

When the \rightsquigarrow transformation applies, the optimizer adds a top-level definition for the lower counterpart of \widehat{f} , called \underline{f} , and remembers the association $\widehat{f, \underline{f}}$:

$$\frac{\Gamma \cup \widehat{f, \underline{f}} \vdash e \rightsquigarrow (\mathbf{dip} (\vec{x}) e') \quad \Gamma \cup \widehat{f, \underline{f}} \vdash e \rightsquigarrow (\mathbf{dip} () e'')}{\Gamma \vdash (\mathbf{define} \widehat{f} e) \rightsquigarrow (\mathbf{begin} (\mathbf{define} \widehat{f} (\mathbf{dip} (\vec{x}) e')) (\mathbf{define} \underline{f} e''))}$$

The above rule expands the scope of the optimization to include interprocedural optimization. On the other hand, if a definition does *not* have a lower counterpart then only the dipped version is defined:

$$\frac{\Gamma \vdash e \rightsquigarrow (\mathbf{dip} (\vec{x}) e') \quad \Gamma \cup \widehat{f, \underline{f}} \vdash e \rightsquigarrow (\mathbf{dip} () e'')}{\Gamma \vdash (\mathbf{define} \widehat{f} e) \rightsquigarrow (\mathbf{define} \widehat{f} (\mathbf{dip} (\vec{x}) e'))}$$

If a program contains a sequence of definitions, each definition is dipped separately:

$$\frac{\Gamma \vdash e_i \rightsquigarrow (\mathbf{dip} (\vec{x}_i) e'_i)}{\Gamma \vdash (\mathbf{begin} e_i \dots) \rightsquigarrow (\mathbf{dip} (\vec{x}_i \dots) (\mathbf{begin} e'_i \dots))}$$

For concision and clarity, the above judgements do not describe the full mechanism for interprocedural optimization. Adding this would be straightforward but would increase the size of the judgements considerably.

3.4 Conditionals

The criterion for dipping **if** expressions is the same as for all other expression types: all of their subexpressions must have lower counterparts. Moreover, the consequence is also the same, namely that the resulting node depends on the union of the subexpressions' dependencies.

$$\frac{\begin{array}{l} \Gamma \vdash c \rightsquigarrow (\mathbf{dip} (\vec{x}_c) c') \\ \Gamma \vdash t \rightsquigarrow (\mathbf{dip} (\vec{x}_t) t') \\ \Gamma \vdash f \rightsquigarrow (\mathbf{dip} (\vec{x}_f) f') \end{array}}{\Gamma \vdash (\mathbf{if} c t f) \rightsquigarrow (\mathbf{dip} (\vec{x}_c \cup \vec{x}_t \cup \vec{x}_f) (\mathbf{if} c' t' f'))}$$

Conditional evaluation in FrTime is relatively expensive, so dipping conditionals can improve performance significantly. Moreover, dipping of conditionals is necessary in order to define lower counterparts for recursive functions, which makes it possible to collapse a potentially long chain of graph fragments into a single node.

3.5 Higher Order Functions

Higher order function applications, which evaluate a closure passed as an argument, cannot be dipped using only the strategy defined in this paper. For example, consider the type of \widehat{map} :

$$\widehat{map} : \text{sig}(\text{sig}(t) \rightarrow \text{sig}(u)) \times \text{sig}(\text{list}(t)) \rightarrow \text{sig}(\text{list}(u))$$

\widehat{map} 's first argument is a signal, which can be called to produce another signal. That is, the choice of which function to apply can change over time, as can the result of applying the function. Dipping only removes the first kind of time dependency, not the second. If $\langle \widehat{map}, \underline{map} \rangle$ were a valid upper/lower pair, then the type of \underline{map} would have to be:

$$\underline{map} : (\text{sig}(t) \rightarrow \text{sig}(u)) \times \text{list}(t) \rightarrow \text{list}(u)$$

Clearly this could cause a problem at runtime, since the actual \underline{map} doesn't support functions that may produce signals. In order to avoid this problem, we ensure that the optimizer never associates a lower counterpart with a higher order function. For the built-in higher order functions such as \widehat{map} and \widehat{apply} , we just omit them from the optimizer's initial mapping. However, this still leaves the question of higher order functions written by users.

The only way a user-defined function can be assigned a lower counterpart is if its body can be completely lowered; however, no higher order function can satisfy this requirement, since at some point it must call the closure passed as an argument. Lexical scoping guarantees that the function's arguments will have fresh names, so the optimizer cannot possibly know of a lower counterpart for the argument closure. Since the function makes a call with no known lower counterpart, the body is not lowerable.

We could address this weakness by using a static dataflow analysis to identify closures that have known lower counterparts. However, we have not yet found a need for such an extension, and, in any case, it is always safe to assume the absence of lower counterparts. It just means we cannot optimize certain expressions.

3.6 Data Constructors

If the upper version of a function that allocates memory (such as \widehat{cons} or \widehat{append}) is defined by lifting the lower version, then the resulting upper function will be forced to reallocate a new segment of memory every time any of its input signals change. For large or recursive data structures this can be a tremendous waste of time, as well as an unnecessary burden on the garbage collector. For example, imagine a list of 10,000 integers being recreated from scratch whenever one of those integers changed.

In order to avoid this inefficiency, FrTime defines custom versions of these functions. These work by allocating a single segment of memory at node construction time. They later update the memory in place whenever any of the input signals change. This reduces pressure on the garbage collector, and in applications where signals are changing rapidly it can lead to a significant speedup. This in-place updating is safe because the dataflow engine is (assumed to be) the only entity that uses mutation (the FrTime application should in general be purely declarative). A complication does arise from operators such as **delay**, which must now explicitly copy any value that might be overwritten before it is needed again.

We must be careful when optimizing expressions that call these hand-crafted functions. If the expression is blindly dipped, then all the performance consequences of frequently reallocating memory will immediately reappear. In order to prevent that from happening, we do not allow the optimizer to lower allocating functions. Thus, expressions that construct data structures are not optimizable. However, accessor functions such as *car* and *cdr* are still lowerable.

3.7 Inter-Module Optimization

DrScheme's module framework makes it easy to write the optimizer in such a way that it processes each module individually. An unfortunate consequence of this approach is that the associations between user-defined functions and their lower counterparts is not shared between modules. Unless the optimizer can recover these associations, it will lose many opportunities for optimization. It will be unable to optimize any expression containing a call to a function whose entry was forgotten, even if that call is in a deeply nested subexpression. For commonly used functions such as those that manipulate low-level data types, this can have a significant domino effect.

In order to recover the lost associations, the FrTime optimizer uses a consistent naming convention to identify the lower counterpart of an upper function (Scheme doesn't understand the underlines and overlines we have been using in this paper, so we must perform some name mangling anyway). Because of this naming convention, the optimizer can recover the forgotten associations simply by inspecting a module's list of exported identifiers. This allows the optimizer to perform inter-module optimization.

The flexibility of this mechanism provides an additional usability benefit: it allows the programmer to define a hand-

coded lower counterpart for any function that the optimizer was unable to lower automatically.

3.8 Macros

Since macros must be fully expanded before runtime, they can have no time-varying semantics. They are therefore easy to support; the optimizer simply expands all macros before attempting to apply the lowering optimization.

3.9 Pathological Cases

In most cases, lowering reduces execution time and memory requirements, but there are instances in which it can have the opposite effect. The reason is that lowering combines several small fragments of code, each depending on a few signals, into a large block that depends on many signals. For example, consider the following simple expression:

(expensive-operation (quotient milliseconds 10000))

Though *milliseconds* changes frequently, the *quotient* changes relatively rarely. If run under the standard FrTime evaluator, the *quotient* node will stop propagation when its result doesn't change, thus short-circuiting the recomputation of the *expensive-operation* most of the time. However, in the "optimized" version, this whole computation (and perhaps more) is combined into a single node, which must recompute *in its entirety* each time *milliseconds* changes.

As discussed in Section 3.3, interprocedural optimization requires that the optimizer produce two versions of each lowerable procedure definition: one that is merely dipped, and one that is actually lowered. Lowering thus has the potential to double the size of a program's code. We have so far chosen not to worry about this because the optimized code is static and, in most cases, accounts for a relatively small fraction of a program's overall dynamic memory usage. However, for large programs, this may become a concern. In particular, recent versions of DrScheme employ a just-in-time compiler, which generates native code for each executed procedure body. Since native code occupies considerably more space than expression data structures, lowering has the potential to increase a program's memory usage significantly.

4. Evaluation

In this section we present the impact of optimization on several FrTime benchmarks. We also discuss the impact on the usability of FrTime.

4.1 Performance

We employ four different benchmarks to evaluate the effect of our optimization on the resource requirements of various programs. Other than the Count microbenchmark, none of these applications were written with lowering in mind, so we expect the findings to be broadly representative.

Table 1 summarizes the performance results.³ Size denotes the program's size measured by the number of expressions ("parentheses"). The *orig* and *opt* subscripts denote the original and optimized versions. Start is the initial graph construction time, while Run is reaction time, i.e., the time for

³Measured on a Dell Latitude D610 with 2Ghz Pentium M processor and 1GB RAM, running Windows XP Pro SP2 with SpeedStep disabled. The numbers are the mean over three runs from within DrScheme version 360, restarting DrScheme each time.

	Count	Needles	S'sheet	TexPict
Size (exprs)	7	62	2,663	13,022
Start _{orig} (sec)	9.5	89.0	9.2	35.2
Start _{opt} (sec)	<0.1	35.3	11.8	28.9
Mem _{orig} (MB)	204.7	581.4	34.8	170.7
Mem _{opt} (MB)	0.2	240.5	50.9	119.4
Shrinkage (ratio)	971	2.4	0.7	1.4
Run _{orig} (sec)	4.8	5.6	19.3	273.4
Run _{opt} (sec)	<0.1	2.0	20.5	3.5
Speedup (ratio)	16,000	2.8	0.94	78.1

Table 1. Experimental benchmark results.

a change to percolate through the graph. Times <0.1 are too small to be measurable. Mem denotes memory footprint beyond that of DrScheme (which is 72MB). Speedup denotes the ratio between the unoptimized run-time and the optimized run-time, and Shrinkage denotes the analogous ratio for memory usage.

The Count microbenchmark consists of a function that takes a number, recursively decrements it until reaching zero, and then increments back up to the original number, i.e., an extremely inefficient implementation of the identity function for natural numbers. The purpose of the benchmark is to quantify the potential impact of lowering for code that involves a large number of very simple operations (in this case only addition, subtraction, comparison, and conditionals). The results are unsurprisingly dramatic: for inputs around 600, the original version takes several seconds to start and then takes nearly five seconds to recompute whenever the input value changes. In contrast, even for inputs in the hundreds of thousands, the optimized version starts immediately and uses less than a tenth of a second to recompute.

The Needles program (due to Robb Cutler) displays a 60×60 grid of unit-length vectors. Each vector rotates to point towards the mouse cursor, and its color depends on its distance from the mouse cursor. The main effect of optimization is to collapse the portions of the dataflow graph that calculate each vector's color and angle. Since these constitute the majority of the code, optimization has a significant effect. The optimized version runs nearly three times faster and uses just about half as much memory.

The Spreadsheet program implements a standard 2D spreadsheet. Formulas are evaluated by calling Scheme's built-in *eval* procedure in the FrTime namespace. The startup phase has several calculations for drawing the grid, setting the size of scroll-bars, etc., which are optimized. We were somewhat surprised to see that the "optimized" spreadsheet requires more time and space than the original version. One reasonable explanation is that, because the spreadsheet was designed from the beginning to run in FrTime, its dataflow graphs already work efficiently under the default FrTime evaluator. Also, as explained in Section 3.9, there are certainly cases where lowering can make programs less efficient. In most cases this inefficiency is more than outweighed by the corresponding reduction in dataflow evaluation, but apparently not in this case.

TexPict is the image-compositing subsystem of Slideshow, which initiated this project by its abysmal performance as a transparently reactive FrTime program. As can be seen from the results of our experiments, lowering yields a speedup of almost two orders of magnitude. The result is still significantly slower than a pure Scheme analog, but

fast enough to make it usable for many applications. This offers strong evidence in support of our hypothesis that large dataflow graphs arising from implicit, fine-grained lifting can lead to a significant slowdown. Moreover, it demonstrates that with lowering, transparent reactivity becomes feasible for real legacy programs.

The `TexPict` benchmark is also interesting because it frequently uses higher order functions. Since our first order analysis yields a dramatic improvement even in this case, we posit that our current approach is sufficient for a broad range of applications, even those that use higher order functions extensively.

4.2 Usability

In `DrScheme`, any collection of syntax and value definitions can be bundled into a module that comprises a “language”. For example, the `FrTime` language is a set of lifted primitives, along with special definitions for certain syntactic forms (e.g., conditionals). The optimized language is defined similarly, except that it defines a syntax transformer for a whole `FrTime` program. `FrTime` programmers enable optimization simply by changing the module language declaration from `frtime` to `frtime-opt`. The optimizer will be shipped with the next standard `DrScheme` release, so no additional installation or configuration is necessary.

Even though the `FrTime` optimizer works by performing a source-to-source transformation, it does not adversely affect the programmer’s ability to understand the original program. In particular, the optimizer preserves source location information within the transformed code, so the runtime system reports errors in terms of the original source code [7, 11]. Furthermore, if optimization fails for some section of code (perhaps due to the use of an unsupported feature, or even due to a bug in the optimizer itself), the optimizer will silently fall back to using the original code, and continue the optimization at the next top-level definition.

Users can discover whether or not a particular piece of code was optimized by examining the fully expanded result; un-optimized code will be preceded by a literal string explaining what went wrong during optimization. On the other hand, code that *was* optimized will stand out because the names of upper functions will have been replaced with the mangled names of their lower counterparts.

The overhead of the optimization pass is quadratic in the nesting depth of function definitions and linear in the size of the code base. This makes it practical to apply optimization to large systems, such as the `TexPict` benchmark presented above. Furthermore, an optimized module can be precompiled so that the overhead of static analysis does not need to be repeated when the module is used later.

5. Related Work

Deforestation [18] and listlessness [17] are optimization techniques that eliminate intermediate data structures from functional programs. Their purpose is analogous to that of lowering, which eliminates intermediate nodes from a dataflow graph. Although the mechanics of these transformations are quite different from those of lowering, for stream-based FRP implementations [8, 12], we imagine that deforestation and listlessness could have an effect similar to lowering: namely, the weaving of multiple stream iterators into a single processing loop. `FrTime`, however, seems

to require more special techniques because of its imperative implementation.

Most other FRP implementations [6, 15, 20, 21] do not attempt to provide transparent reactivity through implicit lifting. A notable exception is `Fran` [8], which provides a substantial set of lifted primitives and might therefore benefit from the sort of optimization we’ve presented. The other systems leave the decision of when to lift up to the programmer. In Haskell, the static type system ensures that an unlifted function cannot be applied to signals, but the programmer chooses the granularity of lifting.

`Yampa` [15] implements a dynamic optimization that achieves essentially the same effect as lowering. When it evaluates a composition of pure signal functions, it replaces them with a single signal function that computes the composition of the original functions. In `FrTime`, such a dynamic optimization would be difficult to implement without loss of sharing. Specifically, without examining the program’s syntactic structure, we cannot determine which intermediate signals can escape their context of creation, in which case they must exist as separate nodes.

`Nilsson` [14] explores the use of generalized abstract data types (GADTs) to support optimizations in `Yampa` [15]. The idea is to use GADTs to define special cases of signal processors, such as constants and the identity function, and implement special, optimized logic for them in the evaluator. In particular, `Nilsson`’s implementation performs constant-propagation and automatically eliminates calls to the identity function, yielding measurable improvement in various benchmarks. Moreover, the GADT-based optimizations can be applied to networks of stateful signal processors, which our approach cannot handle.

Real-time FRP (RT-FRP) [21] is an implementation of FRP that shares certain similarities with `FrTime`, such as the explicit connection to an underlying host language with a collection of base primitives. The goal of RT-FRP is not to produce highly efficient code so much as to establish provable bounds on the time and space required by each round of execution. The language achieves these bounds through a conservative static analysis, but it does not perform any optimizing program transformations.

Event-driven FRP (E-FRP) [22] is a modification of RT-FRP designed to support compilation to efficient imperative code. E-FRP adds some crucial restrictions to RT-FRP that make such compilation possible. Primarily, it takes away the ability to perform dynamic switching, thereby making the program’s data dependencies static. It also requires that only one external event can stimulate the system in any given update cycle. As in RT-FRP, the language performs no optimizing program transformations; rather, it uses a syntactic restriction to guarantee limits on the program’s resource requirements. In forbidding dynamic switching, E-FRP more closely resembles traditional synchronous dataflow languages, such as `Lustre` [4], `Esterel` [3], and `Signal` [2]. These languages have a common goal of compiling to efficient straightline code, which they achieve by design. This is in contrast to `FrTime`, whose primary goal is to provide expressive power, often at the expense of performance.

6. Conclusions and Future Work

We have presented a novel optimization technique for functional reactive languages, specifically those that use implicit

lifting to achieve transparent reactivity. Our technique works by processing the source program in a bottom-up fashion, recursively combining calls of lifted primitives into larger *dipped* expressions. This has the effect of shifting significant computation from the dataflow mechanism back to the underlying (in this case) call-by-value evaluator. Though our analysis is still unable to handle certain language features, such as higher order functions, experimental results indicate that the technique can achieve a significant reduction in a program's time and space needs, making transparent reactivity a viable approach for realistic systems.

The notion of lowering applies outside of FRP, for example to any monad [19] where the *lift* operator distributes over function composition. Specifically, wherever $(\text{lift } g) \circ (\text{lift } f) \equiv \text{lift } (g \circ f)$, and *lift* is expensive, it is beneficial to rewrite to reduce the number of *lifts*. Lowering may therefore be useful in general for languages that use monads extensively. For example, the Glasgow Haskell Compiler [16] optimizes code by rewriting expressions according to such identities.

One limitation of the technique described here is that if a subexpression has no lower counterpart, then the expression containing that subexpression cannot be lowered either. This limitation could be avoided by hoisting the problematic subexpression out and storing its result in a temporary variable; however, in a call-by-value language like Scheme, such a transform must take care not to affect evaluation order. Translating to continuation-passing style would make evaluation order easier to deal with, but would make it more difficult to identify *dippable* subexpressions.

For languages that support runtime code generation, it would be possible to explicitly build the dataflow graph first, and then collapse nodes into call-by-value subexpressions. This approach would trivially support inter-procedural optimization, and would be able to collapse arbitrary nodes in the dataflow graph, whether or not they contained unlowerable subexpressions in the original program text. This approach would depend on the ability of the runtime environment to compile dynamically-generated subexpressions into efficient code.

Finally, we plan to apply the lowering optimization to Flapjax [1], a new functional reactive language designed for writing modern Web applications. Since the language provides transparent reactivity and employs a FrTime-like evaluation model, we expect to achieve similar results.

Acknowledgements

We thank Phil Wadler for providing valuable insight on related work in Haskell, and we are especially grateful to Henrik Nilsson for his detailed feedback on the content of the paper. We also thank the anonymous reviewers for advice on the presentation.

References

- [1] The Flapjax programming language. <http://flapjax-lang.org/>.
- [2] A. Benveniste, P. L. Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16(2):103–149, 1991.
- [3] G. Berry. *The Foundations of Esterel*. MIT Press, 1998.
- [4] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A declarative language for programming synchronous

- systems. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 178–188, 1987.
- [5] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *European Symposium on Programming*, pages 294–308, 2006.
- [6] A. Courtney. Frappé: Functional reactive programming in Java. In *Practical Aspects of Declarative Languages*, pages 29–44. Springer-Verlag, March 2001.
- [7] R. K. Dybvig, R. Hieb, and C. Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, Dec. 1993.
- [8] C. Elliott and P. Hudak. Functional reactive animation. In *ACM SIGPLAN International Conference on Functional Programming*, pages 263–277, 1997.
- [9] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, 2002.
- [10] R. B. Findler and M. Flatt. Slideshow: Functional presentations. In *ACM SIGPLAN International Conference on Functional Programming*, pages 224–235, 2004.
- [11] M. Flatt. Composable and compilable macros. In *ACM SIGPLAN International Conference on Functional Programming*, pages 72–83, 2002.
- [12] P. Hudak. *The Haskell school of expression: learning functional programming through multimedia*. Cambridge, 2000.
- [13] E. E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. F. Duba. Hygienic macro expansion. In *ACM Symposium on Lisp and Functional Programming*, pages 151–161, 1986.
- [14] H. Nilsson. Dynamic optimization for functional reactive programming using generalized abstract data types. In *ACM SIGPLAN International Conference on Functional Programming*, pages 54–65, 2005.
- [15] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *ACM SIGPLAN Workshop on Haskell*, pages 51–64, 2002.
- [16] S. L. Peyton Jones. Compiling Haskell by transformation: a report from the trenches. In *European Symposium on Programming*, pages 18–44, 1996.
- [17] P. Wadler. Listlessness is better than laziness. In *ACM Symposium on Lisp and Functional Programming*, pages 45–52, 1986.
- [18] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.
- [19] P. Wadler. The essence of functional programming. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, January 1992.
- [20] Z. Wan and P. Hudak. Functional reactive programming from first principles. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–252, 2000.
- [21] Z. Wan, W. Taha, and P. Hudak. Real-time FRP. In *ACM SIGPLAN International Conference on Functional Programming*, pages 146–156, 2001.
- [22] Z. Wan, W. Taha, and P. Hudak. Event-driven FRP. In *Practical Aspects of Declarative Languages*, pages 155–172, 2002.