Verifying Event-Driven Programs using Ramified Frame Properties

Neelakantan R. Krishnaswami
Microsoft Research
neelk@microsoft.com

Lars Birkedal
IT University of Copenhagen
birkedal@itu.dk

Jonathan Aldrich
Carnegie Mellon University
jonathan.aldrich@cs.cmu.edu

Abstract

Interactive programs, such as GUIs or spreadsheets, often maintain dependency information over dynamically-created networks of objects. That is, each imperative object tracks not only the objects its own invariant depends on, but also all of the objects which depend upon it, in order to notify them when it changes.

These bidirectional linkages pose a serious challenge to verification, because their correctness relies upon a global invariant over the object graph.

We show how to *modularly* verify programs written using dynamically-generated bidirectional dependency information. The critical idea is to distinguish between the footprint of a command, and the state whose invariants depends upon the footprint. To do so, we define an application-specific semantics of updates, and introduce the concept of a *ramification operator* to explain how local changes can alter our knowledge of the rest of the heap. We illustrate the applicability of this style of proof with a case study from functional reactive programming, and formally justify reasoning about an extremely imperative implementation as if it were pure.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms languages, verification

Keywords separation logic, frame rule, ramification problem, dataflow, functional reactive programming, subject-observer

1. Introduction

In many interactive programs, there are mutable data structures which change over time, and which must maintain some relationships with one another. For example, in a spreadsheet, each cell contains a formula, which may refer to other cells, and whenever the user changes a cell, all of the cells which transitively depend upon it must be updated. Since spreadsheets can get very large, this should ideally be done in a lazy way, so that only the cells visible on the screen (and the cells necessary to compute them) are themselves recomputed.

Typically, these dependencies are written using the *subject-observer* pattern. A mutable data structure (the subject) maintains a list of all of the data structures whose invariants depend upon it (the observers). Whenever it changes, it calls a function on each observer to update it in response to the change. (And in turn, the observers of the subject may be subjects of still other observers.)

While natural, these programs are very challenging to verify in a modular way, even when using a resource-sensitive logic adapted to reasoning about aliased mutable data, such as separation logic. The reason is that there are two directions of dependency, both of which matter for program proof. First, our program invariant must have ownership over the subject's data (its *footprint*) in order to prove the correctness of code modifying the subject. This direction of ownership is natural to verify with separation logic.

However, we must explicitly maintain the *other* direction of dependency as well — we track everything which depends upon the subject, and modify the dependent data appropriately whenever the subject changes. Hence, the natural program invariant now becomes a global property: we need to know the full dependency graph covering all subjects and observers to express that the reads and is-read-by relations are relational transposes of one another. The global nature of this invariant means that a naive correctness proof will not respect the modular structure of the program — if we modify the dependency graph in any way, we now have to re-verify the entire program!

But, the intention of the subject-observer pattern is precisely to allow the program to remain oblivious to the exact number and nature of the observers, so that the programmer may add new observers without disturbing the behavior of the rest of the program. Our goal, then, is to find a way of taking this piece of practical software engineering wisdom, and casting it into formal terms amenable to proof.

Concretely, our contributions are as follows:

- We define a library with a monadic API for writing demand-driven computations with dynamic dependencies and local state. This library is implemented using higher-order functions dynamically creating networks of imperative callbacks.

  We then give an "abstract semantics" for this library, structured as a set of separation logic lemmas about our dataflow library. These lemmas permit *modular* correctness proofs about programs using this API, even in the face of the fact that the program invariants must be defined globally upon the whole callback network.

  The key idea is to distinguish between the direct footprint of a command, and the program state which depends upon that footprint. The lemmas are then phrased so that they refer only to the direct footprint of each command in the API. In addition, we structure our lemmas to justify an unusual frame property for our abstract semantics, which we can use to verify different parts of an imperative dataflow network separately.

TLDI '10 January 23, 2010, Madrid, Spain.

Unlike typical frame properties, the frame in our frame rule is not the same in the pre- and the post-states. Instead, the two sides of the frame are related by a *ramification operator* (so named in analogy to the "ramification problem" in AI), which explains how local changes can alter our knowledge of the rest of the heap.

- To illustrate the utility of this proof technique, we verify an imperative implementation of combinators implementing stream transducers in the style of functional reactive programming.

  Ultimately, clients can reason about the behavior of the imperative implementation as if it were purely functional, even though it is implemented using local state and imperative callback procedures.

For space reasons, many of the detailed proofs have been omitted from this extended abstract, but we emphasize that full proofs have been carried out; in particular, each triple in the library specification has been proven using the specification logic in Section 2.

2. Programming Language and Logic

The formal system we present has three layers. First, we have a core programming language we call Idealized ML. It is a predicatively-polymorphic functional language which isolates all side effects inside a monadic type (Pfenning and Davies 2001). Our notion of side effects includes nontermination in addition to the allocation, access, and modification of general references (including pointers to closures). Then, we give an assertion language based on higher-order separation logic (Biering et al. 2007) to describe the state of a heap. Separation logic allows us to give a clean treatment of issues related to specifying and controlling aliasing, and higher-order predicates allow us to abstract over the heap, hiding the exact layout of a module's heap data structures and thereby enforcing encapsulation. Finally, we have a specification logic to describe the effects of programs, which is a first-order logic whose atomic propositions are Hoare triples $\{p\}c\{a : A.\ q\}$, which assert that if the heap is in a state described by the assertion p, then executing the command c will result in a postcondition state q (with the variable a referring to the return value of the command).

Programming Language. The core programming language we have formalized is an extension of the polymorphic lambda calculus with a monadic type constructor to represent side-effecting computations. The types of our language are the unit type 1, the function space $A \to B$, inductive types like the natural number type $\mathbb{N}$, the reference type $\mathsf{ref}\ A$, as well as universal and existential types $\forall\alpha : \kappa.\ A$ and $\exists\alpha : \kappa.\ A$.[1]

In addition, we have the monadic type $\bigcirc A$, which is the type of suspended side-effecting computations producing values of type A. Side effects include both heap effects (such as reading, writing, or allocating a reference) and nontermination.

We maintain such a strong distinction between pure and impure code for two reasons. First, it allows us to use strong equational reasoning principles for our language: we can validate the full β and η rules of the lambda calculus for terms of non-monadic types, such as functions, sums, and products. These rules simplify reasoning even about imperative programs, because we can relatively freely restructure the program source to follow the logical structure of a proof. Second, when program expressions appear in assertions — that is, the pre- and post-conditions of Hoare triples — they must be pure. However, allowing a rich set of program expressions like function calls or arithmetic in assertions makes it much easier to write specifications. So we restrict which types can contain side-effects, and thereby satisfy both requirements.

The pure terms of the language are typed with a typing judgment $\Theta;\ \Gamma \vdash e : A$, which can be read as "In the type context Θ and the variable context Γ, the pure expression e has type A." Computations are typed with the judgment $\Theta;\ \Gamma \vdash c \div A$, which can be read as "In the type context Θ and the variable context Γ, the computation c is well-typed at type A ." The rules for both of these judgments are standard and omitted.

We have $\langle\rangle$ as the inhabitant of 1, natural numbers z and $\mathsf{s}(e)$, and functions $\lambda x : A.\ e$. We also have the corresponding eliminations for each type, including projections for products and case statements for sum types. For the natural numbers, we add a primitive iteration construct $\mathsf{iter}(e, e_z, x.\ e_s)$. If $e = \mathsf{z}$, this computes e_z , and if $e = \mathsf{s}(e')$, it computes $e_s[(\mathsf{iter}(e', e_z, x.\ e_s))/x]$. Bounded iteration allows us to implement (for example) arithmetic operations as pure expressions. We will also freely make use of other polynomial data types (such as sum types, lists, option types, and trees) as needed.

Suspended computations $[c]$ inhabit the monadic type $\bigcirc A$. These computations are not immediately evaluated, which allows us to embed them into the pure part of the programming language. Furthermore, we can take fixed points $\mathsf{fix}\ x : D.\ e$ of terms, to give us a general recursion. Because we wish to permit nonterminating programs only at monadic types, we must restrict fix to a limited family of types D (given in Figure 2), so that we do not contaminate our language with infinite loops at every type.[2] We will write recursive functions as syntactic sugar for fix.

The computations themselves include all expressions e, as computations that coincidentally have no side-effects. Furthermore, we have sequential composition $\mathsf{letv}\ x = e\ \mathsf{in}\ c$. Intuitively, the behavior of this command is as follows. We evaluate e until we get some $[c']$, and then evaluate c', modifying the heap and binding the return value to x. Then, in this augmented environment, we run c. The fact that monadic commands have return values explains why our sequential composition is also a binding construct. Finally, we have primitive computations $\mathsf{new}_A(e)$, $!e$, and $e := e'$, which let us allocate, read and write references (inhabiting type $\mathsf{ref}\ A$), respectively. To save space, we will also write $\mathsf{run}\ e$, when e is a term of monadic type, as an abbreviation for $\mathsf{letv}\ x = e\ \mathsf{in}\ x$. Consider the following example, which creates and swaps the contents of two references:

```
1   letv r = [new_ℕ(5)] in
2   letv s = [new_ℕ(14)] in
3   letv x = [!r] in
4   letv y = [!s] in
5   letv _ = [r := y] in
6   letv _ = [s := x] in
7   x + y
```

On lines 1 and 2, we allocate two references to natural numbers, r and s, initializing them with the contents 5 and 14. Then, on lines 3 and 4, we dereference r and s, binding their contents to the variables x and y, respectively. Then, in lines 5 and 6, we swap the contents of the two references, assigning y to r and x to s, so that r now points to 14 and s now points to 5. Finally, on line 7, we return $x + y$, the sum of the two contents.

The primitive commands are all composed using the $\mathsf{letv}\ x = e\ \mathsf{in}\ c$ construct. Because this form expects the bound term to be an expression of monadic type, each of the primitive commands

[1] These quantifiers are actually all restricted to *predicative* quantification (i.e., they can only be instantiated with terms lacking any quantifiers themselves) in order to keep the denotational semantics simple, though recent work (Birkedal et al. 2009) has studied how to combine store with impredicative polymorphism.

[2] The allowed types are those whose interpretations are pointed CPOs in domain theory.

Kinds	κ	::=	$\star \mid \kappa \to \kappa$
Monotypes	τ	::=	$\mathbf{1} \mid \tau \times \tau \mid \tau \to \tau$ $\mathbb{N} \mid \mathsf{ref}\ A \mid \bigcirc \tau \mid$ $\alpha \mid \tau\ \tau \mid \lambda\alpha : \kappa.\ \tau$
Polytypes	A	::=	$\mathbf{1} \mid A \times B \mid A \to B$ $\mathbb{N} \mid \mathsf{ref}\ A \mid \bigcirc A \mid$ $\alpha \mid \tau\ \tau \mid$ $\forall\alpha : \kappa.\ A \mid \exists\alpha : \kappa.\ A$
Type Contexts	Θ	::=	$\cdot \mid \Theta, \alpha : \kappa$

Figure 1. Language Types

Pure expressions	e	::=	$\langle\rangle \mid \langle e, e'\rangle \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e$ $\mid\ x \mid \lambda x : A.\ e \mid e\ e'$ $\mid\ \mathsf{z} \mid \mathsf{s}(e) \mid \mathsf{iter}(e, e_0, x.\ e_1)$ $\mid\ \Lambda\alpha : \kappa.\ e \mid e\ \tau$ $\mid\ \mathsf{pack}(\tau, e) \mid \mathsf{unpack}(\alpha, x) = e\ \mathsf{in}\ e'$ $\mid\ [c] \mid \mathsf{fix}\ x : D.\ e$
Computations	c	::=	$e \mid \mathsf{letv}\ x = e\ \mathsf{in}\ c$ $\mid\ \mathsf{new}_A(e) \mid\ !e \mid e := e'$
Contexts	Γ	::=	$\cdot \mid \Gamma, x : A$
Pointed Types	D	::=	$\mathbf{1} \mid \bigcirc A \mid D \times D \mid A \to D$ $\mid\ \forall\alpha : \kappa.\ D$

Figure 2. Syntax of the Programming Language

(allocation, dereference, assignment) are wrapped in a suspension $[c]$ before being given to the let-binder.

Assertion Language. The sorts and syntax of the assertion language are given in Figure 3. The assertion language is a version of separation logic (Reynolds 2002), extended to higher order.

In ordinary Hoare logic, a predicate describes a set of program states (in our case, heaps), and a conjunction like $p \wedge q$ means that a heap in $p \wedge q$ is in the set described by p and the set described by q. While this is a natural approach, aliasing can become quite difficult to treat — if x and y are pointer variables, we need to explicitly state whether they alias or not. So as the number of variables in a program grows, the number of aliasing conditions grows quadratically.

With separation logic, we add the *spatial* connectives to address this difficulty. A separating conjunction $p * q$ means that the state can be broken into two *disjoint* parts, one of which is in the set described by p, and the other of which is in the set described by q. The disjointness property makes the noninterference of p and q implicit, letting us avoid the unwanted quadratic growth in the size of our assertions. In addition to the separating conjunction, we have its unit emp, which is true of the empty heap, and the points-to relation $e \mapsto e'$, which holds of the one-element heap in which the value of the reference e has contents equal to the value of e'.

The universal and existential quantifiers $\forall x : \omega.\ p$ and $\exists x : \omega.\ p$ are higher-order quantifiers ranging over all sorts ω. The sorts include the language types A, kinds κ, the sort of propositions prop, and function spaces over sorts $\omega \Rightarrow \omega'$. Constructors for terms of all these sorts are given in Figure 3. For the function space, we include lambda-abstraction and application. Because our assertion language contains within it the classical higher-order logic of sets, we will freely make use of features like subsets, indexed sums, and indexed products, exploiting their definability.

Finally, we include the atomic formulas $S\ \mathsf{valid}$, which are *assertions* that a *specification* S holds. This facility is useful when we write assertions about pointers to code — for example, the assertion $r \mapsto e \wedge (\{p\}\mathsf{run}\ e\{a : A.\ q\}\ \mathsf{valid})$ says that the reference r points to a monadic expression e, whose behavior is described by the Hoare triple $\{p\}\mathsf{run}\ e\{a : A.\ q\}$.

Assertion Sorts	ω	::=	$A \mid \kappa \mid \omega \Rightarrow \omega \mid \mathsf{prop}$
Assertion Constructors	p	::=	$e \mid A \mid x \mid \lambda x : \omega.\ p \mid p\ q$ $\mid\ \top \mid p \wedge q \mid p \supset q \mid \bot \mid p \vee q$ $\mid\ \mathsf{emp} \mid p * q \mid e \mapsto e'$ $\mid\ \forall x : \omega.\ p \mid \exists x : \omega.\ p \mid S\ \mathsf{valid}$
Specifications	S	::=	$\{p\}c\{a : A.\ q\} \mid \{p\}$ $\mid\ S\ \mathsf{and}\ S' \mid S\ \mathsf{implies}\ S' \mid S\ \mathsf{or}\ S'$ $\mid\ \forall x : \omega.\ S \mid \exists x : \omega.\ S$

Figure 3. Syntax of Assertions and Specifications

Specification Language. Given programs and assertions about the heap, we need specifications to relate the two. We begin with the Hoare triple $\{p\}c\{a : A.\ q\}$. This specification represents the claim that if we run the computation c in any heap the predicate p describes, then if c terminates, it will end in a heap described by the predicate q. Since monadic computations can return a value in addition to having side-effects, we add the binder $a : A$ to the third clause of the triple to let us name and use the return value in the postcondition.

We then treat Hoare triples as one of the atomic proposition forms of a first-order intuitionistic logic (see Figure 3). The other form of atomic proposition are the specifications $\{p\}$, which are *specifications* saying that an *assertion* p is true. These formulas are useful for expressing aliasing relations between defined predicates, without necessarily revealing the implementations. In addition, we can form specifications with conjunction, disjunction, implication, and universal and existential quantification over the sorts of the assertion language.

With a full logic of triples at our disposal, we can express program modules as formulas of the specification logic. We can expose a module to a client as a collection of existentially quantified functions and variables, and provide the client with Hoare triples describing the behavior of those functions. Furthermore, modules can existentially quantify over predicates to grant client programs access to module state without revealing the actual implementation. A client program that uses an existentially quantified specification cannot depend on the concrete implementation of this module, since the existential quantifier hides that from it.

The language and specification logic have been given a denotational semantics in the first author's forthcoming PhD thesis (Krishnaswami 2009). We do not give the semantics here both for space reasons, and because it is not central to the contributions of this paper.

3. Demand-Driven Notification Networks

A simple intuition for a "demand-driven notification network" is to think of it as a generalized spreadsheet. We have a collection of cells, each of which contains a program expression whose evaluation may read other cells. When a cell is read, the expression within the cell is evaluated, recursively triggering the evaluation of other cells as they are read by the program expression. Furthermore, each cell memoizes its expression, so that repeated reads of the same cell will not trigger re-evaluation.

In addition, we can modify the code expression within a cell. Any time a cell is updated, its memoized value is cancelled, so that any future reads of that cell will force the evaluation of its new code. Furthermore, every cell also maintains a set tracking every other cell which has read it, so that when its code is updated, it can notify all of its readers — i.e., every other cell whose value may depend upon it – to invalidate their own memoized values.

We will call the entire collection of cells a "notification network", because we have a graph (i.e., network) structure of cells, which maintains dependency information between themselves, and whenever a change is made to a cell, it notifies everything that depends on it of the change.

In this section, we will describe an implementation of a notification network library, informally explaining its design and how it works. Then, in the next section, we will see how to take the informal explanation and turn it into a precise specification suitable for verification.

3.1 Implementing Notification Networks

Our API for creating notification networks is given in Figure 4. First, we'll describe the interface, and then discuss its implementation.

The interface exposes two basic abstract data types, cell and code.

The type cell α (defined on line 3) is the type of dynamic data values. A cell contains a reference to a piece of code, a possible memoized value, plus enough information to correctly invalidate its memoized value when the cell's dependencies change. We can create a new cell by calling newcell $\alpha\ e$, which returns a brand new cell with the code expression e inside it. We can also modify a cell with the command update $\alpha\ cell\ e$, which modifies the cell $cell$ by installing the new expression e in it.

The type code α (defined on line 1) is a monadic type, representing the type of computations that can read cells. It supports the usual operations return $\alpha\ e$ and bind $\alpha\ \beta\ e\ (\lambda x.\ e')$, which embed a pure value into the code type and implement sequential composition, respectively. In addition, the primitive operations on this monad include reading a cell with the read $\alpha\ cell$ function call, and reading and modifying local state with the getref $\alpha\ r$ and setref $\alpha\ r\ v$ operations.

This monadic type is not the state monad of the programming language; it is a user-level monadic type we implement as a library (as is commonly done in Haskell, for example), in order to support the transparent propagation and maintenance of dependency information. For example, assuming that a and b are variables of type cell $\mathbb{N}$, then the following code expression will read the two cells and return their sum (suppressing obvious type arguments to functions):

```
1   bind (read a) (λx : ℕ.
2   bind (read b) (λy : ℕ.
3     return (x + y)))
```

This expression does not explicitly mention any dependency information; it is up to the primitive operations of our library to generate it, and up to the return and bind operations to propagate it appropriately. In this way, we can (1) avoid the error-prone business of explicitly managing the dependencies, and (2) we can use typing to forbid invoking arbitrary stateful operations that might ruin the invariants of the library. The only state manipulations we perform are the ones under our control.

The actual implementation is also given in Figure 4. The abstract type of code is implemented using the underlying monad of imperative commands, so that code τ is implemented with the type $\bigcirc(\tau \times \mathsf{cellset})$. The intuition is that when we evaluate a term we are allowed to read some cells along the way, and so we must return a set of all the cells that we read in order to do proper dependency management. So, cellset is a type representing sets of cells[3]. (The precise specification of cellset is given in Appendix A, since describing it is a distraction from the main development.)

[3] Properly, these are sets of cells packed inside an existential type – i.e., terms of type $\exists\alpha : \star.\ \mathsf{cell}\ \alpha$

```
1   code : ⋆ → ⋆
2   code α = ◯(α × cellset)

3   cell : ⋆ → ⋆
4   cell α = {code : ref code α;
5             value : ref option α;
6             reads : ref cellset;
7             obs : ref cellset;
8             unique : ℕ}

9   ecell = ∃α : ⋆. cell α

10  return : ∀α : ⋆. α → code α
11  return α x = [⟨x, emptyset⟩]

12  bind : ∀α, β : ⋆. code α → (α → code β) → code β
13  bind α β e f = [letv (v, r1) = e in
14                  letv (v', r2) = f v in
15                  ⟨v', union r1 r2⟩]

16  read : ∀α : ⋆. cell α → code α
17  read α a = [letv o = [!a.value] in
18              run case(o,
19                  Some v → [⟨v, singleton a⟩],
20                  None→
21                    [letv exp = [!a.code] in
22                     letv (v, r) = exp in
23                     letv _ = [a.value := Some(v)] in
24                     letv _ = [a.reads := r] in
25                     letv _ = iterset (add_observer pack(α, a)) r in
26                     ⟨v, singleton a⟩])

27  getref : ∀α : ref α → code α
28  getref α r = [letv v = [!r] in ⟨v, emptyset⟩]

29  setref : ∀α : ref α → α → code 1
30  setref α r v = [letv _ = [r := v] in ⟨⟨⟩, emptyset⟩]

31  newcell : ∀α : ⋆. code α → ◯cell α
32  newcell α code = [letv unique =!counter in
33                    letv _ = [counter := unique + 1] in
34                    letv code = new_code α(code) in
35                    letv value = new_option α(None) in
36                    letv reads = new_cellset(emptyset) in
37                    letv obs = new_cellset(emptyset) in
38                    (code, value, reads, obs, unique)]

39  update : ∀α : ⋆. code α → cell α → ◯1
40  update α exp a = [letv _ = mark_unready pack(α, a) in
41                    a.code := exp]

42  mark_unready : ecell → ◯1
43  mark_unready cell = unpack(α, a) = cell in
44                      [letv os = [!a.obs] in
45                       letv rs = [!a.reads] in
46                       letv _ = iterset mark_unready os in
47                       letv _ = iterset (remove_obs cell) rs in
48                       letv _ = [a.value := None] in
49                       letv _ = [a.reads := emptyset] in
50                       a.obs := emptyset]

51  add_observer : ecell → ecell → ◯1
52  add_observer a pack(β, b) = [letv os = [!b.obs] in
53                               b.obs := addset os a]

54  remove_obs : ecell → ecell → ◯1
55  remove_obs a pack(β, b) = [letv os = [!b.obs] in
56                             b.obs := removeset os a]
```

Figure 4. Implementation of Notification Networks

Cells, defined on line 3, are represented with a 5-tuple. (We take the liberty of using record syntax for this tuple.) We have a field $code$, which is a reference pointing to the code expression, as well as a field $value$ which is a pointer to an optional value. The value field's contents will be None if the cell is in an unready, un-memoized state, and will be Some v if the cell's code has already been evaluated to a value v. In addition there are two fields representing the dependencies. If the code expression has been evaluated and a memoized value generated, then the $reads$ field will point to the set of cells that the computation directly read while computing its value. Otherwise it will point to the empty set. Conversely, the field obs contains the cell's observers — the set of cells that have read the current cell as part of their own computations. Obviously, this is only non-empty when the cell has been evaluated. Finally, each cell also has a numeric field $unique$, which is a unique numeric identifier for each cell created by the dependency management library in Figure 4. It allows us to compare cells (even of different type) for equality, which we need to implement the cellset type.

The return operation (defined on line 10) for the library simply returns its argument value and the empty set, since it does not read any cells. Likewise, bind $\alpha\ \beta\ e\ f$ (defined on line 12) will evaluate the argument e and pass the returned value to the function f. It will then return the function's return value, together with the union of the two read sets.

There are two functions getref $\alpha\ r$ (line 27) and setref $\alpha\ r\ v$ (line 29) whose specifications say that they simply read and update their argument reference. These two functions allow us to use local state within a notification network, which we will need to implement things like accumulators when implementing reactive programs. They both return empty read sets, since neither of them read any cells.

Interesting things first happen with the read $\alpha\ e$ operation, defined on line 16. This function will first check to see if the cell has a memoized value ready. If it does, we return that immediately. Otherwise, we evaluate the cell's code, and update the current cell's value and read set. In addition, each cell that was read in the evaluation of the code (i.e., the set returned as the second component of the monadic type's return value) also has its observer set updated with the newly-ready current cell. Now, if any of the dependencies change, they will be able to invalidate the current cell, which observes them. Note that the dependencies between cells are all dynamic — we cannot examine the inside of a code expression to find its "free cells", and so we rely upon the invariant that a code expression will return every cell it read, in addition to its return value.

Further interesting things happen with the newcell $\alpha\ e$ operation, defined on line 31. It creates a new cell value, initializing the code field with the argument e, and generating a unique id by dereferencing and incrementing the variable $counter$. The $counter$ variable occurs freely in this definition, because it is a piece of state global to this module, which must be initialized by whatever initialization routine first constructs the whole module as an existential package. Since $counter$ is otherwise private, we can generate unique identifiers by incrementing it as we create new cells.

Finally, the update $cell\ e$ operation (line 39) updates a cell $cell$ with a new code expression e. (As an aside, it's worth noting that this is a genuine, unavoidable, use of higher-order store: we make use of pointers to code, including the ability to dynamically modify them.) Once we modify a cell, any memoized value it has is no longer necessarily correct.

Therefore, we have to drop the memoized value of the cell, and any cell that transitively observes the cell. The mark_unready function (line 42) does this. Given a cell, it takes all of the observers of the current cell and recursively makes all of them unready. Then it removes the current cell from the observer sets of all the cells it reads, and then it nulls out the current cell's memoized value, as well as setting its read and observer sets to empty. Notice that there is no explicit base case to the recursive call; if there are any cycles in the dependency graph, invalidation could go into an infinite loop.

So far, we have described the implementation invariants incrementally. Before proceeding to describe them formally, we will state them again informally, all in one place:

- Every cell must have a unique numeric identifier
- Every cell is either ready, or unready.
- Every ready cell has a memoized value, and maintains two sets, one containing every cell that it reads, and the other containing every cell it is observed by.
- Every unready cell has no memoized value, and has both an empty read set and an empty observer set.
- The overall dependency graph among the valid cells must form a directed acyclic graph.
- The reads and the observers must be the same, only pointing in opposite directions.

Formalizing these constraints is relatively straightforward, but we have the problem that these constraints are global in nature: we cannot be sure that the dependency graph is acyclic without having it all available to examine, and likewise we cannot in general know that a cell is in the read set of everything in its observed set without knowing the whole graph. Handling this difficulty is one of the primary contributions of this work.

4. The Abstract Semantics of Notifications

We will formalize the informal invariants of the previous section in three stages. In the first stage, we will describe how to accurately formalize the global invariant of the cell graph, albeit in a non-modular way. In the second stage, we will recover a basic modular reasoning principle through an interesting use of polymorphism, which will suffice to let us reason modularly about adding cells to and modifying the cells in the cell graph. However, this will not be strong enough to reason modularly about *evaluating* cells in the network, and so in the third stage we will introduce a generalized frame rule, which we will call a "ramified frame rule" after a similar concept in AI.

4.1 The Structure of the Global Invariant

The key to getting around our difficulties lies in the difference between the implementation of update and of read. The update function calls mark_unready, which recursively follows the observers. The read function, on the other hand, proceeds in the opposite direction — it evaluates code expressions, recursively descending into the footprint of its command. The opposite direction these two functions look is why we end up needing a global invariant: we need to know that these two directions are in harmony with one another.

Now, note that we have given the type of mark_unready the monadic type $\bigcirc\mathbf{1}$. This precludes it from being called from within a code τ, because the user-level monadic type discipline of code τ will only let us compute with pure expressions and other code σ terms. Therefore, when we evaluate a code expression, we will never actually follow the observer fields – we will only add entries to them whenever we evaluate a cell and change it from unready to ready. As a result, an abstract description of the heap which *does not explicitly mention the observer sets* will prove sufficient for reasoning about the behavior of code τ expressions.

With this plan, we introduce *abstract heap formulas*, which are syntactic descriptions of the state of part of the cell heap. These syntactic expressions are given by the following grammar:

$$\begin{array}{rcl} \phi, \psi & ::= & I \mid \phi \otimes \psi \mid \mathsf{cell}^+(a, e, v, r) \mid \mathsf{cell}^-(a, e, \_, \_) \\ & \mid & \mathsf{local}(r, v) \end{array}$$

Informally, a formula I represents an empty abstract heap, and a formula $\phi \otimes \psi$ represents an abstract heap that can be broken into two disjoint parts ϕ and ψ. We will only consider formulas modulo the associativity and commutativity of $\otimes$, and take I to be the unit of this binary operator.

The atomic form $\mathsf{local}(r, v)$ says that r is a piece of local state owned by the network, currently with value v. There are two atomic forms representing cells. $\mathsf{cell}^-(a, e, \_, \_)$ says that a is a cell with code e, which is unready to deliver a value — it needs to be re-evaluated before it can yield a value. $\mathsf{cell}^+(a, e, v, r)$ says that a is a cell with code e. Furthermore, it is ready to deliver the value v, but only if all the cells in its read set r are themselves ready. Otherwise, if anything in a's read set r is unready, then a is unready itself. (Because we will sometimes want to write $\mathsf{cell}^{\pm}(a, e, -, -)$ when we do not care whether a is ready or not, the $\mathsf{cell}^-(a, e, \_, \_)$ formula has two dummy argument positions.)

First, notice the must/may flavor of this reading. The formula $\mathsf{cell}^-(a, e, \_, \_)$ says that a *must* be unready. The formula $\mathsf{cell}^+(a, e, v, r)$ says that a *may* be ready, conditional on the readiness of the elements of its read set r. Second, notice that the backwards dependencies are entirely missing from these formulas. We have simply left out the other half of the dependency graph from this description. Forgetting this information will let us begin to regain local reasoning, as we will see in the statements of Propositions 1 and 2.

We have emphasized that the straightforward invariant is not obviously modular. To elaborate upon this point, we will need to look at the formal statement of the heap invariant, to see how exactly modularity fails. We introduce the predicate $G(\phi)$. This predicate describes the entire heap of cells allocated by our library and ensures they satisfy the conditions described at the end of the previous section. It also enforces the additional constraint that the cell heap agree with ϕ.[4]

$$G(\phi) \triangleq \exists H \in CellHeap.\ Inv(H, \phi)$$

$$\begin{array}{l} Inv(H, \phi) \triangleq \\ \quad R_H^\dagger = O_H \wedge R_H^+ \text{ strict partial order} \\ \quad \wedge\ R_H \subseteq V_H \times V_H \wedge uniqueids(H) \\ \quad \wedge\ satisfies(H, \phi) \wedge heap(H) * localstate(\phi) \end{array}$$

The auxiliary definitions we used in this definition are all given in Figure 5.

We first assert the existence of a cell heap H drawn from the set $CellHeap$. An element of $CellHeap$, defined on line 1 of Figure 5, is a collection of cells, paired with a function mapping each cell in that collection to a code expression, a possible value, a read set, an observed set, and an identifier. We will use H as a variable ranging over cell heaps, and will use the pair pattern (D, h) to range over cell heaps when we need to use the individual components of the pair.

In the first two lines of $Inv(H, \phi)$, we assert all of the global conditions in terms of the mathematical cell heap H. First, we assert that the relational transpose $(\cdot)^\dagger$ of the reads relation R_H is the observes relation O_H. These two relations (defined in lines 9 and 10) are computed from the cell heap. R_H consists of those pairs of cells in H, such that the first component reads the second component. Likewise, O_H consists of those pairs of cells in H such that the first component is observed by the second component. Requiring that $R_H = O_H^\dagger$ enforces the condition that the reads and observe relations be the same, only pointing in opposite directions (i.e., if a reads b, then b is observed by a).

Then, we require that the transitive (but not reflexive) closure of the reads relation, R_H^+ form a strict partial order. Strictness enforces the condition that there be no cycles in the dependence graph (because otherwise there could be elements a, such that $(a, a) \in R_H^+$). Next, we require that the reads relation R_H is a subset of the Cartesian product $V_H \times V_H$ of the set V_H of cells carrying values (defined on line 8). This ensures that (1) there are no dependencies on unready cells, and (2) all unready cells have empty read and observe sets.

Finally, we ask that all of the cells in H have unique identifiers — $uniqueids(D, h)$ (defined on line 11) asserts that there is a bijective map between $Fin(|D|)$ (the finite set consisting of the natural numbers from 0 to the size of the cell heap) and the cells in D, and that each cell carry its uniquely identifying number in its $unique$ field.

```
1  CellHeap =
2    ΣD ∈ P^fin(ecell).
3      (Π(pack(α, _)) ∈ D.(code α × option α ×
4                          P^fin(ecell) × P^fin(ecell) × ℕ)
5  code = π1            obs = π4
6  value = π2           unique = π5
7  reads = π3
8  V_(D,h) = {c ∈ D | ∃v. value(h(c)) = Some(v)}
9  R_(D,h) = {(c, c') ∈ D × D | c' ∈ reads(h(c))}
10 O_(D,h) = {(c, c') ∈ D × D | c' ∈ obs(h(c))}
11 uniqueids(D, h) = ∃i : Fin(|D|) → D.
12                     i ∘ (unique ∘ h) = id ∧
13                     (unique ∘ h) ∘ i = id
14 satisfies((D, h), φ) = sat((D, h), D, φ)
15 sat((D, h), D', local(r, v)) = ⊤
16 sat((D, h), D', I) = ⊤
17 sat((D, h), D', φ ⊗ ψ) = ∃D1, D2. D = D1 ⊎ D2
18                                  ∧ sat((D, h), D1, φ)
19                                  ∧ sat((D, h), D2, ψ)
20 sat((D, h), D', cell⁻(a, e, _, _)) =
21       a ∈ D ∧ code(h(a)) = e ∧ a ∉ V_H
22 sat((D, h), D', cell⁺(a, e, v, r)) =
23       a ∈ D ∧ code(h(a)) = e ∧
24       (if r ∩ V_H = r
25        then value(h(a)) = Some v ∧ reads(h(a)) = r
26        else a ∉ V_H)
27 heap(D, h) =
28   counter ↦ |D| *
29   ∀*c ∈ D. ∃v_r, v_o : cellset.
30              c.code ↦ code(h(c)) *
31              c.value ↦ value(h(c)) *
32              c.reads ↦ v_r *
33              c.obs ↦ v_o *
34              c.unique = unique(h(c)) ∧
35              set(D, v_r, reads(h(c))) ∧
36              set(D, v_o, obs(h(c)))
37 localstate(cell±(a, e, −, −))= emp
38 localstate(I)                = emp
39 localstate(φ ⊗ ψ)           = localstate(φ) * localstate(ψ)
40 localstate(local(r, v))     = r ↦ v
```

Figure 5. Definitions for Heap Invariant

[4] This is why we insisted that the abstract heap formulas are syntactic objects — this permits us to define predicates on them by induction over the structure of the formula.

In the third line of the definition of $Inv(H, \phi)$, we begin by requiring that the cell heap H satisfy the abstract heap formula ϕ, which formalizes the informal reading of the abstract heap formulas given earlier. The definition of the satisfaction relation is given on line 14. The satisfaction relation $satisfies(H, \phi)$ asserts that ϕ describes the heap H.

This relation closely follows the standard pattern of separation logic, with one exception: we need to remember the whole heap in order to check whether or not the read sets of positive cell formulas like $\mathsf{cell}^+(a, e, v, rs)$ are ready. To model this, we use an auxilliary relation $sat((D, h), D', \phi)$, in which (D, h) is the whole heap, and D' is the fragment of the heap within which ϕ must lie. The case for the unit I, on line 16, is satisfied by any cell heap, and the tensor $\phi \otimes \psi$ (on line 17) is satisfied if we can break D' into two disjoint pieces, one of which satisfying ϕ and the other satisfying ψ. In this sense, we are building a domain-specific separation logic on top of separation logic. The clause for $\mathsf{cell}^-(a, e, \_, \_)$, on line 20, says that (1) the cell a must be within D', (2) its code must be e, and (3) it must be unready (i.e., have no value). The clause for $\mathsf{cell}^+(a, e, v, r)$ (on line 22) is a little more complex. It also says that a must be in D' and that a's code must be e. In addition, it says that if all the cells in r have values, then a's value must be v and otherwise a must not have a value. The clause for $\mathsf{local}(r, v)$ is simply the true assertion — since this is a piece of local state that does not participate in dependency tracking, we leave it out of this invariant and use an ordinary separation logic formula to track it.

The second-to-last clause of $Inv(H, \phi)$ is the predicate $heap(H)$. This predicate, defined on line 27, finally connects the cell heap, which is a purely mathematical object, to the actual low-level heap the implementation uses. We ask that the global counter reference $counter$ point to an integer field equal to the size of the cell heap, and then use the iterated separating conjunction $\forall^*$ to require that for each cell in the cell heap, we have pointers to the appropriate code, value, read, observer, and unique identifier fields. The $unique$ identifier field contains the natural number uniquely identifying the cell. The read and observer fields point to values of type cellset, with the predicate $set(D, v_r, reads(h(c)))$ and $set(D, v_o, obs(h(c)))$ asserting that the program value v_r and v_o respectively representing the read and observed sets of the cell. (This predicate is explained in the appendix, as part of the specification of sets of cells.)

The last clause in $Inv(H, \phi)$ is $localstate(\phi)$, which finds each local reference formula in ϕ and asserts that it is in the physical heap.

The global character of this invariant should be evident; we describe *all* of the cells in the heap at once in order to state our invariants. So it is not immediately clear that we have made much progress towards a modular proof technique. However, we are actually very close: with just two more ideas, we will be able to give a solution to this problem.

4.2 Frame Properties via Polymorphism

As we mentioned earlier, our abstract heap formulas essentially give us a small domain-specific separation logic. This means that in order to reason locally over cell heaps, we need to find an application-specific version of the frame rule for our library.

To do this, we will adapt some ideas proposed by Birkedal et. al. (Birkedal et al. 2005). They suggested interpreting the frame rule of separation logic as a form of quantification — instead of having a separate frame rule that allows adding a frame to any triple, they proposed that all of the atomic rules of the program logic be replaced with rules possessing an extra quantifier ranging over "the rest of the heap":

$$\frac{}{\forall R.\ \{(e \mapsto v) * R\}\ e := v'\ \{(e \mapsto v') * R\}}\ \textsc{Example}$$

This quantifier is propagated through the proof, and any use of the frame rule can be interpreted as instantiating the universal quantifier appropriately. The reason this idea is fruitful for us is that it will allow us to give a frame rule, even though the underlying semantics of our library does not actually satisfy any analogues of the traditional safety, monotonicity, and frame lemmas. For example, the update operation certainly does not act locally – it recursively traverses the observers set, possibly mutating a very large part of the cell graph.

Nonetheless, we can prove the soundness of the following triple specifying update.

PROPOSITION 1. *(Update Rule) For all appropriately-typed cells o and code expressions e and e', the following triple is derivable in our specification logic:*

$$\begin{array}{l}\forall \psi : \mathit{formula}.\ \{G(\mathsf{cell}^{\pm}(o, e', -, -) \otimes \psi)\} \\ \qquad\qquad \mathit{run}\ \mathsf{update}\ o\ e \\ \qquad\qquad \{a : 1.\ G(\mathsf{cell}^{-}(o, e, \_, \_) \otimes \psi)\}\end{array}$$

Proof (Sketch). The key to this proof is the conditional interpretation of the $\mathsf{cell}^+(c, e, v, r)$ formula. When the update o e command executes, it recursively finds every cell which depends on o, and modifies it to be unready.

Now consider any positive cell formula in ψ which depends on o, directly or indirectly. The satisfaction relation for ϕ asserts that in order for a positive cell formula to represent a ready cell, everything in its read set also has to be ready. So when o's formula switches to the unready state, we now require that every positive formula depending on o represents an unready cell — which is exactly the effect of executing update. As a result, we can leave the entire frame ψ untouched, even though the physical heap it represents may have been (quite drastically) modified, and many cells may have gone from a ready to an unready state. □

We can prove the soundness of a similar specification for newcell as well:

PROPOSITION 2. *(New Cell Rule) For all code expressions e, the following specification is derivable in our specification logic:*

$$\begin{array}{l}\forall \psi : \mathit{formula}.\ \{G(\psi)\} \\ \qquad\qquad \mathit{run}\ \mathsf{newcell}\ \ e \\ \qquad\qquad \{a : \mathsf{cell}\ \tau.\ G(\mathsf{cell}^{-}(a, e, \_, \_) \otimes \psi)\}\end{array}$$

Proof (Sketch). This is much easier than update: after newcell allocates a new numeric id for the new cell, we can extend the cell heap with the new cell and show that it continues to satisfy the invariant. □

As we can see, the conditional interpretation of $\mathsf{cell}^+(a, e, v, r)$ gives us quite a strong modular reasoning property for update and $newcell$ – we can simply pretend that we are locally changing or creating a cell, and leave the frame unchanged. This property lets us write programs whose components independently modify the cell heap, without having to know what cells might be updated by the change.

4.3 Ramified Frame Properties

While this strategy is sufficient for newcell and update, it is not adequate for defining a frame property for code τ expressions.

As an example, suppose that we want to evaluate the code expression read τ a, in a cell heap described by $\mathsf{cell}^-(a, \mathsf{return}\ 5, \_, \_)$. Clearly, this is a sufficient footprint, and we expect to get the return value 5, and see the cell formula change to $\mathsf{cell}^+(a, \mathsf{return}\ 5, 5, \emptyset)$. However, the fact that we are now changing cells from negative to

$$\frac{\forall a' \in r.\ \exists v'.\ \mathsf{ready}(\phi, a', v')}{\mathsf{ready}(\phi \otimes \mathsf{cell}^+(a, e, v, r), a, v)}\ \textsc{Ready}$$

$$\frac{\exists a' \in r.\ \mathsf{unready}(\phi, a')}{\mathsf{unready}(\phi \otimes \mathsf{cell}^+(a, e, v, r), a)}\ \textsc{UnreadyPos}$$

$$\frac{}{\mathsf{unready}(\phi \otimes \mathsf{cell}^-(a, e, \_, \_), a)}\ \textsc{UnreadyNeg}$$

Figure 6. Ready and Unready Judgments

$$\begin{array}{lcl}
closed(I, s) & = & \top \\
closed(\phi \otimes \psi, s) & = & closed(\phi, s) \wedge closed(\psi, s) \\
closed(\mathsf{local}(r, v), s) & = & \top \\
closed(\mathsf{cell}^-(a, e, _, _), s) & = & \top \\
closed(\mathsf{cell}^+(a, e, v, r), s) & = & r \subseteq s
\end{array}$$

Figure 7. Closedness predicate

$$\begin{array}{lcl}
R(s, I) & = & I \\
R(s, \phi \otimes \psi) & = & R(s, \phi) \otimes R(s, \psi) \\
R(s, \mathsf{local}(r, v)) & = & \mathsf{local}(r, v) \\
R(s, \mathsf{cell}^-(a, e, _, _)) & = & \mathsf{cell}^-(a, e, _, _) \\
R(s, \mathsf{cell}^+(a, e, v, r)) & = & \begin{cases} \mathsf{cell}^+(a, e, v, r) & \text{if } s \cap r = \emptyset \\ \mathsf{cell}^-(a, e, _, _) & \text{otherwise} \end{cases}
\end{array}$$

Figure 8. Definition of the Ramification Operator R

positive means that the conditional character of readiness, which worked in our favor with update and newcell, now works against us.

In particular, suppose that we run this command with a framed abstract heap formula $\psi = \mathsf{cell}^+(b, \mathsf{read}\ a, 17, \{a\})$. Now, the whole starting heap will be described by the formula:

$$\mathsf{cell}^-(a, \mathsf{return}\ 5, \_, \_) \otimes \mathsf{cell}^+(b, \mathsf{read}\ a, 17, \{a\})$$

In any heap satisfying this formula, b will be unready, because it depends on an unready cell. But when we execute read a, simply copying ψ into the post-state will give us the cell formula:

$$\mathsf{cell}^+(a, \mathsf{return}\ 5, 5, \emptyset) \otimes \mathsf{cell}^+(b, \mathsf{read}\ a, 17, \{a\})$$

That is, our satisfaction relation now expects b to be ready and have the value 17, even though read a never touches b at all!

Clearly, we cannot expect to be able to simply copy the same frame formula into the pre- and the post-condition states in the specification of commands like read a.

To deal with this problem, we look back to the original paper introducing the frame problem (McCarthy and Hayes 1969). They described the frame problem as the problem of how to specify what parts of a state were *unaffected* by the action of a command, which inspired the name of the frame rule in separation logic. In that paper, he also described the *qualification problem*. He observed that many commands (such as a flipping a light switch turning on a light bulb) have numerous implicit preconditions (such as there being a bulb in the light socket), and dubbed the problem of identifying these implicit preconditions the qualification problem.

Some years later, Finger (1987) observed that the qualification problem has a dual: actions can have indirect effects that are not explicitly stated in their specification (e.g., turning on the light can startle the cat). He called the problem of deducing these implicit consequences the "ramification problem" — is there a simple way to represent all of the indirect consequences of an action?

We can understand our difficulty as an instance of the ramification problem. When we evaluate a code expression, we may read some unready cells and send them from an unready state in the precondition to a ready state in the postcondition. However, we may have had some cell formulas in our frame which claimed their corresponding cells were unready purely because one of the cells in our footprint was unready. Therefore, when we update the footprint, we must modify the frame formula to account for the ramifications of our update in the footprint. So even though the actual physical storage representing the frame does not change at all, we need to modify our abstract formula to reflect our updated state of knowledge.

In our case, *all* of the effects on the frame will arise from the cell formulas we change from unready to ready. Thus, given the set of cells which became ready, we can repair the framing formula by taking each positive cell formula, and setting it to a negative state if its read set includes anything that went from unready to ready. We define the ramification operator $R(s, \psi)$ in Figure 8. It is a simple structural induction over a framing formula, whose only action is to replace the positive cell formulas in ψ whose read sets intersect with s with a corresponding negative cell formula. The ramification operator has a number of useful properties, which are most easily expressed after we have introduced a few auxiliary judgments and predicates.

In Figure 6 we define the two judgments $\mathsf{unready}(\phi, o)$ and $\mathsf{ready}(\phi, o, v)$, which establish whether a cell is ready or unready, from the syntactic structure of ϕ. $\mathsf{ready}(\phi, o, v)$ is intended to mean that the cell o is ready and will return value v, in any heap described by ϕ. Correspondingly, $\mathsf{unready}(\phi, o)$ means that o is not ready in any heap described by ϕ. Since these are purely syntactic judgments, we need to show that they are consistent with heaps described by ϕ.

PROPOSITION 3. *(Soundness of $\mathsf{ready}(\phi, o, v)$ and $\mathsf{unready}(\phi, o)$) For all ϕ, o, and H such that $H = (D, h)$, the following assertions are tautologies in separation logic.*

- $(Inv(H, \phi) \wedge \mathsf{ready}(\phi, o, v)) \supset value(h(o)) = \mathsf{Some}\ v$
- $(Inv(H, \phi) \wedge \mathsf{unready}(\phi, o)) \supset o \notin V_H$

Next, in Figure 7, we define the $closed(\phi, s)$ predicate, which asserts that every cell formula in ϕ reads at most the cells in s. Now, we can summarize the interactions between the ramification operator R and abstract heap formulas as follows:

PROPOSITION 4. *(Interaction Properties) Given sets of cells s and u, cell o, value v, and formula ϕ, we have that:*

- $R(s, R(u, \phi)) = R(s \cup u, \phi)$
- *If $\mathsf{unready}(\phi, o)$, then $\mathsf{unready}(R(u, \phi), o)$*
- *If $\mathsf{ready}(R(u, \phi), o, v)$, then $\mathsf{ready}(\phi, o, v)$*
- *If $closed(\phi, s)$, then $R(u, \phi) = R(u \cap s, \phi)$*

All of these facts can be proved with simple inductive arguments, since they are all syntactic facts.

The first property means that if we evaluate two expressions, we can simply combine their ramification effects without having to worry about the order that they were evaluated in. The second and third let us know that a ramification cannot make us forget a cell is unready, nor can it make anything ready that was not ready before. The last property permits us to constrain the effect of a ramification — if we know that two parts of the abstract heap formula do not read each other at all, we can deduce that ramifications from one will not affect the other.

Now we can define the abstract semantics of the code monad. We introduce the "judgment" $\langle \phi; e \rangle \Downarrow \langle \phi'; v \rangle\ [r|u]$, which is read

$$\langle \phi; e\rangle \Downarrow \langle \phi'; v\rangle\,[r|u] \triangleq$$
$$\forall \psi.\ \{G(\phi \otimes \boxed{\psi})\}$$
$$\quad \mathsf{run}\ e$$
$$\quad \{a : \tau.\ G(\phi' \otimes \boxed{R(u,\psi)}) \wedge \exists z.\ a = (v,z) \wedge set(u,z,u)\}$$
$$\quad \mathsf{and}$$
$$\quad \{\forall o.\ (\mathsf{unready}(\phi \otimes \psi, o) \wedge o \in \mathrm{dom}(\psi)) \supset \mathsf{unready}(\phi' \otimes R(u,\psi), o)\}$$
$$\mathsf{and}$$
$$\{\forall c \in r \cup u.\ \exists v.\ \mathsf{ready}(\phi', c, v)\ \wedge \forall c \in u.\ \mathsf{unready}(\phi, c)\}$$

Figure 9. Definition of the Abstract Semantics

$$\frac{}{\langle I; \mathsf{return}\ \alpha\ v\rangle \Downarrow \langle I; v\rangle\,[\emptyset|\emptyset]}\ \textsc{AUnit}$$

$$\frac{\langle \phi; e\rangle \Downarrow \langle \phi'; v'\rangle\,[r_1|u_1] \qquad \langle \phi'; f\ v'\rangle \Downarrow \langle \phi''; v''\rangle\,[r_2|u_2]}{\langle \phi; \mathsf{bind}\ \alpha\ \beta\ e\ f\rangle \Downarrow \langle \phi''; v''\rangle\,[r_1 \cup r_2|u_1 \cup u_2]}\ \textsc{ABind}$$

$$\frac{\mathsf{ready}(\phi, a, v)}{\langle \phi; \mathsf{read}\ \alpha\ a\rangle \Downarrow \langle \phi; v\rangle\,[\{a\}|\emptyset]}\ \textsc{AReady}$$

$$\frac{\mathsf{unready}(\mathsf{cell}^{\pm}(a,e,-,-) \otimes \phi, a) \quad \langle \phi; e\rangle \Downarrow \langle \phi'; v\rangle\,[r|u]}{\begin{array}{c}\langle \mathsf{cell}^{\pm}(a,e,-,-) \otimes \phi; \mathsf{read}\ \alpha\ a\rangle \Downarrow \\ \langle \mathsf{cell}^{+}(a,e,v,r) \otimes R(\{a\},\phi'); v\rangle\,[\{a\}|u \cup \{a\}]\end{array}}\ \textsc{AUnready}$$

$$\frac{}{\langle \mathsf{local}(r,v); \mathsf{getref}\ r\rangle \Downarrow \langle \mathsf{local}(r,v); v\rangle\,[\emptyset|\emptyset]}\ \textsc{AGetRef}$$

$$\frac{}{\langle \mathsf{local}(r,v); \mathsf{setref}\ r\ v'\rangle \Downarrow \langle \mathsf{local}(r,v'); \langle\rangle\rangle\,[\emptyset|\emptyset]}\ \textsc{ASetRef}$$

$$\frac{\langle \phi; e\rangle \Downarrow \langle \phi'; v\rangle\,[r|u]}{\langle \phi \otimes \psi; e\rangle \Downarrow \langle \phi' \otimes R(u,\psi); v\rangle\,[r|u]}\ \textsc{AbstractFrame}$$

Figure 10. Abstract Semantics of Notifications

as "from an initial state ϕ, evaluating the code τ expression e will result in a modified state ϕ' and a return value v of type τ. The expression e will have directly read the cells in r, and will have evaluated the cells in u, sending them from an unready to a ready state." The formal definition of the meaning is given in Figure 9, where our "judgment" is revealed to be a notational abbreviation for a formula in our logic of specifications.

The step relation $\langle \phi; e\rangle \Downarrow \langle \phi'; v\rangle\,[r|u]$ really means three things. First, it means that if we run e in a heap $G(\phi \otimes \psi)$, then we will end in a heap $G(\phi' \otimes R(u,\psi))$, and that the return value will be a pair consisting of the value v and the read set u (that is, the return value is a pair (v,z), and the second component of the return value z is a cellset which represents the set u, as indicated by the predicate $set(u,z,u)$). Note the use of quantification over ψ to describe the frame, and that furthermore we need to use the ramification operator to describe the change in the frame in the postcondition.

Second, we assert that anything which was unready in the frame, remains unready in the frame in the postcondition, which is a way of saying that we never read anything outside the footprint ϕ. Third, we assert that everything that was read or evaluated (the cells in r or u) is really syntactically ready in the postcondition, and that everything we claim we evaluated (i.e., the cells in u) was really syntactically unready in the precondition.

The reason we have to maintain these conditions is that the readiness judgments are syntactic derivations which do not know anything about the effect of the execution of the command e. So we need to explicitly construct syntactic facts reflecting any changes in the semantic heap if we wish to use them in further reasoning about the syntactic description. In particular, we need these facts to prove the rules given in Figure 10, where we finally show that terms of type code α behave as we claimed in our informal description. We give a number of implications over specifications in inference rule format, mimicking the structure of a big-step semantics. Again, we emphasize that these rules are really just implications in specification logic — the inference rule format is just a suggestive notation for the actual specifications we use.

In rule AUNIT, we give a specification for the return command, which simply returns its argument and neither reads nor updates any cells or state. The ABIND rule explains how sequential composition works — as expected, we evaluate the first monadic argument, and pass the result to the functional argument, and evaluate that. The read and update sets are simply the union of the two executions. Reading a cell comes in two variants, AREADY and AUNREADY. If a cell is ready, we simply return its memoized value without any further computation. If a cell is unready, we need to evaluate its code body, and then update the cell with its new value. Note that we have to apply the ramification operator in AUNREADY, because the cell we are reading goes from unready to ready itself. We can also read (AGETREF) and write (ASETREF) local state, which do not have any effect on the cells.

Finally, we have the ABSTRACTFRAME rule, which allows us to extend the abstract heap formulas in the style of the frame rule of separation logic — the signal difference being that we have to apply the ramification operator R to the frame in the post-state.

PROPOSITION 5. *(Soundness of Abstract Semantics) All of the rules of the abstract semantics in Figure 10 are derivable in our specification logic.*

We can now reason about the behavior of our imperative notification network library in terms of its action on the abstract heap. Quantification and ramification give us a domain-specific frame property, which allows us to modularly prove the correctness of programs that construct and use this dependency-tracking library.

5. Verifying an Imperative Implementation of Functional Reactive Programming

In this section, we will see how to verify an imperative implementation of a simple synchronous functional reactive programming system. We will begin by giving the purely functional/mathematical semantics of stream transducers. This semantics is easy to reason about, but too inefficient to consider as an implementation. Then, we'll give an imperative implementation of the reactive primitives, which is intended to be driven by an event loop. Finally, we will prove the correctness of a realization relation between the imperative and functional implementations, which will let us reason about the imperative implementation as if it were functional.

5.1 Specifying Functional Reactive Programs

Functional Reactive Programming (Elliott and Hudak 1997) is a style of writing interactive programs based on the idea of *stream transducers*. The idea is to model a time-varying input signal of type A as an infinite stream of A's, and to model an interactive system as a function that takes a stream of inputs $\mathsf{stream}(A)$ and yields a stream of outputs $\mathsf{stream}(B)$. Note that a stream can be viewed either as an infinite sequence of values, or isomorphically as a function from natural numbers to values (i.e., a function from

```
1   ST(A, B) = {f ∈ stream(A) → stream(B) | causal(f)}
2   lift : (A → B) → ST(A, B)
3   lift f as = map f as
4   seq : ST(A, B) → ST(B, C) → ST(A, C)
5   seq p q = q ∘ p
6   par : ST(A, B) → ST(C, D) → ST(A × C, B × D)
7   par p q abs = zip (p (map π1 abs)) (q (map π2 abs))
8   switch : ℕ → ST(A, B) → ST(A, B) → ST(A, B)
9   switch k p q = λas. (take k (p as)) · (q (drop k as))
10  loop : A → ST(A × B, A × C) → ST(B, C)
11  loop a0 p = (map π2) ∘ (cycle a0 p)
12  cycle : A → ST(A × B, A × C) → ST(B, A × C)
13  cycle a0 p = λbs. λn. last(gen a0 p bs n)
14  gen : A→ ST(A × B, A × C) → stream(B)
15            → ℕ → list (A × C)
16  gen a0 p bs 0     = p̂ [(a0, bs0)]
17  gen a0 p bs (n + 1) =
18    p̂ (zip(a0 :: (map π1 (gen a0 p bs n))) (take (n + 2) bs))
```

Figure 11. Semantics of Stream Transducers

times to values). In our discussion, we'll switch freely between these two views, using the most convenient viewpoint.[5]

However, not all functions $\mathsf{stream}(A) \to \mathsf{stream}(B)$ are legitimate stream transducers: we need to restrict our attention to *causal* functions. A transducer is causal if we can compute the first n elements of the output having read at most n elements of the input.

$$causal(f : \mathsf{stream}(A) \to \mathsf{stream}(B)) \equiv$$
$$\exists \hat{f} : \mathsf{list}\ A \to \mathsf{list}\ B.\ \forall as : \mathsf{stream}(A), n : \mathbb{N}.$$
$$take\ n\ (f\ as) = \hat{f}\ (take\ n\ as)$$

Given a causal transducer p, we will write $\hat{p}$ to indicate the corresponding list function which computes its finite approximations. In Figure 11, we define a family of combinators acting on causal transducers.

The type $\mathsf{ST}(A, B)$ of stream transducers (defined on line 1) is the set of causal functions from $\mathsf{stream}(A)$ to $\mathsf{stream}(B)$. The operation *lift* f (line 2) creates a stream transducer that simply maps the function f over its input. Calls to $seq\ p\ q$ (line 4) are sequential composition: it feeds the output of p into the input of q. The operator $par\ p\ q$ (line 6) defines parallel composition — it takes a stream of pairs, and feeds each component to its arguments, respectively, and then merges the two output streams to produce the combined output stream. The function $switch\ k\ p\ q$ (line 8) is a very simple "switching combinator". It behaves as if it were p for the first k time steps, and then behaves as if it were q, only starting with the input stream beginning at time k.

The combinator $loop\ a_0\ p$ is a feedback operation. It acts upon a transducer p which takes pairs of As and Bs, and yields pairs of As and Cs. It turns it into a combinator that takes Bs to Cs, by giving p the value a_0 (and its B-input) on the first time step, and uses the output A at time n as the input A at time $n + 1$. This is useful for constructing transducers that do things like sum their inputs over time, and other stateful operations.

Because this function involves feedback, it should not be surprising that it makes use of the causal nature of its argument operation. The $loop$ function is defined in terms of $cycle$, which also returns the sequence of output As, and $cycle$ is defined in terms of gen, which is a function that given an argument n returns a list of outputs for the time steps from 0 to n. Notice that $gen\ a_0\ p\ bs\ n$ will always return $n+1$ elements (e.g., at argument 0, it will return a 1 element list containing the output at time step 0), which means that the call to $last$ in $cycle$ is actually safe. In order to calculate gen, we need to recursively calculate the outputs for all smaller time steps, and this is why p must be a causal stream function — we need to be able to call the approximation $\hat{p}$ to operate on the list consisting of the first n elements.

All of these definitions are familiar to functional programmers, and there are many techniques to prove properties of these functions — coinductive proofs, the $take$-lemma of Bird and Wadler (1988), arguments based on the isomorphism between streams and functions from natural numbers. All of these serve to make proving properties about stream transducers very pleasant. For example, one property we will need in the next section is the following:

LEMMA 1. *(Loop Unrolling) We have that*

$$cycle\ a_0\ p\ bs = p\ (zip\ (a_0 :: (map\ \pi_1\ (cycle\ a_0\ p\ bs)))\ bs)$$

Proof (Sketch). This is easily proved using $take$-lemma of Bird and Wadler (1988), which states that two streams are equal if all their finite prefixes are equal. □

5.2 Realizing Stream Transducers with Notifications

While the definitions in the previous subsection yield very clean proofs, they are not suitable as implementations — e.g., *loop* recomputes an entire history at each time step! We can derive better implementations by looking at how imperative event loops work.

The intuition underlying event-driven programming is that a stream transducer is implemented with the combination of a notification network, and an *event loop*. The event loop is a (possibly-infinite) loop which does the following on each time step. First, it updates an input cell, to reflect any input events that occurred on that time step. Then, it reads the output cell of the network, to discover the output value for that time step. When the input cell is updated, invalidations propagate throughout the dependency network, and when the outputs are read, only the necessary re-computations are performed.

Before formalizing this idea, we will first discuss the implementation given in Figure 12 in informal terms. (In the definitions, we suppress the type arguments to functions in order to make the code more readable.) We define the type of imperative stream transducers from types α to β as the type $\mathsf{cell}\ \alpha \to \bigcirc(\mathsf{cell}\ \beta)$. This type should be read as saying that the implementation is a function that, given an input cell of type A, will *construct* a dataflow network realizing a transducer, and which returns the output cell of type B that the event loop should read.

The simplest example of this is $\mathsf{lift}\ f$, defined on line 3. It will take an input cell $input$, and build a new cell which reads $input$, and return f applied to that value. Likewise, given two imperative implementations p and q, $\mathsf{seq}\ p\ q$ (defined on line 6) will take an input cell, and feed the input to p to build a network whose output is named $middle$, and will then give $middle$ to q to get the final output cell. The overall network will be the network built by the calls to both p and q, which interact via p's network installing a value in $middle$, which q's network reads and processes.

The operation $\mathsf{switch}\ k\ p\ q$ (defined on line 23) is the first example that uses local state. Given an $input$ cell, we first create a

[5] Given an infinite stream vs, we will use use $take\ n\ vs$ to denote the finite list consisting of the first n elements of the stream vs. Correspondingly, $drop\ n\ vs$ is the infinite stream with vs with its first n elements cut off. With a function f, $map\ f\ vs$ maps f over the elements of vs, and given another infinite stream us, the call $zip\ us\ vs$ returns the infinite stream of pairs of elements of us and vs. If v is an element, $v :: vs$ will denote consing v to the front of vs, and if xs is a finite list, then $xs \cdot vs$ will denote appending the finite sequence xs to the front of vs. Finally, we will write vs_n to denote the n-th element of the stream vs, and $last\ xs$ to denote the last element of a non-empty finite list.

```
1  ST : ⋆ → ⋆ → ⋆
2  ST(α, β) ≡ cell α → ◯cell β

3  lift : ∀α, β : ⋆. (α → β) → ST(α, β)
4  lift α β f input =
5    newcell (bind (read input) (λx : α. return (f x)))

6  seq : ∀α, β : ⋆. ST(α, β) → ST(β, γ) → ST(α, γ)
7  seq α β p q input = [letv middle = p input in
8                       letv output = q middle in
9                        output]

10 par : ∀α,β,γ,δ : ⋆.
11          ST(α, β) → ST(γ, δ) → ST(α × γ, β × δ)
12 par α β γ δ p q input =
13   [letv a = newcell (bind (read input)
14                           (λx : α × β. return (fst x))) in
15    letv b = p a in
16    letv c = newcell (bind (read input)
17                           (λx : α × β. return (snd x))) in
18    letv d = q c in
19    letv output = newcell (bind (read b) (λb : β.
20                           bind (read d) (λd : δ.
21                             return ⟨b, d⟩)))] in
22    output]

23 switch : ∀α, β : ⋆. ℕ → ST(α, β) → ST(α, β) → ST(α, β)
24 switch α β k p q input =
25   [letv r = newℕ(0) in
26    letv a = p input in
27    letv b = q input in
28    letv out = newcell (bind (getref r) (λi : ℕ.
29                        bind (setref r (i + 1)) (λq : 1.
30                          if(i < k, read a, read b)))) in
31     out]

32 loop : ∀α, β, γ : ⋆. α → ST(α × β, α × γ) → ST(β, γ)
33 loop α β γ a0 p input =
34   [letv r = newα(a0) in
35    letv ab = newcell (bind (read input) (λb : β.
36                       bind (getref r) (λa : α.
37                         return ⟨a, b⟩))) in
38    letv ac = p ab in
39    letv c = newcell (bind (read ac) (λv : α × γ.
40                      bind (setref r (fst v)) (λq : 1.
41                        return (snd v)))) in
42     c]
```

Figure 12. Imperative Stream Transducers

local reference r, initialized to 0. Then, we build networks corresponding to p and to q (with outputs a and b, respectively). Finally, we build a cell out, whose code reads and increments r, and which will read a or b depending on whether the reference's contents are less than or equal to k. The demand-driven nature of evaluation means that we never redundantly evaluate p or q's networks — we only ever execute one of them.

The operation loop a_0 p builds a feedback network by explicitly creating a reference to hold an accumulator parameter. It constructs a local reference initialized to a_0, and then constructs a cell ab which reads the input and the local reference to produce a pair of type $A \times B$. This cell is given to p, to construct a network with an output cell ac, yielding pairs of type $A \times C$. Finally, we construct the overall output cell c, which reads ac and updates the local reference with a new value of type A, and returns a value of type C. The use of a local reference (rather than a cell) to store the current state of A is essential, because we need to maintain the acyclicity of the dataflow graph.

With these ideas in mind, we come to the definition of what it means for a dataflow network to realize a stream transducer. This property is quite large, but despite its size is pleasant to work with.

```
1  Realize(i, Φ, o, f) ≜
2  ∀v : stream(A). ∃φ : stream(formula), u : stream(P^fin(ecell)).
3      {∀n : ℕ. closed(φ_n, dom(φ_n) ∪ {i}) ∧
4               dom(φ_n) = dom(φ_{n+1}) ∧
5               ∀ψ. unready(ψ, i) ⊃ unready(ψ ⊗ φ_n, o)}
6       and {Φ = φ_0}
7       and Transduce(i, φ, o, u, f, v)

8  Transduce(ri, φ, o, u, f, v) ≜
9  ∀n : ℕ, φ_in, φ'_in, u_in.
10   ⟨φ_in; read i⟩ ⇓ ⟨φ'_in; v_n⟩ [{i}|u_in]  ∧
11   {dom(φ_in) = dom(φ'_in) ∧
12    closed(φ_in, dom(φ_in)) ∧ closed(φ_in, dom(φ'_in))∧
13    unready(φ_in, i) ∧ i ∈ u_in}
14   implies
15   ⟨φ_in ⊗ φ_n; read o⟩ ⇓ ⟨φ'_in ⊗ φ_{n+1}; (f v)_n⟩ [{o}|u_in ∪ u_n]
16    and {u ⊆ dom(φ_n) ∧ o ∈ u}
17
```

We read $Realize(i, \Phi, o, f)$ as saying "the dataflow network Φ realizes the stream transducer f, when the event loop writes inputs into i and reads outputs from o".

We have highlighted the key pieces of this definition with boxes. On line 2, for each input stream v, we existentially assert the existence of a *stream* of abstract heap formulas ϕ. This stream represents the evolving state of the network over time — because our notification networks contain local state, that state can potentially have a different value at each time step.

Then, in the unboxed formulas, we assert some well-formedness properties. On lines 3 to 5, we assert that (1) the only external cell the network may read is i, (2) that the domain of the network (i.e., the cells whose atomic formulas are in that formula) remains constant over time, and (3) that if the input cell is ever unready, so is the output (i.e, the output genuinely depends on the input). Then, on line 6, we assert that the initial state of the evolution of the network is Φ. (All of these conditions could be relaxed, but in this paper there is no need. It *would* be necessary if we added combinators that dynamically created new transducers as the program ran, since we could then create new cells at each time step.)

Finally, we assert the network implements the stream transducer property on line 7, using the $Transduce$ sub-predicate. On line 9, we first quantify over all times n, and then over ϕ_{in}, ϕ'_{in}, and u_{in}. These extra parameters exist because reading i to get the input, may require evaluating some auxiliary network state.

We read ϕ_{in} as the network state needed to read i, and ϕ'_{in} is that state after i has been read, with u_{in} being the cells in ϕ_{in} updated during that execution. And indeed, on the next line in the boxed formula, we give an abstract triple making exactly this assumption – that reading i will give us the v_n, the n-th element of the stream v, and that doing so requires the state ϕ_{in}, which will become ϕ'_{in} during the execution. (The unboxed formulas are more syntactic conditions we push along.)

The conclusion of this implication over triples, the second boxed formula, says that reading o will give us the $(fv)_n$, the n-th output of the stream transducer, and that in doing so it will also update the input state ϕ_{in} to ϕ'_{in}, in addition to sending the transducer state from ϕ_n to ϕ_{n+1}. With this definition, we can prove the following specifications:

PROPOSITION 6. *(FRP Correctness) We define the* $Relate$ *predicate:*

```
Relate_{A,B}(p, f) ≜
  ∀ψ : formula, i : cell A.
    {G(ψ)} p i {o : cell B. ∃Φ. G(Φ ⊗ ψ) ∧
                                Realize(i, Φ, o, f) valid}
```

Then, the following specifications are provable:

$\forall f : A \to B.\ Relate(\mathsf{lift}\ f, lift\ f)$

$\forall p, f, q, g.\ Relate(p, f)\ \mathsf{and}\ Relate(q, g)$
$\quad \mathsf{implies}\ Relate(\mathsf{seq}\ p\ q, seq\ f\ g)$

$\forall p, f, q, g.\ Relate(p, f)\ \mathsf{and}\ Relate(q, g)$
$\quad \mathsf{implies}\ Relate(\mathsf{par}\ p\ q, par\ f\ g)$

$\forall k, p, f, q, g.\ Relate(p, f)\ \mathsf{and}\ Relate(q, g)$
$\quad \mathsf{implies}\ Relate(\mathsf{switch}\ k\ p\ q, switch\ k\ f\ g)$

$\forall a_0, p, f.\ Relate(p, f)\ \mathsf{implies}\ Relate(\mathsf{loop}\ \ a_0\ p, loop\ a_0\ f)$

These lemmas permit us to reason about our transducer implementation as if it is a pure implementation — for each combinator in the interface, we have a proof that shows the corresponding implementation combinator lifts related arguments to related results. As a result, we can show that, for example, $\mathsf{seq}\ (\mathsf{lift}\ f)\ (\mathsf{lift}\ g)$ and $\mathsf{lift}\ (g \circ f)$ both realize the same pure *lift* $(g \circ f)$.

6. Future Work

We foresee many further applications of the idea of ramifications.

First, there are numerous algorithms — such as unification, the union-find algorithm, and the chaotic iteration constraint propagation algorithm used in dataflow analysis — which rely on using mutation and assignment as a way of globally broadcasting information to the rest of the program state. These algorithms have all been resistant to modular proof, because of the apparent need to know "the rest of the world" in the program invariant. It would be interesting to see if ramifications can help.

Second, we would like to investigate the relationship between ramification operators and methods based on rely-guarantee (Jones 1983). Rely-guarantee imposes a mutual contract between a piece of code and the rest of the world. This is conceptually similar to the idea of a ramification, though we see no obvious direct relationship.

Third, we introduced ramifications as a style of specification useful for verifying a particular library. Might it be useful to make ramification operators part of the basic logical framework? If so, what are their logical properties? $R(u, \phi)$ looks like a family of modal operators on the formula ϕ, but we needed a number of auxiliary interaction lemmas to make them truly useful.

7. Related Work

Versions of separation logic (Reynolds 2002) supporting higher-order languages and quantification over predicates have been proposed by Nanevski et al. (2006) with Hoare Type Theory, and by Parkinson and Bierman (2008). It would be interesting to adapt the proof techniques in this paper to their settings.

Prior work on verifying the observer pattern using separation logic includes work by Krishnaswami et al. (2009, 2007) and work by Parkinson (2007). Similar techniques have also been applied in the setting of regional logic by Banerjee et al. (2008).

In all of these works, the program invariant explicitly tracks the observers listening to a particular subject, and the invariants for each observer. Thus there is an invariant for a cluster of objects (subjects and observers). For a chain of observers only one level deep, this works reasonably well, but it breaks down when there are chains of dependencies — when we have an object which both observes and is observed by yet other objects. Roughly speaking, it is difficult to locally add an observer to a subject, since we then need to touch the invariants of everything the subject itself observes, forcing us to know the transitive closure of the reachability graph of object clusters from the subject.

Even though all of these methods are modular in the sense that subjects and observers can be verified independently, they are non-modular in the sense that clients will have have to track entire dependency graphs when verifying nested uses of the observer pattern. In fact, our present work began when we realized that this style of observer invariant made it difficult to verify the model-view-controller pattern.

Shaner et al. (2007) studied using gray-box model programs to model higher-order method calls (which can be understood as a variant of techniques from refinement calculus) in JML, and Barnett and Naumann (2004) added a friendship system to the Boogie system, with which it is possible to describe some forms of clusters of collaborating objects, including the subject-observer pattern. These works are also non-modular in the above sense, since they require knowing what all of the observers are.

Leino and Schulte (2007) have applied the idea of history invariants (Liskov and Wing 2001) to model observers. The use of monotonic predicates *does* give rise to a modular proof technique, but it sharply restricts the kinds of invariants that can be used, in ways that make it very difficult to model the code-update-based protocol seen in our FRP example.

Acar et al. (2006) have proposed *self-adjusting computation* as a technique for using change propagation to write programs that incrementally recompute answers as the inputs are adjusted. **?** showed how to build a monadic combinator library for self-adjusting computation using techniques strikingly similar to the monadic library described in this paper. This gives us confidence that this is a natural implementation style, and makes us hope that ramifications can help in verifying implementations of this technique.

FRP was proposed by Elliott and Hudak (1997) as a declarative formalism for interactive programming. The API in our paper differs from theirs in two ways. First, our interface is a variation of the *arrowized FRP* interface proposed in Hudak et al. (2002), and secondly, we use a discrete rather than continuous model of time — though we found the idea of using a declarative semantics as a specification for the interface an inspiring one. Our work could also serve as a bridge between the work on purely functional and imperative implementations, such as the work done by McDirmid and Hsieh (2006) on SuperGlue and by Cooper and Krishnamurthi (2006) on the FrTime system.

Acknowledgments

We would like to thank Peter O'Hearn for pointing out the connection of our work with the ramification problem of AI. This work was partially supported by the US NSF grants CCF-0916808 and CCF-0546550, and US DARPA grant HR00110710019.

References

U.A. Acar, G.E. Blelloch, and R. Harper. Adaptive functional programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(6):990–1034, 2006.

Anindya Banerjee, David A. Naumann 2, and Stan Rosenberg. Regional logic for local reasoning about global invariants. In *ECOOP*, pages 387–411, 2008.

Mike Barnett and David A. Naumann. Friends need a little bit more: Maintaining invariants over shared state. In *MPC*, pages 54–64, 2004.

Bodil Biering, Lars Birkedal, and Noah Torp-Smith. BI-hyperdoctrines, higher-order separation logic, and abstraction. *ACM TOPLAS*, 29(5):24, 2007. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/1275497.1275499.

R. Bird and P. Wadler. *An introduction to functional programming*. Prentice Hall International (UK) Ltd. Hertfordshire, UK, UK, 1988.

L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules. In *Proc. of LICS'05*, pages 260–269, 2005.

Lars Birkedal, Kristian Støvring, and Jacob Thamsborg. Realizability semantics of parametric polymorphism, general references, and recursive types. In Luca de Alfaro, editor, *FOSSACS*, volume 5504 of *LNCS*, pages 456–470. Springer, 2009. ISBN 978-3-642-00595-4.

Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In Peter Sestoft, editor, *ESOP*, volume 3924 of *LNCS*, pages 294–308. Springer, 2006. ISBN 3-540-33095-X.

C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of ICFP'97*, pages 263–273. ACM New York, NY, USA, 1997.

J. J. Finger. *Exploiting constraints in design synthesis*. PhD thesis, Stanford University, Stanford, CA, USA, 1987.

Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In Johan Jeuring and Simon L. Peyton Jones, editors, *Advanced Functional Programming*, volume 2638 of *LNCS*, pages 159–187. Springer, 2002. ISBN 3-540-40132-6.

Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(4):596–619, 1983.

N. Krishnaswami, J. Aldrich, and L. Birkedal. Modular verification of the subject-observer pattern via higher-order separation logic. In *Proceedings of FTfJP: Formal Techniques for Java-like Programs*, 2007.

N. Krishnaswami, J. Aldrich, L. Birkedal, K. Svendsen, and A. Buisse. Design patterns in separation logic. In *Proceedings of TLDI'09*, pages 105–116. ACM New York, NY, USA, 2009.

Neelakantan R. Krishnaswami. *Verifying Higher-Order Programming Languages with Higher-Order Separation Logic*. PhD thesis, forthcoming. Carnegie Mellon University, Pittsburgh, PA, USA, 2009.

K. Rustan M. Leino and Wolfram Schulte. Using history invariants to verify observers. In Rocco De Nicola, editor, *ESOP*, volume 4421 of *LNCS*, pages 80–94. Springer, 2007. ISBN 978-3-540-71314-2.

Barbara H. Liskov and Jeannette M. Wing. Behavioural subtyping using invariants and constraints. In *Formal Methods for Distributed Processing: a Survey of Object-Oriented Approaches*, pages 254–280. Cambridge University Press, New York, NY, USA, 2001. ISBN 0-521-77184-6.

John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.

Sean McDirmid and Wilson C. Hsieh. Superglue: Component programming with object-oriented signals. In Dave Thomas, editor, *ECOOP*, volume 4067 of *LNCS*, pages 206–229. Springer, 2006. ISBN 3-540-35726-2.

Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in hoare type theory. In *Proceedings ICFP*, pages 62–73, New York, NY, USA, 2006. ACM. ISBN 1-59593-309-3. doi: http://doi.acm.org/10.1145/1159803.1159812.

M. Parkinson. Class invariants: The end of the road. *Proceedings IWACO*, 2007.

Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In George C. Necula and Philip Wadler, editors, *POPL*, pages 75–86. ACM, 2008. ISBN 978-1-59593-689-9.

Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001. ISSN 0960-1295.

John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS 2002)*, pages 55–74. IEEE Computer Society, 2002. ISBN 0-7695-1483-9.

Steve M. Shaner, Gary T. Leavens, and David A. Naumann. Modular verification of higher-order methods with mandatory calls specified by model programs. In *OOPSLA*, pages 351–368, 2007.

A. Appendix: The cellset Interface

In this section we describe the interface to the cellset type, representing pure collections of existentially quantified cells (terms of type $\exists\alpha : \star.\ \mathsf{cell}\ \alpha$). Specifying this interface is not entirely trivial, because of the way equality works for this type.

Ordinarily, we give a two-place predicate $set(v, elts)$ relating a value v, and the mathematical set of elements $elts$ it represents. However, this approach is not sufficient in our case. In order to manage dependencies, we need to be able to test cells of *different* concrete type for equality, and the natural equality for references only permits testing references of the same type. As a result, we cannot unpack an existentially-quantified cell and compare the elements in its tuple, because we do not know that the two cells are of the same type (and indeed, they might not be).

To deal with this problem, we assign a unique integer identifier to each cell we create, and compare those identifiers to establish equality. Since these identifiers are generated dynamically along with the cells, the precise partial equivalence relation we need to use is determined dynamically as well. So we add an additional index to the set predicate $set(W, v, elts)$. The extra parameter W is the *world*, the set of all the cells allocated so far, whose elements must be equal if and only if their identifier fields match.

The usual set operations are supported: the expression $\mathsf{emptyset}$ represents an empty set of cells; $\mathsf{addset}\ v\ x$ adds the element x to the set v represents; and $\mathsf{removeset}\ v\ x$ removes x from the set v represents. We also have $\mathsf{iterset}\ f\ v$, which iterates over the elements of v's set and applies f to each element in some sequential order. (The specification makes use of two auxiliary predicates: $matches$, which assert that a set and a list have the same elements; and $iterseq$, which constructs a command representing the sequential execution over those elements.)

We have three axioms any implementation must satisfy. First, if a cellset value v represents a set $elts$ in a world W, it must also represent a set in any larger world W' — i.e., allocating new cells cannot disturb already-existing cells. Second, the values in a set are always a subset of the world W. Third, we require that $set(W, v, elts)$ is a pure predicate (i.e., is not heap-dependent). That is, we ask that cellset have a purely functional implementation (for example, as a binary tree). This is not necessary, but does simplify the other invariants in this paper.

$$
\begin{array}{ll}
1 & World = \\
2 & \quad \{D \in \mathcal{P}^{\mathrm{fin}}(ecell) \mid \forall c, d \in D.\ unique(c) = unique(d) \\
3 & \qquad\qquad \iff c = d\} \\
4 & \exists \mathsf{cellset} : \star. \\
5 & \exists set : World \Rightarrow \mathsf{cellset} \Rightarrow \mathcal{P}^{\mathrm{fin}}(ecell) \Rightarrow \mathsf{prop}. \\
6 & \exists \mathsf{emptyset} : cellset. \\
7 & \exists \mathsf{addset} : \mathsf{cellset} \to ecell \to \mathsf{cellset}. \\
8 & \exists \mathsf{removeset} : \mathsf{cellset} \to ecell \to \mathsf{cellset}. \\
9 & \exists \mathsf{iterset} : \mathsf{cellset} \to (ecell \to \bigcirc\mathbf{1}) \to \mathbf{1}. \\
10 & \{\forall W \in World.\ set(W, \mathsf{emptyset}, \emptyset)\}\ \mathsf{and} \\
11 & \{\forall W \in World, v : \mathsf{cellset}, x : ecell, elts \in \mathcal{P}^{\mathrm{fin}}(ecell). \\
12 & \quad set(W, v, elts) \wedge x \in W \supset set(W, \mathsf{addset}\ v\ x, elts \cup \{x\})\}\ \mathsf{and} \\
13 & \{\forall W \in World, v : \mathsf{cellset}, x : ecell, elts \in \mathcal{P}^{\mathrm{fin}}(ecell). \\
14 & \quad set(W, v, elts) \wedge x \in W \supset set(W, \mathsf{removeset}\ v\ x, elts - \{x\})\}\ \mathsf{and} \\
15 & \{\forall W \in World, v : \mathsf{cellset}, elts \in \mathcal{P}^{\mathrm{fin}}(ecell), f : (ecell \to \bigcirc\mathbf{1}). \\
16 & \quad set(W, v, elts) \supset \exists L : \mathsf{seq}\ ecell.\ matches\ elts\ L \wedge \\
17 & \qquad\qquad \mathsf{iterset}\ f\ v = iterseq\ f\ L\}\ \mathsf{and} \\
18 & \{\forall W, W' \in World, v, elts. \\
19 & \quad set(W, v, elts) \wedge W \subseteq W' \supset set(W', v, elts)\}\ \mathsf{and} \\
20 & \{\forall W, W' \in World, v, elts. \\
21 & \quad set(W, v, elts) \supset elts \subseteq W\}\ \mathsf{and} \\
22 & \{\forall W, v, elts.\ Pure(set(W, v, elts))\} \\
23 & matches\ elts\ [] \qquad = elts = \emptyset \\
24 & matches\ elts\ (v :: vs) = v \in elts \wedge matches\ (elts - \{v\})\ vs \\
25 & iterseq\ f\ [] \qquad = [\langle\rangle] \\
26 & iterseq\ f\ (v :: vs) = [\mathsf{letv}\ \langle\rangle = f\ v\ \mathsf{in\ run}\ iterseq\ f\ vs]
\end{array}
$$

A. Appendix: The cellset Interface

In this section we describe the interface to the cellset type, representing pure collections of existentially quantified cells (terms of type $\exists\alpha : \star.\ \mathsf{cell}\ \alpha$). Specifying this interface is not entirely trivial, because of the way equality works for this type.

Ordinarily, we give a two-place predicate $set(v, elts)$ relating a value v, and the mathematical set of elements $elts$ it represents. However, this approach is not sufficient in our case. In order to manage dependencies, we need to be able to test cells of *different* concrete type for equality, and the natural equality for references only permits testing references of the same type. As a result, we cannot unpack an existentially-quantified cell and compare the elements in its tuple, because we do not know that the two cells are of the same type (and indeed, they might not be).

To deal with this problem, we assign a unique integer identifier to each cell we create, and compare those identifiers to establish equality. Since these identifiers are generated dynamically along with the cells, the precise partial equivalence relation we need to use is determined dynamically as well. So we add an additional index to the set predicate $set(W, v, elts)$. The extra parameter W is the *world*, the set of all the cells allocated so far, whose elements must be equal if and only if their identifier fields match.

The usual set operations are supported: the expression $\mathsf{emptyset}$ represents an empty set of cells; $\mathsf{addset}\ v\ x$ adds the element x to the set v represents; and $\mathsf{removeset}\ v\ x$ removes x from the set v represents. We also have $\mathsf{iterset}\ f\ v$, which iterates over the elements of v's set and applies f to each element in some sequential order. (The specification makes use of two auxiliary predicates: $matches$, which assert that a set and a list have the same elements; and $iterseq$, which constructs a command representing the sequential execution over those elements.)

We have three axioms any implementation must satisfy. First, if a cellset value v represents a set $elts$ in a world W, it must also represent a set in any larger world W' — i.e., allocating new cells cannot disturb already-existing cells. Second, the values in a set are always a subset of the world W. Third, we require that $set(W, v, elts)$ is a pure predicate (i.e., is not heap-dependent). That is, we ask that cellset have a purely functional implementation (for example, as a binary tree). This is not necessary, but does simplify the other invariants in this paper.

$$
\begin{array}{ll}
1 & World = \\
2 & \quad \{D \in \mathcal{P}^{\mathrm{fin}}(ecell) \mid \forall c, d \in D.\ unique(c) = unique(d) \\
3 & \qquad\qquad \iff c = d\} \\
4 & \exists \mathsf{cellset} : \star. \\
5 & \exists set : World \Rightarrow \mathsf{cellset} \Rightarrow \mathcal{P}^{\mathrm{fin}}(ecell) \Rightarrow \mathsf{prop}. \\
6 & \exists \mathsf{emptyset} : cellset. \\
7 & \exists \mathsf{addset} : \mathsf{cellset} \to ecell \to \mathsf{cellset}. \\
8 & \exists \mathsf{removeset} : \mathsf{cellset} \to ecell \to \mathsf{cellset}. \\
9 & \exists \mathsf{iterset} : \mathsf{cellset} \to (ecell \to \bigcirc\mathbf{1}) \to \mathbf{1}. \\
10 & \{\forall W \in World.\ set(W, \mathsf{emptyset}, \emptyset)\}\ \mathsf{and} \\
11 & \{\forall W \in World, v : \mathsf{cellset}, x : ecell, elts \in \mathcal{P}^{\mathrm{fin}}(ecell). \\
12 & \quad set(W, v, elts) \wedge x \in W \supset set(W, \mathsf{addset}\ v\ x, elts \cup \{x\})\}\ \mathsf{and} \\
13 & \{\forall W \in World, v : \mathsf{cellset}, x : ecell, elts \in \mathcal{P}^{\mathrm{fin}}(ecell). \\
14 & \quad set(W, v, elts) \wedge x \in W \supset set(W, \mathsf{removeset}\ v\ x, elts - \{x\})\}\ \mathsf{and} \\
15 & \{\forall W \in World, v : \mathsf{cellset}, elts \in \mathcal{P}^{\mathrm{fin}}(ecell), f : (ecell \to \bigcirc\mathbf{1}). \\
16 & \quad set(W, v, elts) \supset \exists L : \mathsf{seq}\ ecell.\ matches\ elts\ L \wedge \\
17 & \qquad\qquad \mathsf{iterset}\ f\ v = iterseq\ f\ L\}\ \mathsf{and} \\
18 & \{\forall W, W' \in World, v, elts. \\
19 & \quad set(W, v, elts) \wedge W \subseteq W' \supset set(W', v, elts)\}\ \mathsf{and} \\
20 & \{\forall W, W' \in World, v, elts. \\
21 & \quad set(W, v, elts) \supset elts \subseteq W\}\ \mathsf{and} \\
22 & \{\forall W, v, elts.\ Pure(set(W, v, elts))\} \\
23 & matches\ elts\ [] \qquad = elts = \emptyset \\
24 & matches\ elts\ (v :: vs) = v \in elts \wedge matches\ (elts - \{v\})\ vs \\
25 & iterseq\ f\ [] \qquad = [\langle\rangle] \\
26 & iterseq\ f\ (v :: vs) = [\mathsf{letv}\ \langle\rangle = f\ v\ \mathsf{in\ run}\ iterseq\ f\ vs]
\end{array}
$$