Dependently Typed Web Client Applications

FRP in Agda in HTML5

Alan Jeffrey

Alcatel-Lucent Bell Labs

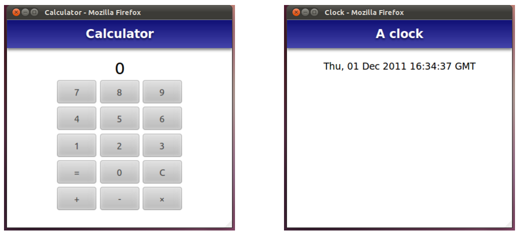
Abstract. In this paper, we describe a compiler back end and library for web client application development in Agda, a dependently typed functional programming language. The compiler back end targets ECMAScript (also known as JavaScript), and so is executable in a browser. The library is an implementation of Functional Reactive Programming (FRP) using a constructive variant of Linear-time Temporal Logic (LTL) as its type system.

1 Introduction

Client-side applications are typically model-view-controller architectures, and often include features such as imperative state, concurrency and continuation-passing. These features can result in code which is difficult to reason about, debug and maintain. In this paper, we propose adapting *Functional Reactive Programming* (*FRP*) [13] to the setting of a pure, dependently typed, functional programming language, Agda [1].

Figure 1 shows some simple applications running in a browser. What is interesting about these applications is that they are written in Agda, and compiled to ECMAScript [7]. We have developed a compiler back end, foreign function interface, and library bindings for FRP, and for HTML5 [15] *Document Object Model* (*DOM*) node and event bindings. The compiler extensions have been released as part of Agda 2.3.0, and the libraries are released under an MIT License [4]. Novel features of the compiler and libraries include:

– *Interoperability with ECMAScript idioms.* The compiler makes use of common ECMAScript idioms, to simplify the use of existing ECMAScript libraries in Agda. For example, the Visitor and Observer patterns [14] are used to implement inductive datatypes and notification.
– *Singleton analysis for type erasure.* We perform a static analysis that conservatively approximates singleton types (which have only one inhabitant at run time). Any term of singleton type is replaced by the singleton value at compile time. In particular, we regard Set as having singleton value null, which allows many type-level computations to be eliminated.
– *View patterns in the FFI.* We support a ECMAScript *Foreign Function Interface* (*FFI*) which, as well as providing bindings for constants and functions, also allows inductive datatypes in Agda to be bound to any ECMAScript type. A variant of view patterns [24] allows pattern-matching to be



Thu, 01 Dec 2011 16:34:37 GMT

Fig. 1. Example Agda programs running in the browser

compiled to any ECMAScript conditional, for example an Agda boolean type can be compiled to ECMAScript native booleans, without any additional support from the Agda compiler.

- *Linear-time Temporal Logic (LTL) types for FRP.* The semantics of FRP is defined in terms of *signals*, which are time-dependent values. In previous work [16], we showed that signals can be typed using time-dependent types, using the combinators of LTL [23], such that any FRP program is a proof of an LTL tautology.
- *Resource reclamation of FRP signals.* The FRP implementation makes use of techniques from *self-adjusting computation* [8], where signals form a dataflow graph, making use of notifications whenever a signal value changes. We are recording the creation time of each signal in its type, and so can maintain time-sensitive invariants which allow resource reclamation of irrelevant signals, even when the garbage collector regards the signal as still live.
- *Inference of DOM node locations.* A difficult problem in GUI libraries for functional languages is the binding of event listeners to GUI components. In an OO language, binding makes use of object identity, which violates referential transparency since components with identical definitions may have different event streams. In a functional language, this could be modeled by a name creation mechanism [22] or nondeterminism [19], but such models are not compatible with Agda's semantics. We provide a novel form of location inference, which supports the creation of DOM event streams from DOM nodes without violating referential transparency.

Agda is used throughout this paper, but we expect the results would apply to other dependently typed languages, such as Coq [3] or Epigram [5].

Thanks to Sebastian Bocq for detailed comments on this paper.

2 Compiling Agda to ECMAScript

We first consider the design of the ECMAScript back end for the Agda compiler, which is included in Agda 2.3. The compiler translates a dependently typed λ-calculus with inductive datatypes and records into an untyped λ-calculus with records. The interesting features of the compiler are its treatment of singleton

```
data List (A : Set) : Set where
  nil : List A
  cons : A → List A → List A

append : ∀ {A} → List A → List A → List A
append nil bs = bs
append (cons a as) bs = cons a (append as bs)
```

Fig. 2. Example program in Agda

```
data $List\,A$ : Set 0 where {
   $nil : List\,A$,
   $cons : \Pi a\ .\ \Pi as\ .\ List\,A$
}
function $append : \Pi A\ .\ \Pi as\ .\ \Pi bs\ .\ List\,A$
   $= \lambda A\ .\ \lambda as$ . case $as$ of {
      $nil \mapsto \lambda bs\ .\ bs$,
      $cons\ a\ as \mapsto \lambda bs\ .\ cons\ a(append\ A\ as\ bs)$
   }
```

Fig. 3. Example program in Agda IL

```
exports = {
   $nil \mapsto \lambda()\ .\ \lambda(v)\ .\ (v.\,nil())$,
   $cons \mapsto \lambda(a, as)\ .\ \lambda(v)\ .\ (v.\,cons(a, as))$,
   $append \mapsto \lambda(A)\ .\ \lambda(as)\ .\ (as(\{$
      $nil \mapsto \lambda()\ .\ \lambda(bs)\ .\ bs$,
      $cons \mapsto \lambda(a, as)\ .\ \lambda(bs)\ .\ (\text{exports}.\ cons(a, \text{exports}.\ append(A)(as)(bs)))$
   $\})$
}
```

Fig. 4. Example program in ECMAScript IL

```
define(["exports",function(exports) {
  exports.nil = function() { return function(v) { return v.nil(); }; };
  exports.cons = function(a,as) { return function(v) { return v.cons(a,as); }; };
  exports.append = function(A) { return function(as) { return as({
    nil: function() { return function(bs) { return bs; }; },
    cons: function(a,as) { return function(bs) {
      return exports.cons(a,exports.append(A)(as)(bs));
    }; }
  }); }; };
});
```

Fig. 5. Example program in ECMAScript

types (including type erasure, since Set is treated as a singleton type) and the translation of datatypes to a use of the visitor pattern.

In Figures 2–5, we show how a simple datatype and recursive function (append over lists) is translated first into an *Agda Intermediate Language* (*IL*), then an *ECMAScript IL*, and finally into ECMAScript:

– The translation from Agda (Figure 2) to Agda IL (Figure 3) is not novel, and handles issues such as making implicit arguments explicit and η-normalizing function applications. In this paper, we give a presentation using case statements in the IL rather than pattern matching. We compile pattern matches to case using decision trees (credited by Cardelli [11] to Kahn and MacQueen in the HOPE compiler [10]).
– The translation from the Agda IL (Figure 3) to the ECMAScript IL (Figure 4) is the interesting one, and is discussed in more detail below. Note that in this translation, case statements over inductive datatypes have been replaced by uses of the visitor pattern, and that top-level declarations in an Agda module have been replaced by fields in an ECMAScript record exports.
– The translation from the ECMAScript IL (Figure 4) to ECMAScript (Figure 5) is routine. We make use of the *Asynchronous Module Definition* (*AMD*) [2] module system for ECMAScript, which supports a special object exports. The translation of an Agda module is an ECMAScript module which assigns to the appropriate exports field.

Figure 6 shows a simplified grammar for the Agda IL, which is a λ-calculus with records, inductive datatypes, Π types, stratified Set types and postulates (uninterpreted constants). The main differences between this presentation of the IL and the actual implementation are modules, namespacing, type information, and the use of case expressions rather than pattern matching functions.

Figure 7 shows a simplified grammar for the ECMAScript IL, which is an untyped λ-calculus with records. The main differences between this presentation of the IL and the actual implementation are namespacing, conditionals and infix and prefix operators. Note that many features of ECMAScript are missing from the ECMAScript IL, such as mutable state, prototypes and constructors. The ECMAScript IL allows importing arbitrary AMD modules, so these features can still be used, as long as they are in an imported module.

We define β-reduction as per usual in a λ-calculus with records. The only point of interest in the definition is the use of undef in ECMAScript's semantics. For example, we define capture-avoiding substitution $M[\vec{N}/\vec{x}]$ in the usual way whenever $|\vec{N}| = |\vec{x}|$, then generalize to arbitrary $\vec{N}$ and $\vec{x}$ by substituting undef if necessary:

$$\begin{aligned} M[(\vec{N},\vec{L})/\vec{x}] &= M[\vec{N}/\vec{x}] && \text{when } |\vec{N}| = |\vec{x}| \\ M[\vec{N}/(\vec{x},\vec{y})] &= M[\vec{N}/\vec{x},\mathsf{undef}/\vec{y}] && \text{when } |\vec{N}| = |\vec{x}| \end{aligned}$$

from which we define β-reduction of functions:

$$(\lambda(\vec{x})\,.\,M)(\vec{N}) \to M[\vec{N}/\vec{x}]$$

$$\begin{aligned}
A, B, C &::= x\,\vec{A} \mid \lambda x \,.\, A \mid \{\vec{\ell} \mapsto \vec{B}\} \mid A.\ell \mid g\,\vec{A} \mid k \mid \\
&\mid \mathit{\Pi} x \,.\, A \mid \mathsf{Set}\ A \mid c\,\vec{A} \mid \mathsf{case}\,A\,\mathsf{of}\,\{\vec{P} \mapsto \vec{B}\} \\
P, Q &::= c\,\vec{x} \\
D, E &::= \mathsf{function}\,g : A = B \mid \mathsf{data}\,g\,\vec{x} : B\ \mathsf{where}\ \{\vec{c} : \vec{C}\} \mid \\
&\quad \mathsf{record}\,g\,\vec{x} : B\ \mathsf{where}\ \{\vec{\ell} : \vec{C}\} \mid \mathsf{postulate}\,g : A
\end{aligned}$$

Fig. 6. Agda IL

$$L, M, N ::= x \mid \lambda(\vec{x}) \,.\, M \mid M(\vec{N}) \mid \{\, \vec{\ell} \mapsto \vec{M} \,\} \mid M.\ell \mid k$$

Fig. 7. ECMAScript IL

Similarly, field access of an object returns undef for missing fields:

$$\{\, \vec{\ell} \mapsto \vec{M} \,\}.\ell \to \begin{cases} M_i & \text{if } \ell = \ell_i \\ \mathsf{undef} & \text{otherwise} \end{cases}$$

In Figures 8–9 we show how Agda IL is translated into ECMAScript IL. Most of the translation is direct, but there are two points of interest: a static approximation of *singleton types*, and the *visitor pattern* [14] for inductive datatypes.

For singleton types, we include a judgement "A has singleton B", meaning that any closed instance of type A must be equal to B. For example:

- $\top$ (a record type with no fields) has singleton $\{\,\}$.
- $\bot$ (an inductive type with no constructors) has no closed instances, so we can declare that $\bot$ has singleton undef. Since $\neg A$ is defined to be $A \to \bot$, it has singleton $\lambda x.\mathsf{undef}$, and so we can eliminate many instances of negations.
- Since we are using a type-erasing translation, Set A (the type of types at universe level A) has singleton null. This eliminates many instances of runtime type computation.

The visitor pattern uses double callbacks to emulate case statements (in [9], this form of visitor is called an *external* visitor, in contrast to an *internal* visitor which emulates a recursion scheme). For example, if as is a list, then:

```
as({ nil: f, cons: g })
```

will call back f() if as is an empty list, and g(b,bs) if as has head b and tail bs. The translation of case statements into visitors is direct.

Recall that recursive declarations are translated to imperative updates to the mutable exports variable. For mutually recursive declarations under a λ (such as the traditional *even* and *odd* functions) this presents no problem, but for top-level recursive declarations, we have to ensure that exports are defined before

$$\begin{aligned}
[\![x\,\vec{A}]\!] &= x([\![A_1]\!])\cdots([\![A_n]\!])\\
[\![\lambda x\,.\,A]\!] &= \lambda x\,.\,[\![A]\!]\\
[\![\{\vec{\ell}\mapsto\vec{A}\}]\!] &= \{\,\vec{\ell}\mapsto[\![\vec{A}]\!]\,\}\\
[\![A.\ell]\!] &= [\![A]\!].\ell\\
[\![g\,\vec{A}]\!] &= \begin{cases} C & \text{if } g\,\vec{A} : B \text{ and } B \text{ has singleton } C\\ \mathsf{exports}.g\,[\![\vec{A}]\!] & \text{otherwise}\end{cases}\\
[\![k]\!] &= k\\
[\![\mathit{\Pi} x\,.\,A]\!] &= \mathsf{null}\\
[\![\mathsf{Set}\ A]\!] &= \mathsf{null}\\
[\![c\,\vec{A}]\!] &= c([\![\vec{A}]\!])\\
[\![\mathsf{case}\,A\,\mathsf{of}\,\{\vec{P}\mapsto\vec{B}\}]\!] &= A(\{\,[\![\vec{P}\mapsto\vec{B}]\!]\,\})\\
[\![c\,\vec{x}\mapsto B]\!] &= c\mapsto\lambda(\vec{x})\,.\,[\![B]\!]
\end{aligned}$$

$$\frac{\mathsf{data}\,g\,\vec{x} : \mathsf{Set}\ A\,\mathsf{where}\,\{\,\}}{g\,\vec{A}\text{ has singleton }\mathsf{undef}}\qquad\frac{\mathsf{data}\,g\,\vec{x} : \mathsf{Set}\ A\,\mathsf{where}\,\{c : g\,\vec{x}\}}{g\,\vec{A}\text{ has singleton }c}\qquad\frac{\mathsf{record}\,g\,\vec{x} : \mathsf{Set}\ A\,\mathsf{where}\,\{\,\}}{g\,\vec{A}\text{ has singleton }\{\,\}}$$

$$\frac{A\text{ has singleton }B}{\mathit{\Pi} x\,.\,A\text{ has singleton }\lambda x\,.\,B}\qquad\frac{}{\mathsf{Set}\ A\text{ has singleton }\mathsf{null}}$$

Fig. 8. Translation of Agda expressions to ECMAScript

their use. Consider the Agda IL declaration:

$$\mathsf{function}\,x : \mathbb{N} = y + 1 \quad \mathsf{function}\,y : \mathbb{N} = 3$$

and then translated into ECMAScript IL it is:

$$\{\,x\mapsto\mathsf{exports}.y+1; y\mapsto 3\,\}$$

Unfortunately, translated directly into ECMAScript, this would be:

```
define(["exports"],function(exports) {exports.x = exports.y + 1;exports.y = 3;});
```

which generates a load-time error, since exports.y is undefined at the point of its use. To avoid this, we inline any occurrences of exports which occur outside of an enclosing λ. In this example, inlining exports produces:

$$\{\,x\mapsto\{\,x\mapsto\mathsf{exports}.y+1; y\mapsto 3\,\}.y+1; y\mapsto 3\,\}$$

which β-reduces to:

$$\{\,x\mapsto 3+1; y\mapsto 3\,\}$$

and translates into ECMAScript as:

```
define(["exports"],function(exports) {exports.x = 3 + 1;exports.y = 3;});
```

$$
\begin{aligned}
[\![\mathsf{function}\, g : A = B]\!] &= g \mapsto \begin{cases} C & \text{if } A \text{ has singleton } C \\ [\![A]\!] & \text{otherwise} \end{cases} \\
[\![\mathsf{data}\, g\, \vec{x} : A\ \mathsf{where}\ \{\vec{c} : \vec{B}\}]\!] &= [\![\vec{c} : \vec{B}]\!] \\
[\![\mathsf{record}\, g(\vec{x} : \vec{A}) : B\ \mathsf{where}\ \{\vec{\ell} : \vec{C}\}]\!] &= \epsilon \\
[\![\mathsf{postulate}\, g : A]\!] &= g \mapsto \begin{cases} B & \text{if } A \text{ has singleton } B \\ \mathsf{undef} & \text{otherwise} \end{cases} \\
[\![c : \Pi\vec{x} \,.\, g\,\vec{A}]\!] &= c \mapsto \lambda(\vec{x}) \,.\, \lambda(v) \,.\, v.c(\vec{x})
\end{aligned}
$$

Fig. 9. Translation of Agda declarations to ECMAScript

As this example shows, we use a nïave strategy of always inlining top-level occurrences of exports. Since Agda is total, this process must terminate (unless Agda's termination checker is disabled, in which case the compiler is not guaranteed to terminate) but may result in an exponential blowup in program resource usage. We leave a more sophisticated treatment of inlining for future work.

3 Foreign function interface

For Agda to be useful for writing web applications, it must interact with native ECMAScript APIs, notably those defined by HTML5. The translation of Agda into ECMAScript is designed to make this as simple as possible (for example, translating functions to functions, and records to records) but there is still a need for a *Foreign Function Interface* (*FFI*) to provide bindings for native types. In Agda, FFIs are defined via pragmas, for example to bind Agda identifier g to ECMAScript term M:

COMPILED_JS g M

In the case of functions, constructors or postulates, the semantics of FFI code is direct: the ECMAScript is inlined (and β-reduced) whenever the identifier is used. For example if we define:

```
data ℕ : Set where      _+_ : ℕ → ℕ → ℕ           _*_ : ℕ → ℕ → ℕ
  zero : ℕ              zero  + y = y             zero  * y = zero
  suc : ℕ → ℕ           suc x + y = suc (x + y)   suc x * y = y + (x * y)
```

then the following pragma declarations bind zero, suc, + and * to their native counterparts:

```
COMPILED_JS zero 0
COMPILED_JS suc function(x) { return x+1; }
COMPILED_JS _+_ function(x) { return function(y) { return x+y; }; }
COMPILED_JS _*_ function(x) { return function(y) { return x*y; }; }
```

By itself, however, this is not sufficient, as user code may include recursive functions over naturals, such as the ever-popular factorial:

```
fact : ℕ → ℕ
fact zero = suc zero
fact (suc x) = suc x * fact x
```

To support this, we allow FFI bindings from datatypes to the *acceptor function* for that datatype, similar to *view patterns* [24]. The acceptor is a function $f(x, v)$ which takes as parameters a value x, and a visitor v, and calls the appropriate visitor method. For example if we declare:

```
COMPILED_JS ℕ function (x,v) {
  if (x < 1) { return v.zero(); } else { return v.suc(x-1); }
}
```

then the generated ECMAScript for the factorial function is:

```
exports.fact = function (x) {
  if (x < 1) { return 0+1; } else { return ((x-1)+1) * exports.fact(x-1); }
}
```

4 Functional Reactive Programming

The style of programming used in web applications such as those in Figure 1 is *Functional Reactive Programming* (*FRP*) [13]. The semantics of FRP is defined in terms of *signals*, which are thought of as time-dependent values. For example, the clock application is:

$$\mathsf{main} = \mathsf{text}(\mathsf{map}\,\mathsf{toUTCString}(\mathsf{every}(1\,\mathsf{sec})))$$

where:

– $\mathsf{every}(1\,\mathsf{sec})$ is a signal of Time, which updates every second,
– $\mathsf{map}\,f(\sigma)$ applies a function $f : A \to B$ to a signal σ of A to get a signal of B, here $\mathsf{toUTCString} : \mathsf{Time} \to \mathsf{String}$, and
– $\mathsf{text}(\sigma)$ converts a signal σ of String to a signal of DOM nodes.

The types of these combinators are (ignoring some issues about the type for DOM nodes, which we return to in Section 6):

$$\begin{aligned}
\mathsf{every} &: \mathsf{Delay} \to [\![\Box\langle\mathsf{Time}\rangle]\!] \\
\mathsf{map} &: [\![A \Rightarrow B]\!] \to [\![\Box A \Rightarrow \Box B]\!] \\
\mathsf{text} &: [\![\Box\langle\mathsf{String}\rangle \Rightarrow \Box\mathsf{DOM}]\!]
\end{aligned}$$

which gives the type of main as $[\![\Box\mathsf{DOM}]\!]$, that is a signal of DOM nodes, suitable for rendering in a browser.

These types are based on *Linear-time Temporal Logic* (*LTL*) [23]. In previous work [16] we showed that FRP programs in a dependently typed programming language can be given types in a constructive variant of LTL, such that any well-typed FRP program is a proof of an LTL tautology. The correspondence between FRP programs and LTL proofs was discovered independently by Jeltsch [18]. Since LTL propositions are parameterized over time, we consider types parameterized over time, that is *reactive types*:

$$\mathsf{RSet} = \mathsf{Time} \to \mathsf{Set}$$

where Time is a totally ordered set (implemented using ECMAScript's time model, which is an integer number of milliseconds since 1 Jan 1970). Some combinators on reactive types are:

$$\begin{array}{rlr}
\langle\cdot\rangle & : \mathsf{Set} \to \mathsf{RSet} & \langle A\rangle = \lambda t \,.\, A \\
(\cdot \Rightarrow \cdot) & : \mathsf{RSet} \to \mathsf{RSet} \to \mathsf{RSet} & A \Rightarrow B = \lambda t \,.\, A(t) \to B(t) \\
[\![\cdot]\!] & : \mathsf{RSet} \to \mathsf{Set} & [\![A]\!] = \forall\{t\} \,.\, A(t) \\
\Box & : \mathsf{RSet} \to \mathsf{RSet} & \Box A = \,?
\end{array}$$

These combinators are:

- $\langle A\rangle$ is a constant reactive type; viewed as a temporal proposition $\langle A\rangle$ is true at time t when A is true.
- $A \Rightarrow B$ is the pointwise function space between A and B; viewed as a temporal proposition, $A \Rightarrow B$ is true at time t when A being true at time t implies that B is true at time t.
- $[\![A]\!]$ embeds RSet back into Set; viewed as a proposition $[\![A]\!]$ is true whenever A is a tautology, that is A is true at all times t.
- $\Box A$ is the type of signals of A; viewed as a temporal proposition, $\Box A$ is true at time t whenever A is true at any time $u \geq t$.

The FRP combinators can be viewed as a proof system for LTL, for example one of the axioms of S4 modal logic is given by:

$$\mathsf{map} : [\![A \Rightarrow B]\!] \to [\![\Box A \Rightarrow \Box B]\!]$$

Note that we do *not* give a definition for $\Box A$ (the library defines it as a postulate). It is isomorphic to LTL's "global" modality:

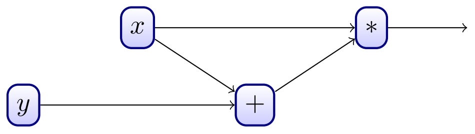
$$\Box A \approx \lambda t \,.\, \forall u \,.\, \mathsf{True}(t \leq u) \to A\,u$$

where:

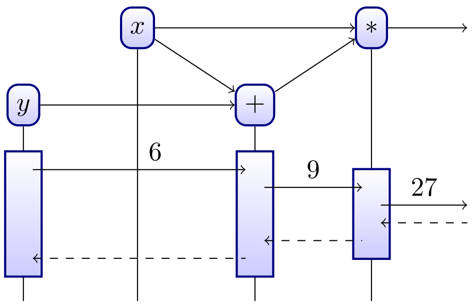
$$\mathsf{True}(\cdot) \;:\; \mathsf{Bool} \to \mathsf{Set}$$
$$\mathsf{True}(b) = \begin{cases} 1 \text{ if } b = \mathsf{true} \\ 0 \text{ otherwise} \end{cases}$$

The implementation of $\Box A$ in ECMAScript is not given functionally. Instead it is given as a dataflow graph, where the nodes implement the observer pattern [14]. The implementation is based on *self-adjusting computation* [8, 20], and is similar to FrTime [12], Flapjax [21] and Froc [6].

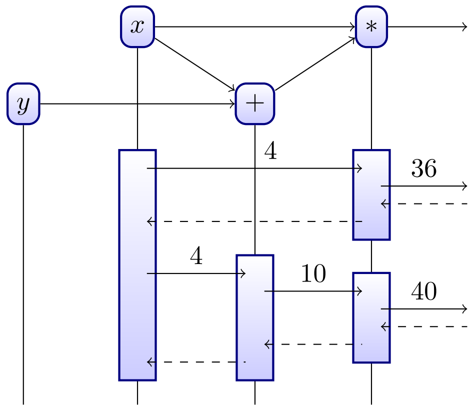
Consider the dataflow graph for the expression $x * (y + x)$:



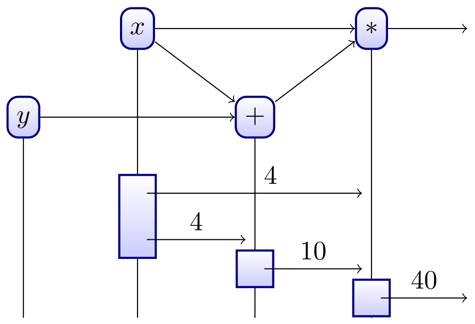
This is implemented as an object graph, where every node implements the Observer pattern, and memoizes its current value. When an update takes place (for example, an external event arrives) the nodes send notifications to their obervers, requesting that they update themselves, and recursively inform their observers if necessary. Note that nodes only notify their observers when their values change, so unchanged nodes are not involved in any updates. For example, if x's value is 3, and y's value is updated to 6 then the following notifications are sent:



Unfortunately, a simple application of the observer pattern results in *glitches*. These are transient erroneous values, due to nodes receiving multiple notifications. For example, if x's value is updated to 4 then the following notifications could be sent:



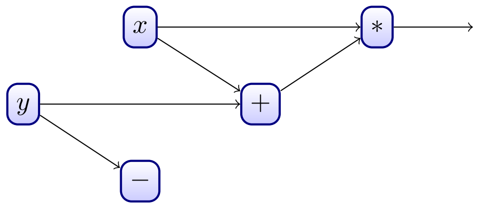
In this example, the $*$ node has been notified twice, and as result has sent two notifications, the first of which does not match the FRP semantics. To avoid such glitches, we adopt the same strategy as [12, 21], and *rank* nodes, such that every node has smaller rank than all of its observers[1]. Notifications are now *asynchronous* rather than synchronous, and are executed in rank order. For example, the above glitchy behaviour is replaced by:



For efficiency, we use synchronous notification for nodes with fan-in one, and reserve asynchronous notification for nodes with fan-in of two or more. The implementation of flow graphs makes use of a task scheduler, which handles asynchronous notification, and ensures that notifications are processed in rank order. The scheduler also supports delayed notification (using the HTML5 time-out mechanism) and rank updates (a node may *switch* its observed neighbours, causing its rank to change, which must be propagated upwards).

5 Garbage collecting FRP

Nodes in the flow graph of an FRP program maintain a set of observer nodes (which should be notified on state updates) and observed nodes (whose memoized values can be queried when a notification is processed). This presents a challenge to a garbage collector, since bidirectional links may keep nodes alive unnecessarily. For example, consider the graph:



Here there is a node $-$ with no observers, which should be reclaimed. Such unobserved nodes can arise dynamically due to switches, which reconfigure the

[1] Acar [8] uses post-order traversal order rather than height order, because his target languages allow for exceptions and other error behaviours, and so (for example) conditionals must be evaluated before branches in an if-expression. Agda is total, and so we can use a simpler ordering strategy, at the possible cost of wasted effort.

node graph. To reclaim unobserved nodes, some FRP implementations [12] make use of *weak pointers* for observers, which would allow garbage collection in this case. Unfortunately, ECMAScript does not support weak pointers.

An alternative is to have the FRP library handle node reclamation. Nodes have addObserver and removeObserver methods: when a node has its last observer removed, it calls removeObserver on each of its observed neighbours to remove itself. Essentially, this is a reference counting garbage collector (cyclic flow graphs are handled by an explicit fixed point function, which does not increase the reference count, so cycles can be collected). These functions are only visible in ECMAScript: they have mutable semantics, so must be kept hidden from Agda.

Unfortunately, this is not always safe, since a node might be added back into the graph after it has been reclaimed:

```
// node1 starts out with just observer node2
node1.removeObserver(node2);
// at this point node1 reclaims its resources
node1.addObserver(node3);
```

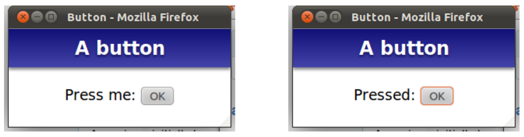
Without some additional guarantees, node1 would reclaim its resources, only later to be added back into the flow graph in an unsafe state. To avoid this, we maintain two invariants:

1. the state of a node is only ever queried by its observers, and
2. a node only ever has observers added during the time slice that it is created.

In the presence of these invariants, we have a safe variant of removeObserver: when a node has its last observer removed *and we have finished processing the time slice that created the node*, it calls removeObserver on each of its observed neighbours to remove itself. In ECMAScript, there is no way to statically enforce the invariants, but in Agda we can do this because the LTL type for a signal $\Box A(t)$ carries a time parameter t which records its start time. The API for signals only allows signals to be built at their start time (for example map f converts a signal of type $\Box A(t)$ to a signal of type $\Box B(t)$, that is the start time is preserved). This technique for tracking creation times is similar to Jeltsch's *era* parameters [17]. Since Agda is a dependent language, we can embed start times directly in types, rather than having to use phantom types for this purpose.

6 Bindings for DOM nodes and events

In Figure 1 we showed a calculator application, built in Agda. A prototypical example of a GUI is a single button:



The source for this program is quite simple:

$$\begin{aligned}&\mathsf{main} = \mathsf{lab} \mathbin{+\!\!+} \mathsf{but}\ \mathsf{where}\\ &\quad \mathsf{but} = \mathsf{element}\,\texttt{"button"}(\mathsf{text}(\mathsf{const}\,\texttt{"OK"}))\\ &\quad \mathsf{clk} = \mathsf{listen}\,\mathsf{click}\,\mathsf{but}\\ &\quad \mathsf{lab} = \mathsf{text}(\mathsf{hold}\,\texttt{"Press me: "}\,(\mathsf{tag}\,\texttt{"Pressed: "}\,\mathsf{clk}))\end{aligned}$$

This declares a button but, and then some text lab whose value depends on the stream clk of click events coming from but. The boilerplate $\mathsf{hold}\,x\,(\mathsf{tag}\,y\,\sigma)$ is a behaviour which starts as value x, and switches to y after the first event from σ.

The types of the functions used here are (somewhat simplified):

$$\begin{aligned} * &: \mathsf{RSet} \to \mathsf{RSet}\\ \mathsf{Mouse} &: \mathsf{RSet}\\ \mathsf{EventType} &: \mathsf{RSet} \to \mathsf{Set}\\ \mathsf{click} &: \mathsf{EventType}\,\mathsf{Mouse}\\ \mathsf{listen} &: \forall\{A\} \to \mathsf{EventType}\,A \to [\![\Box\mathsf{DOM} \Rightarrow *A]\!]\\ \mathsf{const} &: \forall\{A\} \to [\![A]\!] \to [\![\Box A]\!]\\ \mathsf{tag} &: \forall\{A\,B\} \to [\![B]\!] \to [\![*A \Rightarrow *B]\!]\\ \mathsf{hold} &: \forall\{A\} \to [\![\langle A\rangle \Rightarrow *\langle A\rangle \Rightarrow \Box\langle A\rangle]\!]\\ \mathsf{element} &: \mathsf{String} \to [\![\Box\mathsf{DOM} \Rightarrow \Box\mathsf{DOM}]\!]\\ (\cdot \mathbin{+\!\!+} \cdot) &: [\![\Box\mathsf{DOM} \Rightarrow \Box\mathsf{DOM} \Rightarrow \Box\mathsf{DOM}]\!]\end{aligned}$$

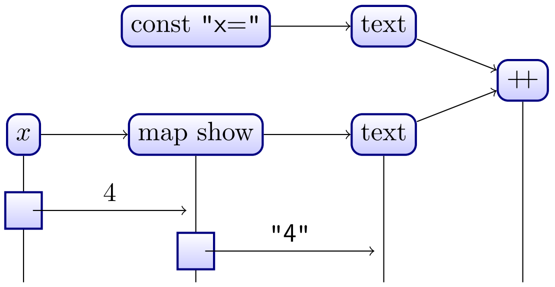
Here:

- $*A$ is the reactive type of event streams, where any event at time t has type $A(t)$. It is implemented in a similar fashion to $\Box A$.
- Mouse is the reactive type of mouse events.
- EventType A is the type of codes for events of type A, for example click is a code for events of type Mouse.
- $\mathsf{listen}\,c\,\sigma$ is an event stream which listens for events with code c coming from DOM nodes σ. For example, listen click but is the stream of click events coming from the button but.
- $\mathsf{const}\,x$ is a constant signal that always returns x.
- $\mathsf{tag}\,x\,\sigma$ is a stream of x events which fires whenever σ fires.
- $\mathsf{hold}\,x\,\sigma$ converts an event stream to a signal, by returning the most recent value from σ (or x if there is none).
- $\mathsf{element}\,a\,\sigma$ constructs a DOM node with tag a and children σ.
- $\sigma \mathbin{+\!\!+} \tau$ concatenates the DOM nodes from σ with those from τ.

The implementation of these functions is fairly straightforward ECMAScript programming using the HTML5 API, for example $\mathsf{Mouse}(t)$ is inhabited by mouse events processed at time t, and $\mathsf{listen}\,\mathsf{click}\,\sigma$ registers a click event handler to each DOM node generated by σ (and deregisters them when the event stream has no observers). DOM nodes are sinks for notifications, for example in the program:

$\mathsf{text\ (const\ "x=")} \mathbin{+\!\!+} \mathsf{text\ (map\ show\ x)}$

if x's value is updated to 4 then the following notifications are sent; note that the text node does *not* update its parent:



The library behaves as expected if a user declares multiple buttons, for example:

$$
\begin{aligned}
&\mathsf{main} = \mathsf{lab} \mathbin{+\!\!+} \mathsf{but}_1 \mathbin{+\!\!+} \mathsf{but}_2\ \mathsf{where}\\
&\quad \mathsf{but}_1 = \mathsf{element}\,\texttt{"button"}(\mathsf{text}(\mathsf{const}\,\texttt{"OK"}))\\
&\quad \mathsf{but}_2 = \mathsf{element}\,\texttt{"button"}(\mathsf{text}(\mathsf{const}\,\texttt{"OK"}))\\
&\quad \mathsf{clk} = \mathsf{listen\,click\,but}_1\\
&\quad \mathsf{lab} = \mathsf{text}(\mathsf{hold}\,\texttt{"Press me: "}\,(\mathsf{tag}\,\texttt{"Pressed: "}\,\mathsf{clk}))
\end{aligned}
$$

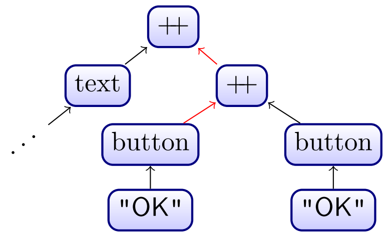
only changes the text when the first button is pressed, not the second, since it only listens to but_1 and not but_2. On the surface, this appears to violate Agda's semantics, which includes β-equivalence, and hence referential transparency. It appears that but_1 and but_2 have the same definition, but yet the behaviour depends on which button we listen to.

Krishnaswami and Benton [19] resolve this by giving event streams a nondeterministic semantics, which the implementation is given freedom to resolve in any way it likes. In practice, the implementation uses node identity to resolve nondeterminism. Since the semantics is nondeterministic, it is no longer defined in a cartesian closed category of sets and functions, but instead in the monoidal closed category of sets and relations. Krishnaswami and Benton provide a DSL with a linear type system for writing such nondeterministic programs. In our GUI library, we are using Agda's native function space to express reactive programs, so we cannot use a nondeterministic semantics.

In fact, the above example does not violate referential transparency, and instead is using implicit arguments to name components. Above, we noted that we had simplified the presentation of the types for DOM nodes. In fact, we do not have DOM : RSet, instead we have:

$$\mathsf{DOM} : \mathsf{DOW} \to \mathsf{RSet}$$

where DOW is a type of *Document Object Worlds* (or "upside-down DOMs"). A value of type DOW records *where* in a DOM tree a node lives, for example in the DOM flow graph:



the route from the the first button node to the root node is left, then right, which we write as $\mathsf{left}(\mathsf{right}(\ell))$ where ℓ is the location of the root node. DOWs are postulated as:

$$
\begin{aligned}
\mathsf{DOW} &: \mathsf{Set} \\
\mathsf{left}, \mathsf{right} &: \mathsf{DOW} \to \mathsf{DOW} \\
\mathsf{child} &: \mathsf{String} \to \mathsf{DOW} \to \mathsf{DOW}
\end{aligned}
$$

Under the hood, a DOW is implemented as a container of DOM nodes, with pointers to all of the DOM nodes at that location (typically there is just one, but since Agda does not support linear types there is no way to enforce that convention).

We can now reveal the "real" types for the DOM-manipulating functions:

$$
\begin{aligned}
\mathsf{text} &: \forall\{\ell\} \to [\![\Box\langle\mathsf{String}\rangle \Rightarrow \Box(\mathsf{DOM}\,\ell)]\!] \\
\mathsf{element} &: \forall a\,\{\ell\} \to [\![\Box(\mathsf{DOM}(\mathsf{child}\ a\ \ell)) \Rightarrow \Box(\mathsf{DOM}\,\ell)]\!] \\
(\cdot +\!\!+ \cdot) &: \forall\{\ell\} \to [\![\Box(\mathsf{DOM}(\mathsf{left}\ \ell)) \Rightarrow \Box(\mathsf{DOM}(\mathsf{right}\ \ell)) \Rightarrow \Box(\mathsf{DOM}\ \ell)]\!] \\
\mathsf{listen} &: \forall\{A\ \ell\} \to \mathsf{EventType}\,A \to [\![\Box\mathsf{DOM}\ \ell \Rightarrow *A]\!]
\end{aligned}
$$

For example, we can make the inferred types explicit in our problematic example:

$$
\begin{aligned}
&\mathsf{main} : \forall\{\ell\} \to [\![\Box(\mathsf{DOM}\ \ell)]\!] \\
&\mathsf{main}\{\ell\}\{t\} = \mathsf{lab} +\!\!+ \mathsf{but}_1 +\!\!+ \mathsf{but}_2\ \mathsf{where} \\
&\quad \mathsf{but}_1 : \Box(\mathsf{DOM}(\mathsf{left}(\mathsf{right}(\ell))))\ t \\
&\quad \mathsf{but}_1 = \mathsf{element}\,\texttt{"button"}(\mathsf{text}(\mathsf{const}\,\texttt{"OK"})) \\
&\quad \mathsf{but}_2 : \Box(\mathsf{DOM}(\mathsf{right}(\mathsf{right}(\ell))))\ t \\
&\quad \mathsf{but}_2 = \mathsf{element}\,\texttt{"button"}(\mathsf{text}(\mathsf{const}\,\texttt{"OK"})) \\
&\quad \mathsf{clk} : *\mathsf{Mouse}\ t \\
&\quad \mathsf{clk} = \mathsf{listen}\{\mathsf{Mouse}\}\{\mathsf{left}(\mathsf{right}(\ell))\}\mathsf{click}\,\mathsf{but}_1 \\
&\quad \mathsf{lab} : \Box(\mathsf{DOM}(\mathsf{left}(\ell)))\ t \\
&\quad \mathsf{lab} = \mathsf{text}(\mathsf{hold}\,\texttt{"Press me: "}\ (\mathsf{tag}\,\texttt{"Pressed: "}\ \mathsf{clk}))
\end{aligned}
$$

With the optional arguments in place, we can see how referential transparency is being maintained: the optional argument to listen is $\mathsf{left}(\mathsf{right}(\ell))$, which is why the value of lab depends on but_1 being pressed but not but_2. Agda's ability to infer expressions as well as types is being used to provide a referentially transparent semantics to a program which looks like it depends on object identity.

References

1. The Agda wiki. http://wiki.portal.chalmers.se/agda/.
2. Asynchronous module definition API. https://github.com/amdjs/.
3. The Coq proof assistant. http://coq.inria.fr/.
4. ECMAScript back end for functional reactive programming in Agda. https://github.com/agda/agda-frp-js.
5. The Epigram 2 programming language. http://www.e-pig.org/darcs/Pig09/web/.
6. Froc: Functional reactive programming in O'Caml. https://jaked.github.com/froc/.
7. ECMAScript language specification. ECMA Standard 262, 5.1 Edition, 2011.
8. U. A. Acar. *Self-Adjusting Computation*. PhD thesis, Carnegie Mellon Univ., 2005.
9. P. Buchlovsky and H. Thielecke. A type-theoretic reconstruction of the visitor pattern. In *Proc. Mathematical Foundations of Programming Semantics*, pages 309–329, 2006.
10. Rod M. Burstall, David B. MacQueen, and Donald Sannella. HOPE: An experimental applicative language. In *Proc. LISP Conf.*, pages 136–143, 1980.
11. Luca Cardelli. Compiling a functional language. In *Proc. ACM Symp. LISP and Functional Programming*, pages 208–217, 1984.
12. Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *Proc. European Symp. Programming*, pages 294–308, 2006.
13. C. Elliott and P. Hudak. Functional reactive animation. In *Proc. Int. Conf. Functional Programming*, pages 263–273, 1997.
14. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
15. Ian Hickson et al. HTML5: A vocabulary and associated APIs for HTML and XHTML. W3C Working Draft, 2011. http://www.w3.org/TR/html5/.
16. A. S. A. Jeffrey. LTL types FRP: Linear-time temporal logic propositions as types, proofs as functional reactive programs. In *Proc. ACM Workshop Programming Languages meets Program Verification*, 2012.
17. W. Jeltsch. Signals, not generators! In *Proc. Symp. Trends in Functional Programming*, pages 283–297, 2009.
18. W. Jeltsch. The Curry-Howard correspondence between temporal logic and functional reactive programming. http://www.cs.ut.ee/˜varmo/tday-nelijarve/jeltsch-slides.pdf, 2011.
19. N. Krishnaswami and N. Benton. A semantic model for graphical user interfaces. In *Proc. ACM Int. Conf. Functional Programming*, pages 45–57, 2011.
20. Ruy Ley-Wild. *Programmable Self-Adjusting Computation*. PhD thesis, Carnegie Mellon Univ., 2010.
21. Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: a programming language for Ajax applications. In *Proc. ACM Conf. Object Oriented Programming Systems Languages and Applications*, pages 1–20, 2009.
22. A. M. Pitts and I. Stark. Observable properties of higher order functions that dynamically create local names, or: What's new? In *Proc. Math. Foundations of Computer Science*, pages 122–141, 1993.
23. A. Pnueli. The temporal logic of programs. In *Proc. Symp. Foundations of Computer Science*, pages 46–57, 1977.
24. P. Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *Proc. ACM Symp. Principles of Programming Languages*, pages 307–313, 1987.