Collecting Hollywood's Garbage

Avoiding Space-Leaks in Composite Events

Tomas Petricek

Faculty of Mathematics and Physics
Charles University in Prague, Czech Republic
tomas@tomasp.net

Don Syme

Microsoft Research, Cambridge, UK
dsyme@microsoft.com

Abstract

The reactive programming model is largely different to what we're used to as we don't have full control over the application's control flow. If we mix the declarative and imperative programming style, which is usual in the ML family of languages, the situation is even more complex. It becomes easy to introduce patterns where the usual garbage collector for objects cannot automatically dispose all components that we intuitively consider garbage.

In this paper we discuss a duality between the definitions of garbage for *objects* and *events*. We combine them into a single one, to specify the notion of garbage for reactive programming model in a mixed functional/imperative language and we present a formal algorithm for collecting garbage in this environment.

Building on top of the theoretical model, we implement a library for reactive programming that does not cause leaks when used in the mixed declarative/imperative model. The library allows us to safely combine both of the reactive programming patterns. As a result, we can take advantage of the clarity and simplicity of the declarative approach as well as the expressivity of the imperative model.

Categories and Subject Descriptors D.3 [*Programming Languages*]: Processors—Memory Management (garbage collection)

General Terns Algorithms, Design, Languages

1. Introduction

Writing applications that need to react to events is becoming increasingly important. A modern application needs to carry out a rich user interaction, communicate with web services, react to notifications from parallel processes, or participate in cloud computations. The execution of reactive applications is controlled by events. This principle is called *inversion of control* or more anecdotally *the Hollywood principle* ("Don't call us, we'll call you"). Dealing with inversion of control affects not just the programming model, but also the memory model.

The techniques for writing reactive applications can be categorized either as *declarative* (also called *data-flow*) or *imperative* (or *control-flow*). The first approach is based on composing programs from primitives (e.g. Lustre [10], techniques originating from Fran and FRP [1, 17]). When using the second approach, we encode programs as sequences of imperative actions (e.g. Imperative Streams [3], Esterel [11, 12], but also Erlang [13]).

F# provides libraries for both types of reactive programming

models [19, 18]. The common concept shared by both of them is the notion of *event*. It represents an asynchronous output channel of some running computation. Application's components can register with events to receive values from the channel. Combining the two programming models allows the user to choose the approach more appropriate for a particular task.

In this paper, we look at garbage collection in this scenario. For events, we need to consider not only whether the event value is reachable, but also whether it can have any effect. An incorrect treatment can lead to unexpected behavior and can cause memory leaks when combining the two styles of reactive programming. In particular, the key contributions of this paper are the following:

- We review two reactive programming models for a mixed functional/imperative language (Section 2) and analyze a design issue, arising when combining them. This problem is present in the current version of F# libraries (Section 3).

- We state which events are garbage and show that this definition is dual to the notion of garbage for objects. We compose these two concepts into a single one that is useable for an environment containing both events and objects (Section 4). To build a better intuition, we present a formal algorithm for garbage collection (Section 5) in this environment.

- Finally, we present an implementation of combinators for declarative event-driven programming in F#, which does not suffer from memory leaks (Section 6) and we show how it follows from our formal model (Section 7).

Our approach is pragmatic, so we target mainly established platforms. We aim to show how to develop a correct, memory-leak free reactive library without modifying the GC algorithm and using only features available at most of the platforms.

We present the examples in F# (ML family) which should be more familiar to the academic community. However the concepts from the paper are directly applicable to mixed functional/imperative languages including Scala, C# 3.0 or Python. It is also highly relevant to modern JavaScript frameworks such as [24, 25].

2. Reactive programming

We review the two approaches to reactive programming using examples from F#. As our main interest is the memory model and we cover the semantics of the programming models only informally. We start by looking at the unifying concept of an *event*.

2.1 Event as the unifying concept

Events in F# appear as an abstract type which allows the user to register and unregister handlers. A handler of type `Handler<'a>`, is a wrapped function (`unit -> 'a`), with a support for comparison via reference equality. We use a simplified event type in this paper, so we define it as a record:

```
type IEvent<'a> =
  { AddHandler    : Handler<'a> -> unit
    RemoveHandler : Handler<'a> -> unit }
```

A simple event keeps a mutable list of registered handlers (in a closure) and invokes all handlers from the list when it is triggered.

2.2 Declarative event handling

In the declarative style, we write code using an algebra of events (combinator library) that allows us to compose complex events from simple ones. The following example demonstrates how the declarative approach looks in F#. We take a primitive event `btn.MouseDown` (representing clicks on a button named `btn`) and constructs a composed event value called `rightClicks`:

```
1: let rightClicks = btn.MouseDown
2:   |> Event.filter (fun me ->
3:       me.Button = MouseButtons.Right)
4:   |> Event.map (fun _ -> "right click!")
```

We're using the pipelining operator `|>` (also known as the reverse application), which takes a value together with a function and passes the value as an argument to the function. This means that the `MouseDown` event will be given as an argument to `filter` function and the overall result is then passed to `map`.

The `MouseDown` event carries values of type `MouseEventArgs` and is triggered whenever the user clicks on the button. We use the `filter` primitive (line 2) to create an event that is triggered only when the value carried by the original event represents a right click. Next, we apply the `map` operation (line 4) and construct an event that always carries a string value "right click!". This approach is similar to Fran [2] and has the following properties:

- **Composability.** We can build events that capture complex logic from simpler events using easy to understand combinators. The complexity can be hidden in a library.

- **Declarative.** The code written using combinators expresses *what* events to produce, not *how* to produce them.

- **Limited expressivity.** On the other hand, the F# combinator library is limited in some ways and makes it difficult to encode several important patterns (e.g. arbitrary state machine)[1].

2.3 Imperative event handling

In the imperative style, we attach and detach handlers to events imperatively. To create more complex events, we construct a new event and trigger it from a handler attached to other events.

In F#, we can embed this behavior into asynchronous workflows [4] (essentially, a one-shot continuation with additional features such as cancellation). A workflow allows us to perform long-lasting operations without blocking the program. When running an *asynchronous operation*, the operation starts and registers a callback that will be triggered when the operation completes.

To work with events, we can define a primitive asynchronous operation `AwaitEvent`, which takes an event value, waits for the first occurrence of the event and then runs the continuation. The implementation relies on imperatively registering a handler when the asynchronous operation starts and unregistering it when the event occurs for the first time (a complete implementation is available in Appendix A [27]). We can use this primitive to get a powerful imperative programming model which is similar to Imperative Streams [3] or the Esterel language [11, 12]:

---

[1] It is possible, but there is no obvious "natural" encoding of a state machine (such as using mutually recursive functions).

```
1: let clickCounter = new Event<int>()
2:
3: let rec loop count = async {
4:   let! _ = btn.MouseDown |> Async.AwaitEvent
5:   clickCounter.Trigger (count + 1)
6:   let! _ = Async.Sleep 1000
7:   return! Loop (count + 1) }
8: loop 0 |> Async.StartImmediate
```

The listing shows a function `loop`, which asynchronously waits for an occurrence of the `MouseDown` event (line 4) and then triggers the `clickCounter` event (created on line 1) with the incremented number of clicks as an argument. Next, it asynchronously waits one second and recursively calls itself and starts waiting for another click. Finally, we imperatively start the loop (line 8). The sane example implemented using event combinators is shown in Appendix B [27] and is much harder to understand.

When the `AwaitEvent` operation completes the handler registered with the `MouseDown` event is unregistered, so all clicks that occur while sleeping (line 6) are ignored (there is no implicit caching of events). This means that the code shows a counter of clicks that limits the frequency of clicks to 1 click per second. In general, this approach has the following properties:

- **Imperative.** The code is written as a sequence of operations (e.g. waiting for an event occurrence) and modifies the state of events by (un)registering handlers or by triggering events.

- **Single-threaded.** Code in this style can be single-threaded using cooperative multi-tasking implemented using coroutines. This makes the concurrency in the model deterministic.

- **Composable.** Even though the implementation is imperative, the created event processors can be easily composed. In the listing, we constructed an event value `clickCounter`, which can be published, while the `loop` function remains hidden.

- **Expressive.** We can easily encode arbitrary finite state machines using mutually recursive functions (using the `return!` primitive). In the example, we have a simple case with just two states: 1) waiting for click and 2) waiting one second.

So far, we demonstrated the reactive programming model using the F# implementation. However, the approach is by no means limited to F# or some highly specific programming environment. It mainly relies on the support for higher order functions, which is now present in many languages including C# 3.0, Scala, Python, Ruby, but also for example JavaScript.

2.4 Compositional events in other environments

An implementation of event-based programming model is already available for C# 3.0 [5] and JavaScript library [24] builds on similar ideas. We created a simple implementation of F# event combinators in JavaScript (available on our web site [27]). The following example shows JavaScript version of the code from section 2.2. Note that this is very similar to code written using the, nowadays very popular, declarative jQuery library [25]:

```
1: var rightClicks = $("btn").mouseDown.
2:   filter (function (me) {
3:     return me.button == 2; }).
4:   map (function (_) {
5:     return "right click!"; });
```

The listing starts by accessing a primitive event value representing clicks on a button (line 1). The event value provides a `filter` function for filtering events and `map` for projection. We use them to create an event that carries the specified string value (line 5) and

is triggered when the button value is 2, corresponding to the right click (line 3). As JavaScript doesn't have any equivalent to asynchronous workflows from F#, the imperative example would be slightly more complicated, but it can be implemented as well.

2.5 Combining event handling techniques
We can view the declarative programming style of event processing as a higher-level approach. It allows us to write a limited set of operations in a way that is succinct, elegant and easy to reason about. On the other hand, the imperative style is lower-level, but as the previous discussion shows, it is extremely important for the ability to easily express state machines.

Now that we have two complementary approaches, it seems like a perfect solution to combine them and use the one that's more appropriate for the part of the problem that we need to solve. Unfortunately, combining the techniques brings some important implementation challenges.

3. Problems with mixing styles
When using the declarative style alone, we don't concern ourselves with removing handlers, because the event processing pipeline remains active during the entire application lifetime. However, when mixing the styles, the `AwaitEvent` primitive needs to add and remove a handler each time we use it.

3.1 Event-based programming model
In our reactive library, events are values like any other. In the formal model, we'll distinguish between these two constructs. This allows us to treat *objects* and *events* differently in the GC algorithm. This is desirable as events are in many ways special.

Private references. One notable property of events is that all references to other events or objects are private. They are captured in a closure and cannot be accessed by the user of the event. This has an important practical implication for our implementation. If an event e_1 references event e_2 and object o_1 references e_1, we cannot directly access the event e_2 from code that uses o_1.

Created by combinators. When developing the reactive programming library, we'll use the described garbage collection techniques only for collecting events that are created by declarative event combinators. Notably, due to the limited set of combinators, this guarantees that there won't be any cyclic references between events. Our formal model (Section 5) is fully general, but our reactive library (Section 6) takes advantage of this simplification. In the absence of cycles, we can safely use a variant of reference-counting in the library implementation (Section 6.6).

Side-effects. Our programming model can be embedded in an impure functional language, so the predicates provided as parameters to combinators (e.g. `Event.map`) may contain side-effects. It is not intuitive when and how often the side-effects should happen, so they are discouraged. However, we make the following very weak guarantees that are fulfilled by both implementations discussed in this paper as well as [5]:

- When a handler is attached to an event created by a combinator with an effectful predicate, the side-effect is executed *one or more times* when the source event occurs.

- When no handler is attached to the event, the effect may be executed *zero or more times* when the source event occurs.

Next, we'll explore an example that motivated this paper. It will clarify which events should be garbage collected.

3.2 Disposing processing chains
It is possible to implement event combinators using a very simple pattern[2]. In this pattern, each combinator creates a new event (a stateful object that stores a list of event handlers) and registers a handler to the source event that triggers the created event:

```
1: let map f (src: IEvent<_, _>) =
2:    let ev = new Event<_>()
3:    src.AddHandler(Handler(fun x ->
4:       ev.Trigger(f x)))
5:    ev.Publish
```

The listing shows the `Event.map` combinator. It registers a handler (lines 3, 4) to the source event and when the event occurs, it applies the function `f` to the carried value and triggers the created event. As you can see, it never unregisters the handler that was attached to the source event using the `AddHandler` member.

Let's demonstrate what exactly happens when we create an event processing chain using several combinators, add an event handler, wait for the first occurrence of the event and then remove the handler. This behavior is just a special case of what we can write using the `AwaitEvent` function:

```
1: let awaitFirstLeftClick src k =
2:    let clicks = src.MouseDown
3:       |> Event.filter (fun m ->
4:          m.Button = MouseButtons.Left)
5:       |> Event.map (fun m -> (m.Y, m.X))
6:    let rec hndl = new Handler<_>(fun arg ->
7:       clicks.RemoveHandler(hndl)
8:       k arg)
9:    clicks.AddHandler(hndl)
```

The function takes a continuation `k` and a source event `src` as parameters. It uses event combinators to create an event that is triggered only when the source event was caused by a left click (lines 2 to 5). On the line 6, we create a handler object (using F# value recursion [20]). When it is called, it unregisters itself from the event and invokes the continuation (lines 7, 8). Finally, the function registers the handler returning a unit value as the result.
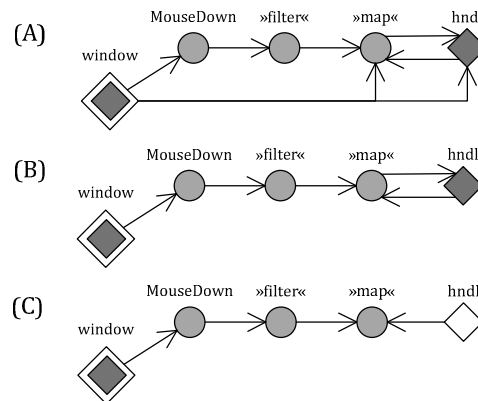

Figure 1. References in the processing chain after it is created (A); when references from `window` are lost (B); after the handler is removed (C); events are shown as circles, objects as diamonds; window is marked as a root object; white color means that object or event is not referenced (and will be garbage collected)

The Figure 1 shows objects and references between them, which are created by running the previous function. When constructing the event clicks, each combinator creates a reference from the source event to the newly created event (by registering an event handler). The initial situation, right after running the line 7, is shown in (A). The executing program still keeps references to the local values of the function (hndl and »map«) and the closure of the hndl value references the event »map« (captured because of the reference on line 5).

We can see what happens when the function returns in (B). The stack frame for the call is dropped, so the root object loses direct references to the constructed event and the handler. At this point, we don't want to dispose any part of the chain! It can still do something useful (run the handler and trigger the continuation). As the diagram shows, there are still references from window to all events, so the implementation above behaves as we need.

The situation displayed in (C) is more interesting. When the event occurs, the handler unregisters itself from »map«. There are no other references to hndl and it can be garbage collected. The rest of the processing chain isn't disposed, because it is still referenced from the root. Arguably, the chain cannot do anything useful now, as there are no attached handlers. When the source event occurs, it will only run the specified predicates, which is allowed, but not necessary (as discussed in 3.1).

More importantly, when we add and remove handlers in a loop (e.g. the example in section 2.2), we'll create a large number of abandoned events that cannot be garbage collected. Obviously, this isn't the right implementation.

4. Garbage in the dual world

Due to the inversion of control, event-driven applications are in a way dual to "control-driven" applications. To our knowledge, this duality hasn't been described formally in the academic literature, but it has been observed by Meijer [5]. He explains that a type representing events is dual to a type representing sequences. Interestingly, we can use the principle of duality when talking about garbage collection in the reactive scenario as well.

4.1 Garbage in worlds of objects and events

In section 3.1, we've discussed a case where an event intuitively appeared to be useless, but wasn't disposed by the GC, because it was still referenced. This example suggests that we need a different definition of "garbage" for reactive applications.

Formally, we model references between objects as an oriented graph $G = (V, E)$ consisting of *vertices* V and *edges* E. A set of *roots* $R \subseteq V$ models objects of a program that are not the subject of garbage collection (such as objects currently on the stack etc).

A vertex $v \in V$ is *object-reachable* iff there exist a path $(v_o, \dots, v_n, v)$ where $v_o \in R$. Objects that are garbage are those that are not *object-reachable*. $\quad$ (2)

Let's now focus on events. In section 3.2, we stated that events are useless if they cannot trigger any handler. We'll define this notion more formally. We take *leaves* $L \subseteq V$ to be events with attached handlers. Then the event value is useful if we can follow references from it and reach one of the leaf events (meaning that triggering of an event can cause some action). If we were in a world where everything is an event and the events are triggered from the outside, then we could use the following definition:

A vertex $v \in V$ is *event–reachable* if and only if there exists a path $(v, v_o, \dots, v_n)$ where $v_n \in L$. Events that are garbage are those that are not *event-reachable*. $\quad$ (3)

We can observe that the definition of *event-reachable* (3) is equivalent to the definition of *object-reachable* (2) in the inverted reference graph (taking *leaves L* as *roots R*).

This explains why we were referring to the duality principle in the introduction of this section. The reactive world isn't dual only when it comes to types, but also when it comes to the definition of garbage.

4.2 Garbage in the mixed world

In practice, we're working with an environment that contains both objects and events. When collecting garbage, we want to mix the two approaches. We want to follow the *object-reachable* definition for objects and *event-reachable* definition for events.

Combining the two notions requires some care. We will take the *roots R* of the graph to be the root *objects*, but how can we incorporate events? In this section, we'll look at an intuitively clear way to define collectability for a mixed world. We start by distinguishing between objects and events:

Let vertices V be a union of two disjunct sets $V_e \cup V_o$ where V_e is the set of *events* and V_o is the set of *objects*.

Events in the mixed environment aren't triggered from the outside, but by other *events* or *objects* that reference them. This means that events that are not *object-reachable* are also garbage in the mixed world. A more subtle problem is determining *leaf events* that "are useful". We will explain the definition shortly.

We define the set of leaf events T as follows:

$$T = \left\{ v \in V_e \,\middle|\, \begin{array}{l} \exists v_o \in V_o\colon v_o \text{ is } object\text{-}reachable \\ \text{and } \big((v_o, v) \in E \vee (v, v_o) \in E\big) \end{array} \right\} \quad (4)$$

An object or event $v \in V$ is *collectable* if and only if it is not *object-reachable* given roots R or if it is an event ($v \in V_e$) and it is not *event-reachable* given leaves T. $\quad$ (5)

As already discussed, we check *object-reachability* of both events and objects (5). For events we apply an additional rule, using a constructed set of event *leaves T*. The elements of T in (4) are defined as a disjunction of two conditions. The first one specifies *events* that are directly referenced by some *object*. In this case, we mark them as "useful", because they can be directly accessed by program and the program may intend to register a handler with them at some later point. The second condition specifies *events* that directly reference some object, which corresponds to the fact that there is some registered event handler.

The definition is demonstrated by **Error! Reference source not found.**. As we can see, all events directly referenced by objects or referencing an object (handler) are marked as leaves. Using the first part of (5), we mark objects and events in the lower part as garbage, because they are not referenced from the root object. The upper part demonstrates events that become garbage due to the second part of the definition. They are referenced from the *root object*, but there is no path leading to any *leaf event*.

In the next section, we will describe a garbage collection algorithm for the mixed environment of objects and events, taking advantage of the aforementioned duality.
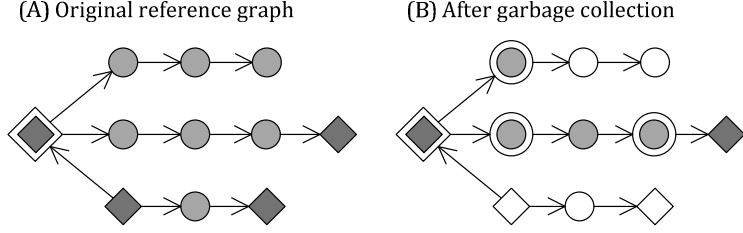
(A) Original reference graph   (B) After garbage collection

Figure 2. Mixed garbage definition. (A) shows initial reference graph that mixes objects (diamonds) with events (circles); root objects and calculated leaf events are marked with rings. (B) shows which objects and events are garbage (filled with white color).

5. Garbage collection algorithm

Implementing a specialized GC algorithm is a difficult task in practice, so we instead describe how to build an algorithm using a standard GC algorithm for collecting objects that are not *object-reachable*. Such algorithms are already well-understood and are implemented and optimized on many platforms.

5.1 Constructing the algorithm

The input of our algorithm is a reference graph $G = (V, E)$ with a set of root objects R. It works in three steps. The first two steps perform pre-processing dealing with the integration of environments and the third step uses the duality principle.

Pre-collection. As already discussed, only events that are referenced by an object or another event can be triggered. As a result, we first need to collect objects and more importantly also events that are not *object-reachable* using (2). This corresponds to running a GC algorithm on the original reference graph containing both events and objects. This can be described as follows:

$$V^{pre} = \{v \in V | v \text{ is } object\text{-}reachable\}$$

$$(V^{pre}, E^{pre}) = G^{pre} = G[V^{pre}] \qquad (6)$$

The graph G^{pre} is a subgraph of G induced by the reduced set of vertices V^{pre}. We'll also need a reduced set of object vertices $V_o^{pre} = V^{pre} \cap V_o$ and event vertices $V_e^{pre} = V^{pre} \cap V_e$. We can easily see that the constructed graph doesn't contain any *collectable* objects or events as defined by the first part of (5).

Mock references. From this point, we want to treat events separately from objects, so events will no longer keep other objects alive by referencing them. To make sure that all *object-reachable* will remain *object-reachable* we add *mock references* to simulate chains of events. We'll add references from event sources (objects that reference events) to all event handlers (objects that are referenced by events) that are reachable by following only event vertices. More formally:

$$E^{pre'} = E^{pre} \cup \{(v, u) | v, u \in V_o^{pre} : p_e(v, u)\}, \qquad (7)$$

$$p_e(v, u) = \exists path(v, v_1, \ldots, v_n, u) : n > 0 \land \forall i : v_i \in V_e^{pre}$$

The predicate p_e states that there is a path between two vertices, which visits only *events* and its length is at least two. By adding edges to the graph (7), we ensure that all event handlers that can be triggered by an event will be referenced. Note that these event handlers correspond to the second part of the condition specifying leaf events T in the definition (4).

Duality principle. The previous two steps performed pre-processing that is necessary when we want to integrate events and objects. Now we can use the key idea of this article, which is the duality between the definitions of *object-reachable* and *event-reachable*. We construct a transformed *garbage graph* G^*:

$$G^* = \left(V^{pre}, \{d(e) | e \in E^{pre'}\}\right) \text{ where} \qquad (8)$$

$$d(u, v) = \begin{cases} (u, v) \text{ when } u \in V_o^{pre} \\ (v, u) \text{ when } u \in V_e^{pre} \end{cases}$$

The function *d* reverses references leading from an event to any other vertex. It unifies the notion of *object-reachable* from (2) and *event-reachable* from (3). This allows us to handle the second part of the collectability definition (5) using a standard GC algorithm for passive objects. Finally, we run it on the *garbage graph G^**:

$$V^{fin} = \{v \in V^{pre} | v \text{ is } object\text{-}reachable \text{ from } R \text{ in } G^*\} \qquad (9)$$

The definition (9) gives us the final set of objects and events that are not garbage. To get the final reference graph, we take the result of pre-processing (6) and take a subgraph induced by V^{fin}.

5.2. Garbage collection example

Before we discuss the correctness of the algorithm, let's look at Figure 3, which demonstrates the construction steps using a minimal example with most of the important situations.
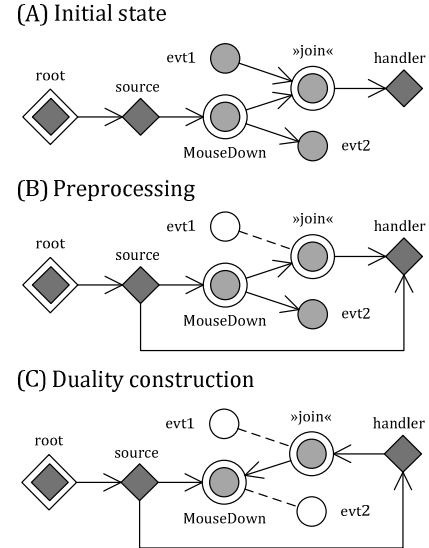

(A) Initial state

(B) Preprocessing

(C) Duality construction

Figure 3. Graph construction. Objects are shown as diamonds, events as circles; dashed lines are references to events and objects that were garbage collected.

The diagram (A) shows an initial state with a *root* object referencing an event source. The source can trigger a MouseDown event, which is propagated through the chain to a handler. During the pre-processing, we run garbage collection to remove events that cannot be triggered, which removes the evt1 event and we also add a mock reference from source to handler, which allows us to reverse references of the chain (B).

Finally, (C) shows what happens after reversing the edges that lead from events. Thanks to this operation, we start applying the dual rule to events, so all events that cannot trigger any *leaf* event become garbage (e.g. evt2 in the diagram). However, thanks to pre-processing, we don't affect collectability of non-events. The algorithm should intuitively follow the definition, but a formal proof of its correctness is available in Appendix C [27].

5.3 Removing events during collection
As noted earlier, we use the formal algorithm mainly as a formal model and it motivates the implementation of a reactive library in the next section, but first we briefly discuss an aspect that would be important for an actual implementation of the algorithm.

When removing an event from the memory (for example evt2 in Figure 3), there won't be any references to it from objects, because that would make the event a leaf event and it wouldn't be collected. However, it may be referenced from other events. To avoid dangling references, we need to deal with these possible references when collecting the event.

Ideally, the garbage collector would have some knowledge about events and it could remove the reference to the collected event from the source event (for example if an event in the actual system contained a list of referenced events). Another approach would be to redirect the reference to a special *null event*, which doesn't perform any action when triggered.

6 Implementing the reactive library
The algorithm proposed in section 5 has the advantage that it can be built on top of a standard garbage collection algorithm, but we wanted to avoid modifying the runtime altogether. In this section, we discuss an implementation of combinators which is inspired by the previous discussion and implements almost identical behavior.

As discussed in section 3.2, a naïve implementation of event combinators registers a handler to the source event (using the AddHandler function). It keeps a mutable list of handlers and returns a new IEvent<'a> value, which adds or removes handlers to or from this internal list. When the source event fires the combinator chooses whether to trigger the handlers and also calculates a value to use as an argument. This implementation is faulty, because it never removes the handler from the source event.

6.1 Implementation requirements
Let's briefly summarize the requirements for the new implementation. Section 3.2 already discussed the most important aspect:

- **Collectable event chains.** When the user unregisters all previously registered handlers and when there is no other reference to the event value, the entire processing chain should be available for garbage collection. This corresponds to the definition of garbage for events from section 4.

- **No explicit referencing.** When there is a handler attached to the event value representing the chain, the chain shouldn't be collected as the handler can still perform some useful work.

- **Stateful.** As we'll clarify in section 6.2, the current semantics of event combinators is stateful, meaning that the mutable state of a combinator can be observed by multiple handlers. We want to preserve this semantic property.

- **Composability.** When we create a chain using a sequence of pipelined Event.xyz transformations, we should be able to freely split it into several independent pieces (e.g. we should not require adding a special combinator to the end or the beginning of the event processing pipeline).

The original implementation didn't meet the first requirement. However, when designing a library that satisfies the first condition, it is easy to accidentally break another one. The requirement for stateful combinators deserves more explanation.

6.2 Stateful and stateless model
Certain combinators maintain a state. The same state is shared by all handlers attached to a single event value. We can demonstrate this behavior using one more example that counts clicks:

```
1: let counter = btn.MouseDown
2:   |> Event.map (fun _ -> 1) |> Event.scan (+) 0
3:
4: let rec loop() = async {
5:   let! num = counter |> Async.AwaitEvent
6:   printf "count: %d" num
7:   return! loop() }
```

We take the MouseDown event and project it into an event that carries the value 1 each time the button is clicked. The stateful combinator Event.scan uses the second argument as an initial state (line 2). Whenever the source event occurs, it uses the function provided as the first argument (in our case +) to calculate a new state from the previous one and the value carried by the event. The returned event is triggered (carrying the current state) each time the state is recalculated.

Next, we switch to the imperative event handling style and implement a processing loop that prints the current count whenever counter event fires. The loop repeatedly attaches a handler to the event (line 5), so the code works only if the state stored by the event is shared between all event handlers.

Let's compare the two possible implementations of event combinators that maintain a state between two occurrences of the event (such as Event.scan):

- **Stateless.** In this model, we create a unique instance of the state for each attached event handler. This model keeps the code referentially transparent, but it works well only in a purely declarative scenario. If we ran the example above using a stateless implementation, it would print 1 for every click, which is somewhat unexpected.

- **Stateful.** The state of an event created by the combinator is shared between all handlers. This approach is consistent with the imperative event handling (the example in section 2.2 works this way). As the previous example demonstrates, the two styles work very well together in this setting.

The stateless approach has its benefits, especially for pure languages. It has been used in Haskell [2] and is also being used by the Reactive Framework [5], which builds on LINQ [7]. We'll follow the pragmatic ML tradition and use the stateful implementation.

The next section shows an implementation inspired by the formal algorithm that follows all the technical requirements.

6.3 Constructing event chains
We'll discuss the implementation by looking at an example similar to the problematic case from section 3.2, but we'll also analyze other important situations. We start by constructing an event chain and a handler that we'll later attach to the composed event:

```
1: let clicks = src.MouseDown
2:   |> Event.filter (fun m ->
3:       m.Button = MouseButtons.Left)
4:   |> Event.map (fun m -> (m.Y, m.X))
5:
6: let hndl = new Handler<_>(...)
```
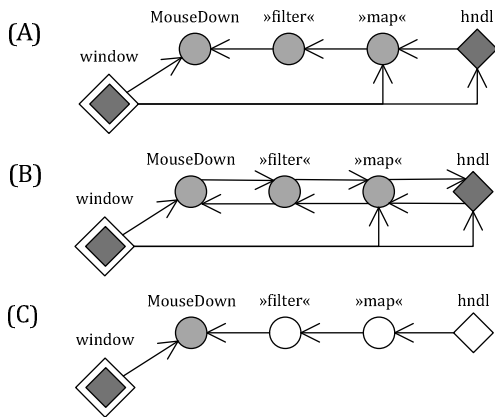
Figure 4. Implementation using reversed links. The state right after creating an event chain (A); after registering a handler, forward references are created (B). Removing the handler also removes forward references. If we also lose direct references from the program, the entire chain becomes collectable (C).

Figure 4 (A) shows what happens after running this code. Our implementation doesn't attach any event handler to the original event source. Each event only keeps a reference to the source, so that it can attach the handler later. We call this *lazy initialization*. If we compare this diagram with the references in the naïve implementation (Figure 1, A), we can see that edges leading from events are reversed, which corresponds to our construction in (9).

6.3 Adding event handlers

Now that the chain is initialized, let's look at what happens when we register `hndl` as a handler for the event constructed by the `map` function (displayed as »map« and aliased by the `clicks` value). This can be done by calling `clicks.AddHandler(hndl)`.

The state after adding the handler is displayed in Figure 4 (B). There is a new link from the »map« event to the `hndl` value. This represents the fact that the event keeps a reference to the handler, so that it can trigger the handler. More interestingly, there are also new links from the event source along the entire event chain up to the »map« value. This is a result of the lazy initialization.

When we register the handler, the event checks whether the number of registered handlers was originally zero. If that's the case, it means that it hasn't yet registered an event handler for its source, which is the »filter« event in the diagram. The »filter« event performs the same check and possibly also registers event handler with its source. This way, the registration is propagated up to the primary source, which is an external event.

Thanks to the propagation, the call to `AddHandler` adds all the necessary *forward references*, so when the primary source event (`MouseDown`) occurs, all the event transformations will trigger and the provided handler will be called. Note that before adding the event handler to the constructed `clicks` event, the transformations wouldn't be called (and none of the functions we provided as an argument to `Event.map` and other combinators would run).

6.4 Removing handlers and losing references

There are several situations that can arise after we add an event handler. In this section, we'll analyze interesting combinations of removing the event handler and losing references to the chain.

Losing references. If the program returns from a routine that constructed the chain, it loses references to both the handler and the chain. As follows from our definition of garbage and the requirements (Section 5.1), we don't want to dispose the event chain, because if the source event fires, it can trigger some useful handler.

We can see that our implementation behaves correctly in this case. Even if we lose the direct references to »map« and `hndl`, there are still references from the source to the handler. The links that keep the handler "alive" are forward references, which were created by propagating the call to `AddHandler`.

Removing handler. If we unregister the handler, but keep a reference from the program to the constructed event chain, the event checks whether the number of handlers reached zero (after the removal). If yes, it propagates the removal and removes its handler from its source event. Again, this may continue up to the original source event. This is a valid approach, because if there are no event handlers to trigger, we don't need to listen to the source.

The state of the event chain after adding and removing a single handler is exactly the same as the initial state after creation. In the diagram, we return back to the state in Figure 4 (A). This means that the whole event chain is still in a usable state. As long as we keep a reference to the object representing the chain »map« we can again attach a handler and start listening to the event.

Removing handler and losing references. Now, let's look at the situation that motivated this paper. What if we remove the event handler and then lose references to the object representing the event chain (or remove handler after losing references)?

In this case, the event chain becomes garbage. No one is listening to it, so it doesn't need to fire and we cannot attach an event handler to the chain, because we don't keep any reference to it. The original implementation of combinators is deficient in this case, because it keeps references from the event source to the end of the event chain, even though there are no attached handlers.

Our implementation using reversed links behaves differently. After removing the last handler, the removal is propagated, so there are no forward references in the chain. As shown in Figure 4 (C), the only remaining references are reverse links from the end of the chain. We don't keep any reference to the chain, so all events that form the chain become garbage and can be collected.

The same situation occurs if the handler references the »map« value and unregisters itself, which was our original motivation (Section 3.2). Until it does so, there are forward references, which keep the event chain alive, but once the last handler is removed, the removal is propagated and the chain is garbage collected.

6.5 Implementing sample combinator

In this section, we demonstrate our implementation of event combinators, by examining the `map` combinator. We already explained several aspects of the implementation:

- Due to laziness, no handler is registered with the source event when the combinator is called.
- When the first handler is added, the combinator adds a handler to its source event.
- Similarly, when the last handler is removed, the combinator also unregisters itself from the source event.

The implementation directly follows these rules. It is more complicated than the version in section 3.2, but it can be simplified by composing combinators using primitive higher-order functions.

```
1:  let map f (source:IEvent<_>) =
2:    let list = new ResizeArray<Handler<_>>()
3:    let this = new Handler(fun arg ->
4:      for h in list do h.Invoke(f arg))
5:    let add h =
7:      list.Add(h)
8:      if list.Count = 1 then
9:        source.AddHandler(this))
10:   let remove h =
12:     list.Remove(h)
13:     if list.Count = 0 then
14:       source.RemoveHandler(this))
15:   { AddHandler = add; RemoveHandler = remove }
```

The combinator first initializes a mutable list of registered handlers (line 2) and a handler (line 3) that will be later attached to the source event. The handler implements the projection, so when it is called, it iterates over all currently registered handlers and invokes them with the projected value as the argument (line 4).

On the next few lines, we define two local functions that will be called when the user adds a handler to the returned event (line 5) and when a handler is removed (line 10). Both of the functions have similar structure. They first add or remove the handler given as an argument to or from the list attached handlers. Next, they check whether the change should cause propagation and register with or unregister from the source event (lines 9 and 14).

6.6 Relations to standard GC techniques
At this point, it is worth discussing the relations between our implementation and standard garbage collection techniques.

Reference counting. As noted in section 3.1, our algorithm bears similarity to reference counting. A well-known problem of reference counting is that it fails to reclaim objects with reference cycles. We rely on GC implemented by the runtime when collecting objects or events that are not *object-reachable* and uses reference counting to collect events that are not *event-reachable*. This means that only cyclic references between events would be a problem. However, as noted in section 3.1, the set of combinators provided by F# library doesn't allow creating cyclic references between events, so the problem is avoided in our setting

Weak references. Many runtime environments support advanced features for controlling the garbage collector, such as *weak references* and so it seems natural to ask whether these could be used in our implementation. We consider using weak references for forward or backward links, but as we'll see both of these uses would break the implementation.

If forward links were weak, the GC could collect events with attached handlers that perform some useful work (e.g. if we registered a handler with the event constructed in section 2.2 and then lost reference to the event chain). On the other hand, if backward links were weak, the GC could collect parts of a chain before registering the first handler (e.g. the »filter« event in Figure 4 (A)). This discussion raises some interesting problems for future research (Section 8.1).
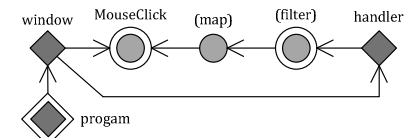
7 Correspondence with the model
In the previous two sections of the paper, we've introduced a formal model and an implementation that is inspired by the model. In this section, we discuss the correspondence between the algorithm from section 5 and the library implementation.

(A) **Initial:** Same in both models
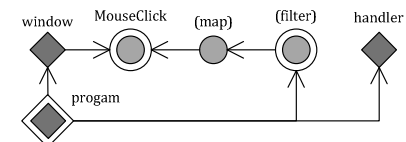
(B) **Registered:** Garbage graph

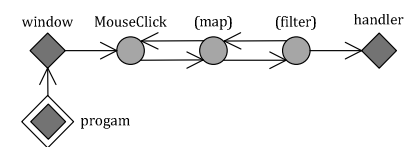(C) **Registered:** Implementation

Figure 5. Formal algorithm and implementation. In the initial state, the implementation and the formal model give the same result (A). After registering a handler, the formal model (B) and the implementation (C) differ, but are equivalent in terms of reachability.

7.1 Duality of event references
When we create an event using a combinator, it only keeps a reference to previous event of the chain. This corresponds to the formal model, where we reverse all links leading from an event (8). If we look at a simple case, where none of the pre-processing steps take place, the formal model and implementation yield exactly the same result. An example is shown in Figure 5 (A).

Once we attach an event handler to the constructed event, the situation becomes more complicated. We'll discuss this case in 7.2. The other pre-processing step is collecting unreferenced event values, which we'll discuss in section 7.3.

7.2 Mock references
When an event references a handler (object) in the formal model, we add a *mock reference* (7) from the event source (an object that references the first event of the chain) to the event handler (an object referenced by the last event). On the other hand, the implementation adds forward references by propagating the registration of the handler back to the source.

The Figure 5 demonstrates what happens in the formal model (B) as well as in the implementation (C). It shows the situation in which we're not directly referencing the event chain and the handler from the program (other cases would be similar). Even though the diagrams look different, it isn't difficult to observe that they are equivalent in terms of collectability of objects.

This equivalence is in details discussed in Appendix D [27]. Briefly, a mock reference in the formal model corresponds to a situation when a handler was registered in the implementation, which causes the propagation of handler registration back to the original event source. Now, events that become reachable thanks to mock references correspond to events that become reachable after registration propagation. To show this formally, we need to analyze the paths to events of the event chain. Informally, you can verify this by looking at Figure 5 (B) and (C).

7.3 Event collection

The first pre-processing step in the formal model guarantees that we will collect all events that can't be triggered. In the implementation, there is no feature corresponding to this step. Let's demonstrate this mismatch using an example:

```
1: let test() =
2:   let form = new Form(Visible=true)
3:   form.MouseDown
4:      |> Event.filter (...)
5:      |> Event.map (...)
6: let evt = test()
7: evt.AddHandler(Handler(fun e -> printf "!"))
8: eventsList.Add(evt)
```

When we invoke the function in the code snippet (line 6), it constructs a visible form and returns an event. We add a handler to the event (line 7), and keep a reference to it in some global list of events (line 8). When the form is later closed, the runtime no longer keeps a reference to it, so the form as well as the constructed event chain should be collected (with the exception of the last event, which is directly referenced).

However, because of backward references, there is a link from the last event to all preceding events in the chain, so the entire event chain is kept alive. We don't find this limitation a problem in practice. In order to keep the event chain alive, the user needs to reference the constructed event value, but keeping a list of created events is rarely done in practice.

Similar situation arises in .NET and can be solved using the "Weak Delegate" pattern [21]. We provide combinator `Event.weak` [8] which allows the user to overcome this problem. This issue is, however, orthogonal to the one that motivated this paper.

8. Related work and conclusions

In this section, we first look at programming models that are related to those available in F# (Section 2) and then review the related work in the area of garbage collection.

Synchronous languages. The reactive programming model for mixed functional-imperative languages is related to the work in synchronous languages [6]. Our model isn't synchronous and isn't aimed at the development of real-time embedded systems, but it shares similar aspects with two of the synchronous languages.

The declarative programming model from 2.1 is similar to the data-flow model of the Lustre language [10], as we also declaratively construct computations and have limited expressive power. In Lustre this makes programs verifiable. On the other hand, the imperative programming model from section 2.2 is similar to Estrel [11, 12], which is described as "imperative and suited for describing control" in [6]. However, since Lustre and Estrel exist separately and cannot be mixed, the authors didn't have to face the integration problem that motivated this paper.

From Fran to Reactive Framework. Our research is inspired by functional reactive programming, started by Fran [1, 2]. Modern FRP [17] inspired the design of F# event combinators, which is the declarative model discussed in section 2.1. The imperative model from section 2.2 is more similar to Imperative Streams [3].

Meijer's Reactive Framework [5, 9] is very relevant to our work, because it is also directly useable from the F# language. It builds on the declarative model of composing computations using combinators. It provides large number of combinators, which makes it very expressive, but for example encoding state machines in this model is not straightforward.

Reactive Framework overcomes the memory problems by being stateless. It creates a new "instance" of an event chain for every handler and then disposes the entire chain when the handler is removed. This is compelling, but we cannot use it cannot be used with the stateful semantics of F# combinators.

Garbage collection research. When collecting events, we aim to collect objects and events that we consider as garbage, but that wouldn't be collected by the standard GC algorithm. This is related to research that provides the algorithm with an additional liveness property and collects not only objects that are *unreachable*, but also objects that are not *alive*. (e.g. [22, 23]).

The question whether we could collect objects matching the definitions in section 4 using similar techniques is an interesting future research problem that complements our work. One notable difference is that an event can be considered garbage even if it can still be triggered. As discussed in 5.3, this problem would have to be handled carefully in a GC algorithm implementation. Liveness analysis usually deals with objects that won't be accessed by the program, but the GC algorithm fails to recognize this.

Garbage collection of actors. The Actor model [16] describes concurrent programs in terms of active objects that communicate using messages, which is in many ways similar to the popular language Erlang [13]. Our imperative model is related to the actor model – waiting for an event plays the role of receiving a message and triggering an event corresponds to sending a message.

Actors face a similar problem, because garbage is defined in terms of the state of the actor (however, it cannot be expressed using the duality principle as in our work). Various specialized algorithms for collecting the garbage have been developed for the actor model [14]. However, the most relevant work is [15], which describes how to implement garbage collection for actors in terms of standard garbage collection algorithm by translating the actor reference graph to a passive object reference graph. The specific steps performed by the algorithm are very different, but the general approach of manipulating the reference graph and using a standard GC algorithm is shared with our work (Section 5).

8.1 Future work

In this paper, we have focused on the definition of garbage and finding an implementation that doesn't cause memory leaks in the situations when we combine declarative and imperative style of reactive programming as introduced in sections 2 and 3. However, it would be also desirable to study the semantics of event combinators more formally.

Combinator semantics. The alternative implementation of combinators, which is presented in this paper doesn't change the meaning of any of the pure combinators (such as `Event.map` and `Event.filter`), but it slightly modifies the semantics of stateful combinators (e.g. `Event.scan`). The semantics remains stateful (as required in section 5.1), but differs from the original F# implementation. In particular, when we initialize `Event.scan` in the original implementation, it starts updating its internal state right away even before we attach a handler. On the other hand, our implementation updates the internal state only when there are some handlers attached to the event constructed by our combinator. In fact, it is difficult to judge which of the implementations is more intuitive, so we believe this isn't a problem in practice as long as it is clarified in the library documentation. However, it shows the need for a more formal treatment of the semantics of event combinators.

Garbage collection in reactive scenarios. Our paper deals with garbage collection in a concrete reactive programming model built using events. However, similar problems have been observed for other programming models such as the actor model [15]. This suggests that we may need more powerful GC algorithms for reactive programming in general. It is not yet clear to us whether the work on liveness analysis [22, 23] can provide a more general solution. The algorithm we proposed serves mainly as a useful formal model for the design of reactive library, but it would be interesting to see if it can be generalized to cover all known reactive scenarios and implemented in practice.

Another problem is to show that garbage collection in the reactive (or more generally, in any non-standard) memory model in fact needs a specialized GC algorithm. In section 6.6, we discussed why we cannot use weak references in our model, but there are other advanced features such as ephemerons [26] that may provide the necessary expressive power. In general, this demonstrates the need for a solid framework for formal reasoning about the expressivity of garbage collection techniques.

8.2 Discussion

This paper provided a review of reactive programming models that can be used in mixed functional/imperative languages and we looked at the F# implementation of these models. We discussed a problem with garbage collection of events, which emerges when we attempt to mix the declarative and imperative style and is also present in the current implementation of F# event combinators.

We defined the problem formally by providing a simple definition of *collectability* for reactive programming model. It combines both *objects* and *events* and benefits from the duality principle. Using this definition, we presented a garbage collection algorithm, which reclaims all *collectable* objects and events. The algorithm is based on graph transformation. This technique could be used to reduce the problem to well known GC algorithms.

However, our implementation aim wasn't to actually replace a garbage collection algorithm. Instead, we have shown an alternative implementation of library of F# event combinators, solely in terms of object references. Our implementation closely corresponds to the formal model and doesn't cause memory leaks when used in an environment that combines both declarative and imperative approach to reactive programming.

Acknowledgements

References

[1] C. Elliott and P. Hudak, Functional reactive animation. *In Proceedings of ICFP 1997, pp. 263-273*

[2] C. Elliott. Declarative event-oriented programming. *In Proceedings of PPDP 2000*

[3] E. Scholz. Imperative streams - a monadic combinator library for synchronous programming. *In Proceedings of ICFP 1998*

[4] D. Syme, A. Granicz, and A. Cisternino. Expert F#, Reactive, Asynchronous and Concurrent Programming. *Apress, 2007.*

[5] E. Meijer. LiveLabs Reactive Framework. *Lang.NET Symposium 2009, Available at: http://tinyurl.com/llreactive*

[6] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, R. De Simone. The Synchronous Languages Twelve Years Later. *In Proceedings of the IEEE, vol. 91, pp. 64-83, 2003*

[7] E. Meijer, B. Beckman, G. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. *In Proceedings of COMAD 2006*

[8] T. Petricek, D. Syme. Collectable event chains in F#. *To appear as MSR Technical Report.*

[9] Microsoft. Reactive Extensions for .NET. *Retrieved from: http://msdn.microsoft.com/en-us/devlabs/ee794896.aspx,2010*

[10] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, LUSTRE: A declarative language for programming synchronous systems. *In Proceedings of POPL 1987.*

[11] F. Boussinot and R. de Simone, The Esterel language. *In proceedings of the IEEE, vol. 79, pp. 1293–1304, 1991.*

[12] G. Berry and G. Gonthier, The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *In Science of Computer Programming vol. 19, n°2, pp 87-152, 1992.*

[13] J. Armstrong, R. Virding, C. Wikström and M. Williams, Concurrent Programming in ERLANG, 2nd ed. *Prentice Hall International Ltd., 1996.*

[14] D. Kafura, D. Washabaugh, and J. Nelson, Garbage collection of actors. *In OOPSLA'90, vol. 25(10), pp. 126–134*

[15] A. Vardhan and G. Agha, Using Passive Object Garbage Collection Algorithms for Garbage Collection of Active Objects. *In Proceedings of ISMM'02*

[16] G. Agha. Actors: A Model of Concurrent Computation in Distributed Systems. *MIT Press, Cambridge, Mass., 1986.*

[17] Z. Wan and P. Hudak. Functional Reactive Programming from First Principles. *In Proceedings of PLDI, 2000*

[18] T. Petricek and J. Skeet. Real-World Functional Programming, Chapter 16, *Manning, 2010.*

[19] D. Syme. Simplicity and Compositionality in Asynchronous Programming through First Class Events. *Online at: http://tinyurl.com/composingevents, Retrieved: Jan 2010*

[20] D. Syme. Initializing Mutually Referential Abstract Objects. *In Proceedings of ML Workshop, 2005*

[21] G. Schechter. Simulating "Weak Delegates" in the CLR. *Online at: http://tinyurl.com/weakdelegates, Retrieved: Feb 2010*

[22] O. Agesen, D. Detlefs and J. Eliot B. Moss. Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines. *In Proceedings of PLDI 1998.*

[23] R. Shaham, E. K. Kolodner and M. Sagiv. Estimating the Impact of Heap Liveness Information on Space Consumption in Java. *In Proceedings of ISMM 2002.*

[24] L. Meyerovich, A. Guha, J. Baskin, G. Cooper, M. Greenberg, A. Bromfield, S. Krishnamurthi. Flapjax: A Programming Language for Ajax Applications. *In OOPSLA 2009.*

[25] The JQuery Project. jQuery. *Available at http://jquery.com*

[26] B. Hayes. Ephemerons: a new finalization mechanism. *In Proceedings of OOPSLA 1997.*

[27] T. Petricek, D.Syme. Collecting Hollywood's Garbage Avoiding Space-Leaks in Composite Events (Extended version) *Available at: http://tomasp.net/academic/event-chains.aspx*