Collecting Hollywood's Garbage
Avoiding Space-Leaks in Composite Events

Tomáš Petříček
Faculty of Mathematics and Physics
Charles University in Prague
tomas@tomasp.net

Don Syme
Microsoft Research
Cambridge, UK
dsyme@microsoft.com

Talk overview

Introduction and motivation

- Imperative and declarative reactive programming
- Space leaks in composite events

The key idea of the talk

- We discuss objects and events separately
- We combine them into a single model

How to use it in the real-world?

- How to solve this using a novel GC algorithm
- How to solve this in a library

Event-driven programming

- Specify reaction imperatively by adding *handler*:

```
let onRightClick info =
  if (info.Button = MouseButtons.Right) then
    label.Text <- sprintf "[%d, %d]" info.X info.Y
win.MouseDown.Add(onRightClick)
```

Function declaration

Register as handler

  - We can add and remove handlers to the *event* (e.g. `win.MouseDown`)

- Reactive programming with events in F#

```
let locations = win.MouseDown
  |> Event.filter (fun info -> info.Button = MouseButtons.Right)
  |> Event.map (fun info -> sprintf "[%d, %d]" info.X info.Y)

locations.Add(fun msg -> label.Text <- msg)
```

Only right clicks

Register handler

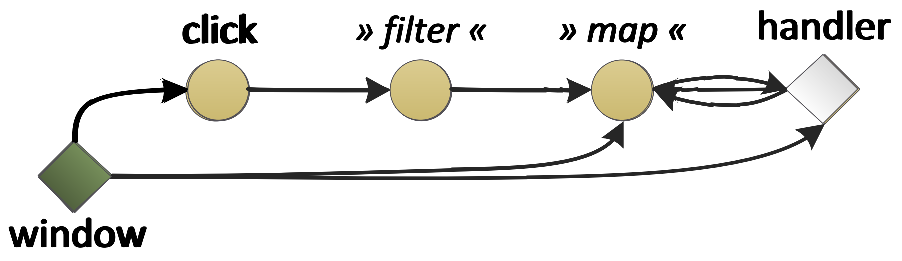
Carries information as formatted string

  - Event (e.g. `Event<string>`) modeled as a list of time/value pairs
  - We construct new events declaratively and use them imperatively

Introducing memory-leaks

- Implementing event combinators (e.g. `Event.map`)
  - Create new event value, add new handler to the source event, trigger the returned event when triggered

- **Problematic scenario:** removing handler imperatively



1. Construct event using combinators
2. Add handler that removes itself when triggered
3. Lose references to the constructed event and handler
4. When event fires, the event chain becomes garbage

   Question: When should we consider event garbage?

Reactive programming in practice

- Declarative event handling is useful!
  - Better ways for providing abstractions
  - Allows composable descriptions of handling
  - Works well with syntactic extensions (e.g. query comprehensions)

- Used in several research and real-world projects
  - Functional programming languages
    - Event combinators available in F# libraries
    - Functional Reactive Programming research in Haskell
  - .NET implementation using C# query comprehensions
    - Reactive Extensions to .NET *(Meijer, 2009)*
  - Client-side web programming libraries
    - jQuery and Flapjax libraries *(jQuery, 2010), (Meyerovich, et al. 2009)*

Talk overview

Introduction and motivation

- Imperative and declarative reactive programming
- Space leaks in composite events

The key idea of the talk

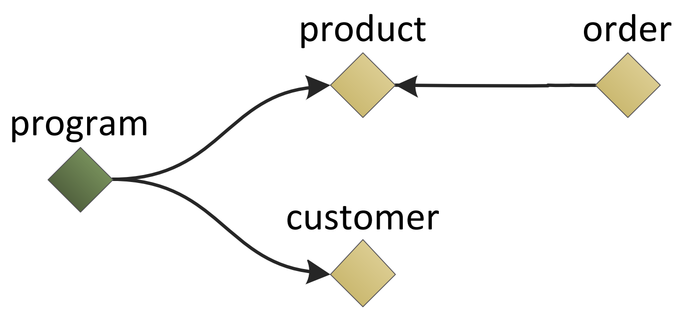
- We discuss objects and events separately
- We combine them into a single model

How to use it in the real-world?

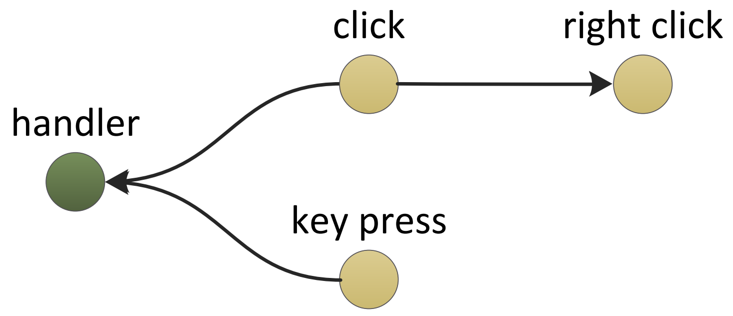
- How to solve this using a novel GC algorithm
- How to solve this in a library

Introducing the dual world

- World of passive objects
  - Assuming that the root object (green) has the control initially
  - Object can temporarily transfer the control to a referenced object
  - Objects that are in control can perform observable actions

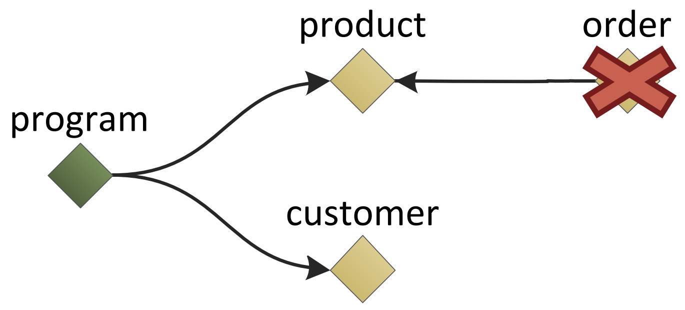


- World of active events
  - Assuming that any event can be triggered from the "outside"
  - When triggered, source event can trigger the target event
  - Only special events (green) can perform observable actions

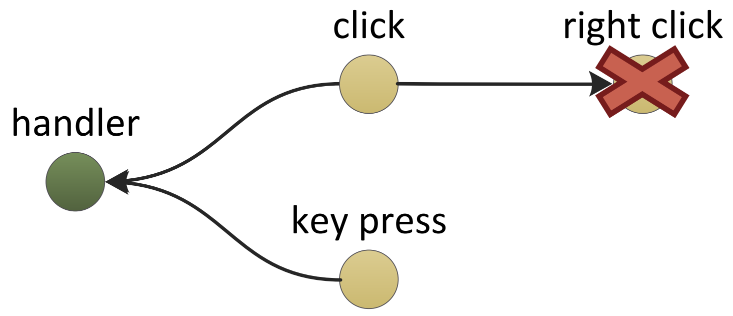


Garbage in the dual world

- World of passive objects
  - Oriented graph with a set of *roots* (special objects in green)
  - Object is *object-reachable* if there is a path to it from some root.
  - Unreachable objects can be garbage collected



- World of active events
  - Oriented graph with a set of *leaves* (special events in green)
  - Event is *event-reachable* if there is a path from it to some leaf.
  - Unreachable events can be garbage collected

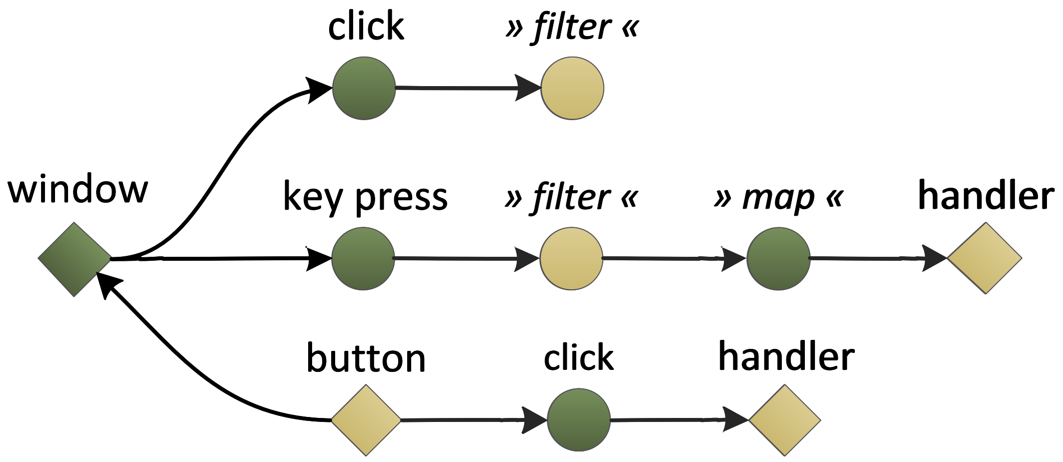


Reactive model based on events

- Interesting challenge for garbage collection
  - Astonishing duality between reactive events and passive objects
- Integrating the two dual worlds
  - Programming model with both objects and events
  - Events are not triggered from the "outside" but by objects
  - Leaf events are those that directly interact with objects
- Related work in the GC community
  - **Actors** – concurrent systems modeled as active objects *(Agha, 1985)*
    - There are specialized GC algorithms for the Actor model
  - **Liveness** – collect objects not *alive* *(Agesen, et al., 1998), (Shaham, et al., 2002)*
    - Difference – event considered garbage even if it can still be triggered (!)

Integrating events and objects

- Distinguish between objects (diamonds) and events (circles)
- Example scenario
  - Window is a root object
  - First chain doesn't have handler attached
  - Third starts from an unreferenced object



- Garbage collection in the mixed world
  1. Event is a leaf iff there is reference to it from object (it is directly referenced) or from it to object (has handler attached)
  2. Object or event is *collectable* iff it is not *object-reachable* from root object or if it is event and is not *event-reachable* from calculated leaves

Talk overview

Introduction and motivation

- Imperative and declarative reactive programming
- Space leaks in composite events

The key idea of the talk

- We discuss objects and events separately
- We combine them into a single model

How to use it in the real-world?

- How to solve this using a novel GC algorithm
- How to solve this in a library

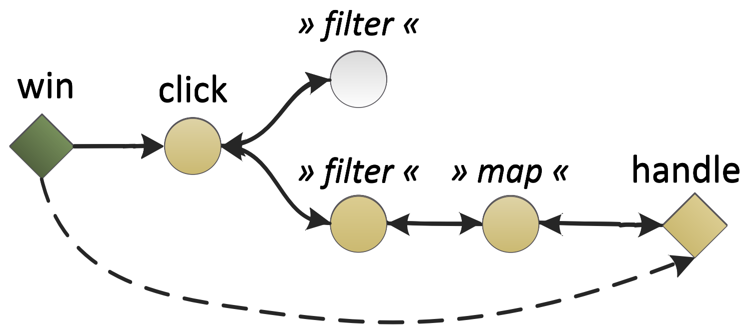
Collecting events in the real-world

- Garbage collection algorithm
  - Specialized GC algorithm would work with events and objects
  - Alternative – transform the reference graph *(Vardhan and Agha, 2002)*
  - Could be implemented using standard GC algorithms
  - There may be a better way to do this!

- Correct implementation of a reactive library
  - Modify event combinators to avoid the motivating issue
  - Event combinators in F# are limited
    - All special events are created by event combinators
    - We cannot construct recursive event loops
    - As a result, we can use “reference-counting-like” solution
  - There may be a better way to do this!
    - *Weak references* don’t seem to work, but there are other concepts…

Reference graph transformation

- Pre-processing steps
  - First, identify *leaf events*
  - Add *mock references* to keep reference to objects reachable by event chains

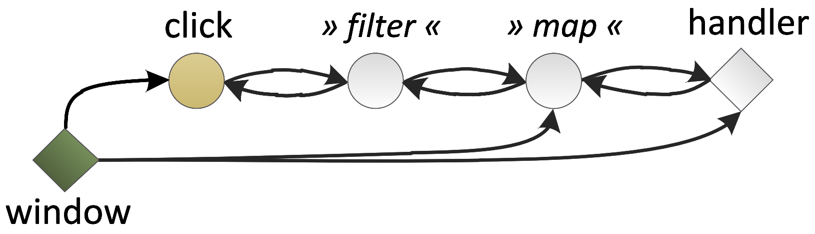


- Key idea – use the duality
  - Reverse all references in the graph leading to events
  - Now, we can use standard GC algorithm for passive objects
  - Collects objects according to the earlier (mixed world) definition

- Some notable practical aspects
  - **Removing events** – There may be references to collected events
    - Invalidate the reference or redirect it to an *null event*

Reactive library implementation

- Combinators create events with “reference counting”
  - Created with *backward references* to source events (based on duality principle)
  - When first handler is added, add *forward references* (so that handler can be run)
  - Direct reference to active events or handlers are not needed
  - When last handler is removed, we remove *forward references*



- Simple, but works well in practice
  - Combinators don’t allow recursive references
  - Events are also objects, but created in a special way

Thanks for your attention!

Summary

- Reactive programming with events is an important area
- Duality between object and event references is interesting
- Can be used in GC algorithm or in library implementation

Tomáš Petříček (tomas@tomasp.net) and **Don Syme** (dsyme@microsoft.com)

Thanks to: Dmitry Lomov, Wes Dyer, James Margetson, Claudio Russo, Anonymous referees of the paper for useful feedback, Microsoft Research for partial funding

Stateful and stateless events

- Some event combinators maintain state
  - E.g. `scan` – updates state each time event fires, using given function

```
let counter =
  btn.MouseDown |> Event.map (fun _ -> 1)
                |> Event.scan (+) 0

counter.Add(fun num -> printf "A %d" num)
counter.Add(fun num -> printf "B %d" num)
```

Carries "1" every time

Accumulate using initial value 0 and +

Immediately after 'counter' is created…

Later, after 'counter' fires a few times…

- Stateful approach *(our work)*
  - All occurrences will get the same number (e.g. `A0, A1, A2B2, A3B3`)
  - More difficult – events are active, standalone objects
- Stateless approach *(Meijer, 2009)*
  - Separate state for every handler (e.g. `A0, A1, A2B0, A3B1`)
  - Removing handler can kill the entire chain

Possible concurrency issues

- Reactive applications are often single-threaded in practice
  - F# reactive library uses the main GUI thread
  - JavaScript-based solutions are single-threaded
- The duality principle
  - The principle doesn’t specify any technical details
- GC using graph transformations
  - Transformation would make parallel GC difficult
  - Standard GC algorithm can run concurrently
- Reactive library
  - Depends on standard GC algorithm, which can be concurrent

Selected references

- **Vardhan and Agha, 2002** A. Vardhan and G. Agha, Using Passive Object Garbage Collection Algorithms for Garbage Collection of Active Objects. *In Proceedings of ISMM'02*
- **Agha, 1985** G. Agha. Actors: A Model of Concurrent Computation in Distributed Systems. *MIT Press, Cambridge, Mass., 1986.*
- **Agesen, et al., 1998** 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.*
- **Shaham, et al., 2002** 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.*

Related technologies

- **Meijer, 2009** E. Meijer. LiveLabs Reactive Framework. Lang.NET Symposium 2009, Available at: http://tinyurl.com/llreactive
- **Meyerovich, et al. 2009** L. Meyerovich, A. Guha, J. Baskin, G. Cooper, M. Greenberg, A. Bromfield, S. Krishnamurthi. Flapjax: A Programming Language for Ajax Applications. *In Proceedings of OOPSLA 2009.*
- **jQuery, 2010** The JQuery Project. jQuery. Available at http://jquery.com