Glendale Project: Benchmarks

Hesam Samimi,Yoshiki Ohshima, Todd Millstein,
Alan Borning

VPRI Research Note RN-2013-001

Viewpoints Research Institute, 1209 Grand Central Avenue, Glendale, CA 91201 t: (818) 332-3001 f: (818) 244-9761

Glendale Project: Benchmarks

Hesam Samimi Yoshiki Ohshima Todd Millstein Alan Borning

Draft of 26 September 2013

1 Introduction

The Glendale language is a playground for exploring a variety of issues in language design. This document provides a brief sketch of the approach we are pursuing along with a set of benchmark examples.

Goals for the Glendale language include the following:

Reactivity Objects should be continuously updated based on events of interest from other objects. Reactivity is useful for interactive graphics among other applications.

Distribution Objects can interact with one another in the same manner regardless of whether they are on the same machine or across the world from one another.

Dynamic Reconfiguration It should be easy to dynamically add, remove, and replace objects in the system.

Dynamic Discovery It should be possible to dynamically discover and interact with other objects of interest.

Strong Encapsulation It should be easy to isolate objects from one another in various ways when desired.

Glendale is being designed as an extension of KScript [4], an implementation of the functional reactive programming (FRP) paradigm [2] in Javascript. The KScript programming model provides reactivity. We augment this model with a *publish-subscribe* mechanism in order to explore the other goals listed above. Our model is loosely inspired by an analogy with the way nodes in the internet communicate and interact.

2 KScript Overview

We augment the programming model of KScript [4], which is an implementation of functional-reactive programming (FRP) paradigm [2, 3] in Javascript.

Each KScript object defines a set of local variables and listening IO events (mouse, keyboard, system timer, etc.) and a set of formulas relating those. Whenever a relevant external IO event occurs the reactive runtime (which keeps track of dependencies) triggers all expressions with transitively affected dependencies to recompute. The new set of values of the local variables are obtained and stored representing the state of the object in the new time step (old values are kept around and can be accessed in the formulas).

VPRI Research Note RN-2013-001

3 The Glendale Model

In Glendale we have a set of (KScript) objects distributed over the network. Communication among objects only occurs through a *publish-subscribe* system (here referred to as the *server*) deployed over this network [1].

3.1 Publishing and Subscribing

Each object in Glendale can *publish* data to the server in the form of tuples in a *relation* (or in Bloom [1]'s terminology a *channel*). The server and the channels don't belong to anybody and are part of the infrastructure "plumbing" the same way internet routers are. Each tuple has a `publisherID` as an implicit first argument, which is inserted by the server and records the object who published the tuple.

As in Bloom, channel relations can be restricted to be *functions*. So depending on the definition publishing a new tuple may mean simply adding to the tupleset in the server, or in the case of a function, possibly replacing an older tuple with the newly received one. Publishing can also involve just removing en existing tuple from the tupleset in the server.

Each object can also *subscribe* to some published tuples by communicating with the server. A subscription request includes the name of a channel and an implicit `subscriberID` inserted by the server.

For example, a triple of the form `property⟨publisherID, propertyID, value⟩` could be used to represent the `value` of the `propertyID` property of object `publisherID`; the value itself may be another object identifier. The object denoted by `publisherID` would publish such tuples in order to provide values of its properties to other objects, who would subscribe to receive them.

When declaring this channel, we can define it as a function: `#property⟨(publisherID, propertyID) -> value⟩`, where the prefix "`#`" is used to denote a channel name. This ensures only one property `value` will be published at a given time for an object and publishing a new `property` tuple will replace the old tuple having `(publisherID, propertyID)` pair as the key, should it have been published earlier.

As another example, a tuple `setProperty⟨publisherID, objectID, propertyID, value⟩` could be used to represent a desire by object `publisherID` to update another object's property to a new value. In this case, the object denoted by `objectID` would subscribe to receive such tuples, which would be published by other objects in order to update `objectID`'s properties. This channel need not be a function since multiple parties can be requesting to update the same object's same property and that's just fine.

Say there's an object identified by `@p` which likes to publish its `"x"` property, stored in its local variable `x`, onto the `#property` channel. Fig. 1 introduces the publishing syntax in Glendale. As we can see the "`<~`" left arrow (borrowed from Bloom) is used instead of "`<-`" to denote a publish instead of the usual local assignment. Note in the figure that we represent tuples using simple KScript dictionary objects and the `publisherID` element in any published tuple stored at key `"publisherID"` is implicit and gets added automatically on the server side.

The Glendale evaluator works as in KScript, recomputing formulas with updated dependencies at each evaluation cycle. Publishing formulas, however, are left alone until the end of an evaluation cycle. At that point the evaluator ships off any published values in the formulas to the server.

```
#property <~ {propertyID: "x", value: x}
```

Figure 1: Publishing example

Another object may then subscribe to this channel to receive `@p`'s published data, as shown in Fig. 2. This time the channel name appears on the right-hand side to denote a subscription. We use a `receiveE` combinator

[1] The publish-subscribe mechanism need not be a single global server assumed here for simplicity, but perhaps itself a network of dedicated and distributed objects.

(same functionality as KScript's `mapE` but specifically for receiving from channels) to receive each arrived tuple from the `#property` channel, access the `value` element of that tuple, and store it in the local variable `x`.

```
x <- #property.receiveE((t) -> t.value) where ((t) -> t.publisherID = @p && t.propertyID = "x")
```

Figure 2: Subscribing example

What follows the `where` keyword is a *subscription filter*—a predicate over the tuple that can accompany a `receiveE` combinator in order to control which published tuples within the specified channel to receive from the server. Because here we are only interested in object `@p`'s `"x"` property the predicate shown is used. The publish-subscribe server has a mechanism to keep track of queries like above for each subscription in order to push only the tuples of interest to subscribers.

In the examples below, we use the more familiar object-oriented notation `@objectID.propertyID` as a shorthand for a request to subscribe to tuples of the form `property⟨@objectID, propertyID, value⟩`.

3.2 Object Identifiers

Object IDs that Glendale programs refer to are not object *references* in the traditional sense. Rather, they are *names* that the server translates into a particular object in the network (not unlike the way a DNS translates URLs). This approach allows for dynamic reconfiguration, since the target of an object ID may change over time, thereby implicitly updating all clients of that object ID. To distinguish object identifiers from regular variables, programs always prefix object identifiers with the "`@`" sign.

3.3 Execution Model

The server maintains a subscription list for each channel. Subscription lists change since subscribers can change their set of subscriptions dynamically during each of their evaluation cycles [2].

When the server receives a subscription, it'll forward any published tuples (should they exist) to the channel as well as every subsequent update to the subscriber.

The server asynchronously receives updates from publishers and processes them one at a time. Upon each new update, the server goes through the subscriptions to the associated channel from each subscriber and pushes the updates to the subscriber. Each subscriber receives new updates to the subscribed channels and reacts to them, one tuple at a time, in the same way it would react to an external IO event in KScript. That is, dependent formulas are reevaluated transitively until quiescence.

3.4 Encapsulation

As described thus far, any object can subscribe to any published tuples. To enforce encapsulation requirements, publishers can put restrictions on who can subscribe to their tuples via an optional *publication filter*—a predicate over a subscription request (s) and the published tuple (t). For example, if object `@a` allows object `@b`, but no other objects, to access its properties, `@a` can provide a publication filter `(s, t) -> s.subscriberID = @b` to its publication of `property` tuples. Filters can place restrictions on any aspect of a tuple; for example we could modify our example filter to further require that the property being accessed by `@b` must be named `"x"`: `(s, t) -> s.subscriberID = @b && t.propertyID = "x"`

[2] Maintaining an object's subscriptions done by the Glendale evaluator during its evaluation cycle for the object can be complicated. For example, the expression `@a.b.c` corresponds to two subscriptions, and the object whose `c` property is being subscribed to depends on the dynamic value of `@a.b`. Some details will be discussed in the example in Sec. 4.

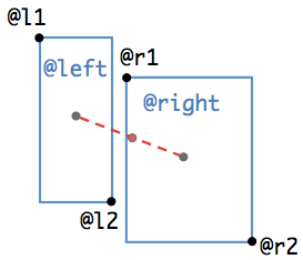

Figure 3: Example 1: The rectangles and points

Analogously, as introduced earlier subscribers can provide an optional subscription filter. However, unlike publication filters, subscription filters are merely a convenience: an object can instead receive all tuples pushed into a given channel and then throw away the ones that are not of interest.

4 Example 1: Rectangle Objects and Internal Point Objects

We'll use a simple example that is inherently point-to-point and has nothing to do with publish-subscribe. Although it'll feel unnecessarily complicated, we'll use it to show the model is more general and therefore handles the traditional examples just fine [3].

Consider the example shown in Fig 3. There are two rectangles (IDs `@left` and `@right`) each internally represented by two points (one at top-left, other at bottom-right). Here we have six independent objects: two rectangles and four points.

4.1 Accessing Other Objects

Let's assume that `@left`'s objective is to compute the `X` coordinate of the mid-point of the line connecting the centers of the two rectangles.

For simplicity, let's say all objects have opted to publish all their properties without any restrictions on who can subscribe. That means points push updates to their `x` and `y` attributes and rectangles their `topLeft` and `bottomRight` attributes to the server in the form of new tuples for the `#property` channel. So the current tupleset of published values on the server may look like shown in Fig. 4

```
property⟨@left, "topLeft", @l1⟩, property⟨@left, "bottomRight", @l2⟩,
property⟨@right, "topLeft", @r1⟩, property⟨@right,"bottomRight", @r2⟩,
property⟨@l1, "x", 0⟩, property⟨@l1, "y", 0⟩, property⟨@l2, "x", 1⟩,
property⟨@l2, "y", 5⟩, property⟨@r1, "x", 1⟩, property⟨@l1, "y", 1⟩,
property⟨@r2, "x", 3⟩, property⟨@r2, "y", 6⟩
```

Figure 4: Example 1 - Published property values

Now let's look inside the formulas for the `@left` object, given in Fig. 5. Since object `@left` is aware of its point objects `@l1` and `@l2` it directly subscribes to their `x` attributes (lines 1 and 2). Assume, however, that this object is not aware of the object IDs for the two points belonging to the other rectangle object `@right`, but it knows about `@right` object itself. Thus it subscribes to `@right.topLeft` and `@right.bottomRight` to obtain these points (lines 3 and 4), and subsequently subscribes to the `x` attribute of each such point (lines 5 and 6). Once all `x` coordinates are obtained the mid-point `x` can be easily computed (as done in lines 7-9).

[3] For example, for local computations what's described in this paper can be only happening "under the hood" and on the surface the programmer may just use traditional message sending.

```
x1 <- @l1.x
x2 <- @l2.x
p3 <- @right.topLeft
p4 <- @right.bottomRight
x3 <- p3.x
x4 <- p4.x
xm1 <- (x1 + x2) / 2
xm2 <- (x3 + x4) / 2
xm <- (xm1 + xm2) / 2
```

Figure 5: Formulas in `@left` to compute the mid-point's `x` value

The above program is reactive. Any time an update for any of the subscribed values is received from the server, all the affected formulas are reevaluated and the object will move into a new time.

4.1.1 Details

How does evaluation of formulas containing subscriptions to the `#property` channel (the "." symbol) work? First, Fig. 6 shows the desugared versions of the program above, showing explicit subscriptions to the `#property` channel. There are six subscriptions there.

```
x1 <- #property.receiveE((t) -> t.value) where ((t) -> t.publisherID = @l1 && t.propertyID = "x")
x2 <- #property.receiveE((t) -> t.value) where ((t) -> t.publisherID = @l2 && t.propertyID = "x")
p3 <- #property.receiveE((t) -> t.value)
    where ((t) -> t.publisherID = @right && t.propertyID = "topLeft")
p4 <- #property.receiveE((t) -> t.value)
    where ((t) -> t.publisherID = @right && t.propertyID = "bottomRight")
x3 <- #property.receiveE((t) -> t.value) where ((t) -> t.publisherID = p3 && t.propertyID = "x")
x4 <- #property.receiveE((t) -> t.value) where ((t) -> t.publisherID = p4 && t.propertyID = "x")
xm1 <- (x1 + x2) / 2
xm2 <- (x3 + x4) / 2
xm <- (xm1 + xm2) / 2
```

Figure 6: Refactored formulas in `@left` show explicit subscriptions

Note that four of the subscriptions have concrete where-clauses (namely those assigned to `x1`, `x2`, `p3`, `p4`), but the other two (`x3`, `x4`) have where-clauses that depend on the dynamic value of local variables which themselves come from other subscriptions. Because of this, the evaluator has to do multiple passes of subscribing actions to fully process one evaluation cycle, as explained below.

The evaluator sends subscription requests only for formulas containing concrete subscription filters. It then waits until those values are received. Once the dynamic values of variables `p3` and `p4` are known (for that time step), the subscription filter for the other two formulas become concrete and the evaluator sends subscription requests for them. The process continues until all formulas have been fully evaluated. That marks the end of the evaluation cycle and the object moves into the new time step.

Given a new update to an object property published onto the server side, the server will forward it to subscribers who have signed up to receive the data. For example, let's assume new update `property⟨@r3, "x", 4⟩` arrives at the server. This is a new object and obviously not in `@left`'s subscription list, so nothing is forwarded to `@left`. Next, the server receives a new update `property⟨@right, "bottomRight", @r3⟩`. This property is subscribed by `@left` so it is sent by the server to `@left`.

At the subscribing object end, the evaluator notices that this update affects the subscription assigned to `x4` (corresponding to expression `@right.bottomRight.x`) and thus sends a subscription request for `#property` channel where `publisherID = @r3` and `propertyID = "x"`, canceling the previous subscription pertaining to the old value of `@right.bottomRight.x`. This causes the server to also forward the update `property⟨@r3, "x", 4⟩` to `@left`.

4.2 Restricting Getting and Setting Properties of Other Objects

Now let us assume `@l1` and `l2` points would like to prevent anybody other than their parent `@left` rectangle to subscribe to their properties. In that case, they can place a restriction `(s, t) -> s.subscriberID = @left` along with their publication to the `#property` channel.

Furthermore, assume `@right` rectangle is allowed to modify the `x` and `y` coordinates of it's internal points `@r1` and `@r2`. But how can these two points prevent any updates from any other objects, including `@left`.

To receive updates from the outside world, `@r1` can subscribe to the channel `#setProperty⟨publisherID, objectID, propertyID, value⟩` adding the filter `(t) -> t.objectID = @r1`. It then can prevent anybody other than `@right` to send it anything by adding the clause `t.publisherID = @right` to its subscription filter [4].

5 Example 2: Multi Chat

The previous example was awkward, since it was really point-to-point communication simulated with the publish-subscribe model. Let's now consider a multi-way chat program to better demonstrate the usefulness of the model.

In the previous example we used the sugar "." to imply an implicit subscription to a channel named `#property`. Now, we'll use the explicit way of publishing and subscribing.

Let us start with a simple case of a broadcast. Say an object `@b` is broadcasting messages in the form of a string, by publishing to a binary channel `#broadcast⟨publisherID, message⟩`. Assuming the object has a variable `outText` holding the outgoing message, Fig. 7 illustrates how every change to the text can be explicitly published into the `#broadcast` channel.

Therefore, the published data in this case should the current value of `outText` variable be `"lol"` will be the object `{publisherID: @b, message: "lol"}` representing the tuple `broadcast⟨@b, "lol"⟩`.

```
#broadcast <~ {message: outText}
```

Figure 7: Example 2 - Broadcaster publishing to `#broadcast` channel

Now any number of objects on the network can subscribe to the `#broadcast` channel to receive every message sent by object `@b`. An example is given in Fig. 8. Thus the variable `inText` will hold every string message sent by the broadcasting object `@b`. Note in this example that the subscriber has no idea from which objects it is going to receive messages.

```
inText <- #broadcast.receiveE((t) -> t.message)
```

Figure 8: Example 2 - A receiver of `#broadcast` channel

Finally, to turn this program into a multi-chat where every user both can send and receive messages, we can simply combine the previous two programs above (see Fig. 9). The only caveat is that an object should add a subscription filter to the `#broadcast` channel to avoid receiving messages sent by itself. This can be done using the `where`-clause as seen in the second line.

[4] A reasonable default behavior can be that objects only allow updates by whoever instantiated them but nobody else.

```
#broadcast <~ {message: outText}
inText <- #broadcast.receiveE((t) -> t.message) where ((t) -> t.publisherID != this)
```

Figure 9: Example 2 - Multi chat program

6 Example 3: The Car and the Wheel

Consider this basic Etoy example. We have a `@car` object that moves forward by 5 units every 1 second in the direction told by another object called `@wheel`. That object's `heading` property is a real value representing the wheel's heading in degrees. The requirement is that another object—the `@user`—should be able to send a message to `@wheel` to set its heading, thereby controlling the direction of the car.

The example is illustrated by Fig. 10. Line 3 uses the syntactic sugar to subscribe to the `heading` property published by an object by the name of `@wheel`. Note that this constitutes two subscriptions: In either case of (1) the actual object that the name `@wheel` resolves to, or (2) the `heading` property of the current `@wheel` undergo change, this should trigger an update to `wHeading` on line 3. On line 5 the displacement is recalculated on every tick of the timer, and the next line computes the new `position` for the car [5].

```
1  timer <- timerE(1000)
2  speed <- 5
3  wHeading <- @wheel.heading
4  heading <- heading' + wHeading
5  displacement <- P(0, speed) fby speed * P(cos(heading), sin(heading)) on timer
6  position <- P(0, 0) fby position' + displacement
```

Figure 10: Example 3 - `car`'s movement controlled by the user by setting to a `@wheel` object's `heading` property

7 Example 4: Bank Account

The next example concerns a bank account object. Multiple users (objects) should be able to view and modify the account concurrently and without atomicity violations.

Let's say the account `@acc` publishes its `balance` property, allowing a subscription only by users `@u1` and `@u2` (see Fig. 11). The two users can then simply subscribe to and access the balance with `@acc.balance` syntax. Alternatively they can do so explicitly, as shown in Fig. 12.

```
#property <~ {propertyID: "balance", value: balance}
      where ((s, t) -> (s.subscriberID = @u1 || s.subscriberID = @u2))
```

Figure 11: Example 4 - Bank account publishing its balance property

```
balance <- #property.receiveE((t) -> t.value)
    where ((t) -> t.publisherID = @acc && t.propertyID = "balance")
```

Figure 12: Example 4 - Users accessing the bank account's balance using the explicit subscribing syntax

Given this setup, any time `@acc` internally modifies its `balance` property the change is pushed to the server and the server forwards it to the two subscribers.

[5] Representing points as primitive values for simplicity.

@acc now wants to add a withdrawal service. We'll set up a channel #withdraw⟨publisherID -> amount⟩ for this purpose (making it a function to specify that only a single withdraw amount can be requested). @acc subscribes to this channel but adds a filter to only receive data from publishers @u1 and @u2. See Fig. 13.

The example involves some concepts from KScript worth noting. The notation v' denotes the previous value of variable v. Also note that the definition of balance starts with 0 fby. Since this variable is not a discrete event but a continuous value, it needs to be always defined. What is given to the left of fby ("followed by") is the initial value and the expression to the right is the value thereafter. Finally in the else branch of the conditional expression we have the special value undefined. This has special semantics for the evaluator: the occurrence of undefined cuts the flow of changes downstream in the dependency graph of variables. So if we visit undefined during the evaluation of balance, reevaluation of any expressions in other variables depending on balance will not be triggered, nor will the current value of balance variable itself be affected [6].

```
amount <- #withdraw.receiveE((t) -> t.amount)
    where ((t) -> t.publisherID = @u1 || t.publisherID = @u2)
balance <- 0 fby (amount <= balance') ? (balance' - amount) : undefined
```

Figure 13: Example 4 - bank account handling withdrawal requests

The above rules along with the publishing of the balance property (Fig. 11) will push the new balance to all subscribers upon a withdrawal. Also the semantics prevent overdrafts caused by concurrent withdrawal requests, as the account object processes one message (i.e. received withdraw tuple) at a time.

For fun, let's consider now a transfer between two accounts. There are two account objects @acc1 and @acc2. @acc1 initiates a transfer of some amount to account @acc2. The requirement is that the sum of the two account balances should never change from the outside world's point of view.

This example requires an inter-object synchronization. @acc1 and @acc2 should agree on going into a transfer protocol, during which they should not proceed on processing new messages (thereby preventing a view of "inconsistent" state).

Let's start by noting that this requirement is impossible to ensure in a purely asynchronous system without global synchronization, since messages can be reordered and indefinitely delayed. But even if we assume published messages are instantaneously pushed to the server and instantaneously received on the subscriber's side, it is possible to view an inconsistent state without some kind of inter-object synchronization, which we'll implement as follows.

We set up two binary channels #transfer⟨publisherID -> amount⟩ and #transferAck⟨publisherID -> amount⟩ for this purpose, which @acc1 and @acc2 will both publish and subscribe to. Each object also defines a property named block. The objects don't react to any incoming messages other than transferAck tuples when block is set to true. @acc1 initiates the process (see Fig. 14) by updating its property named transferRequestAmount (lines 2, 3) and setting block to true (line 5) [7]. Once @acc2 receives the transfer request from @acc1 (see Fig. 15, line 3) it sets its block to property to true (line 5), adds the amount to its balance (line 7), and publishes a transferAck tuple of the same amount for @acc1 (line 9). Upon receiving that, @acc1 subtracts the amount from its balance (back in Fig. 14, line 11), sets block back to false (line 5), and publishes a transferAck tuple for @acc2 (line 13). When @acc2 receives that, it can set its block property back to false (back in Fig. 15, line 5).

Note that in a case when an ack gets lost and never resent, the target object continues in blocking mode, thus the inconsistent state can never be observed from the outside world.

[6] This is not unlike the *cut* operator "!" in Prolog. In this case, we could have just replaced undefined with balance' and that would reassigned the old value of balance to it, yet it still registers as a "change" and therefore can trigger reevaluation downstream. We may or may not want this to happen. It's a domain specific choice, and irrelevant for this specific example.

[7] In KScript we use the mergeE combinator to assign different values to a variable based on which depended upon variable is changed. In this case either change of amount variable or a receive event from #transferAck channel triggers the update in line 5.

```
1 // sanity check transfer amount
2 OKTransferAmount <- (!block' && balance' >= transferRequestAmount) ?
3     transferRequestAmount : undefined
4 // block when starting transfer process, or stop blocking when received ack
5 block <- false fby mergeE(OKTransferAmount.doE(() -> true), #transferAck.receiveE((t) -> false))
6 // publish transfer request only to @acc2
7 #transfer <~ {amount: OKTransferAmount} where ((s, t) -> s.subscriberID = @acc2)
8 // subscribe to receiving ack from target account
9 amount <- #transferAck.receiveE((t) -> t.amount) where ((t) -> t.publisherID = @acc2)
10 // update balance once we have received ack
11 balance <- 0 fby balance' - amount
12 // send an ack for the ack to @acc2
13 #transferAck <~ {amount: amount} where ((s, t) -> s.subscriberID = @acc2)
```

Figure 14: Example 4 - bank account transfer - `@acc1`'s side

```
1 // subscribe to receiving transfer request from @acc1
2 amount <- block' ? undefined :
3     #transfer.receiveE((t) -> t.amount) where ((t) -> t.publisherID = @acc1)
4 // start blocking when amount request received, stop when received ack
5 block <- false fby mergeE(amount.doE(() -> true), #transferAck.receiveE((t) -> false))
6 // update balance
7 balance <- 0 fby balance' + amount
8 // send a transfer ack to @acc1
9 #transferAck <~ {amount: amount} where ((s, t) -> s.subscriberID = @acc1)
```

Figure 15: Example 4 - bank account transfer - `@acc2`'s side

8 Example 5: Linked List of Nodes

There's an object `@list` representing a linked list of nodes, with each node being its own separate object. An object from the outside world knows about (has the address of) `@list` but not its internal nodes. Let's say the goal is to compute the sum of the `value` property stored at each node. We should be able to iterate through the nodes of the list, but should not be allowed to modify it.

We can model this example in several ways. Here is one such way. We can think each node itself maintains a link to the next node, and the `@list` object plays the role of the head node. We'll use a channel `#list_links⟨srcNodeID -> dstNodeID⟩` to represent the list. As before, the `value` property for each node is assumed to be published to the `#property` channel. For instance, let's assume the current set of published tuples are as shown in Fig. 16, representing a three element list `[@list, @n1, @n2]` whose sum of `value` properties is *43*.

```
property⟨@list, "value", 4⟩, property⟨@n1, "value", 30⟩, property⟨@n2, "value", 9⟩,
list_links⟨@list, @n1⟩, list_links⟨@n1, @n2⟩
```

Figure 16: Example 5 - Representation of the linked list on the server

A client object can then subscribe to these channels to get the linked list of nodes and iterate through them. Let's walk through the steps to compute the sum, as demonstrated in Fig. 17.

The `node` variable in line 1 represents the current node we are visiting. If we have haven't previously visited any node, then `node` is the `@list` object, otherwise, the client subscribes to `#list_links` channel (in line 2) to receive the node next to the previous one (`dstNodeID` element in the link `lists_link` relation). Once the node is known the client obtains its `value` by subscribing to the `#property` channel where `publisherID = node` (line 3), and and add it to the previous `sum` amount (line 4). Once all tuples have been received, the `sum` variable should reflect the desired sum.

Note that this is another example where the evaluator has to do careful, extra work to handle subscriptions dynamically. Once the `node` variable has a value (line 1), the subscription filter on the next formula (line 3) becomes concrete and has to be sent to the server, in order to compute the new value for the variable `value`.

```
1  node <- @list fby
2      #list_links.receiveE((t) -> t.dstNodeID) where ((t) -> t.srcNodeID = node')
3  value <- #property.receiveE((t) -> t.value) where ((t) -> t.publisherID = node)
4  sum <- 0 fby sum' + value
```

Figure 17: Example 5 - Linked list client code to compute the sum of node values

9 Example 6: Car with Two Controllers

Let's consider another Etoy example. We have a `@car` and two users `@user1` and `@user2`. The car's `speed` is controlled by `@user1`, while its `heading` is set by `@user2`. At each tick of clock, `@user1` publishes a `speed` while `@user2` publishes a `heading`, which `car` subscribes to in order to determine its displacement. This example involves coordination between the two users, and seems best programmed in a synchronous setting. Since we have not yet come up with any synchronous semantics here, we skip this benchmark at the moment.

References

[1] Peter Alvaro, Neil Conway, Joseph Hellerstein, and William Marczak. Consistency analysis in Bloom: a CALM and collected approach. In *Proceedings 5th Biennial Conference on Innovative Data Systems Research*, pages 249–260, 2011.

[2] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, ICFP '97, pages 263–273, New York, NY, USA, 1997. ACM.

[3] 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 *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 1–20, New York, NY, USA, 2009. ACM.

[4] Yoshiki Ohshima, Aran Lunzer, Bert Freudenberg, and Ted Kaehler. KScript and KSWorld: A time-aware and mostly declarative language and interactive gui framework. In *Onward! '13*. ACM, 2013.