A Lightweight Push-pull Mechanism for Implicitly Using Signals in Imperative Programming

YungYu Zhuang[*]

Department of Computer Science and Information Engineering,
National Central University,
No. 300, Zhongda Rd., Zhongli District, Taoyuan City 32001, Taiwan (R.O.C.)

Abstract

While signals can express time-varying values well, they heavily rely on the semantics of dataflow programming and functional programming. Several research have developed mechanisms for using signals with imperative object-oriented design and shown the benefits of its usage. However, they tend to introduce a class for signals, which thus results in the necessity of lifting up/down between variables and signals. We have already proposed an automation mechanism to expand event systems to support signals without introducing a class, and in this paper, we further extend it to a lightweight push-pull model by considering the direction of trigger. The push-pull automation mechanism allows programmers to choose between *push* and *pull* to declaratively express their intention and to reduce the overheads due to unnecessary propagation of value changes. To show the feasibility of our proposal, we implemented *PuPPy* as an extension to Python for helping programmers in declaring fields as signals. With *PuPPy*, programmers can use signals in Python without any event system and do not have to worry about the type of events and handlers. We evaluated *PuPPy* by running preliminary microbenchmarks and comparing with signal class libraries, pure event systems, and the implementation in our previous research.

Keywords: Event-driven programming, reactive programming, signal, behavior

[⋆]This is an author-produced version of an article accepted for publication in the Journal of Computer Languages following peer review. The published journal article is available at https://doi.org/10.1016/j.cola.2019.100903. © 2019. Licensed under the CC-BY-NC-ND 4.0.

[*]Corresponding author, yungyu@acm.org

2010 MSC: 00-01, 99-00

1. Introduction

Reactive behavior is quite common, especially in programs that have GUI design and data update. For example, in an application drawing 3D graphs on screen, we expect the graph can react to user operations such as mouse dragging and finger gesture. The application should be able to rotate and resize based on our operations without delay. If the application polls the position of mouse cursor at a regular interval, users might need to wait and the user experience is bad. If the data of 3D graphs come from a backend, we also expect to get a notification when the data are updated. To write such kind of programs, one approach is to implement the Observer pattern [1], which allows to subscribe a callback for being executed later. When users dragged the graph or the data were modified, the callback will be executed for reacting to the update. The Observer pattern looses the coupling between objects in an object-oriented design and allows subscribing and unsubscribing a handling at runtime. However, massively implementing this pattern for various response in a program makes it hard to understand and maintain. In a naive implementation, the code for a single subject-observer pair might scatter over different objects, and the code for different subject-observer pairs in a system might be tangled with each other. This breaks the rule of separation of concerns [2]. Although several abstraction such as interfaces and aspects can be used to ease the problem by separating pattern code from others [3], the problem is not completely resolved. Programmers cannot help but implement pattern code. Furthermore, the Observer pattern also encourages programmers to violate several software engineering principles such as side effects, encapsulation, and data consistency as discussed by Maier and Odersky [4].

Event-driven programming can be a replacement for the Observer pattern. In a language that directly supports events, subscription and notification can be simply set up by language constructs and operators without pattern code.

This makes programs easier to describe reactive behavior. Nevertheless, in this approach, the way of considering reactive behavior is basically the same as using the Observer pattern—monitoring the changes of some variables and executing specified functions in response to those changes. In other words, the changes are discrete and handled imperatively. On the other hand, the approach of reactive programming [5] uses signals to represent such changes. This concept describes changes such as the position of mouse cursor as time-varying values but not static states. Signals can be given to functions for drawing an animation rather than a static graph. Instead of imperatively saying how to do for the reactive behavior, it lets programmers to define what the reactive behavior is to improve the abstraction of reactive programs. However, this approach has a functional flavor, and thus there is some space to design a mechanism fitting imperative object-oriented languages. How to make the design of signals simple is an issue.

To address this issue, we propose a push-pull automation mechanism to improve the usage of signals in imperative programming. We base this research on our previous research result, the automation mechanism [6]. In this paper we further discuss the push and pull model upon the automation mechanism to clarify the direction of trigger, and then propose a declarative push-pull model for enabling the automation. The contributions of this paper are twofold. First, we show a source and sink model to discuss the direction of propagation and trigger for signals and events, and propose a push-pull automation based on this observation. Second, we compare the design of signals and events in different systems. We implemented our proposal, PuPPy, as an extension to Python [7] to show how the push-pull automation can bring signals into an imperative object-oriented programming language even without an event system. The reasons why we implemented on Python include its simplicity of syntax and dynamic typing. Python is one of the mainstream languages that are frequently used in scientific computing, and its users might not be programming experts; we expect our users can benefit from signals without worrying about typing. The details of our implementation are mentioned as well.

2. Motivation

The variables we usually talk about in imperative programming store fixed values that represent certain states, but signals are a special kind of variables that hold time-varying values. To distinguish between signals and other variables, in this paper, we use normal variables to refer to those variables that are not signals. The origins of signals might include dataflow programming, but the concept we discussed here comes from functional-reactive programming [5]. Signals are convenient to describe several behaviors in real world, for example hardware signals from devices and the motion of user's mouse input. We can then use signals to compose other signals. For example, suppose that we have two signals named `a` and `b`. Then we can simply compose them to generate another signal `c` with the following signal assignment:

```
c = a + b
```

Whenever the value of `a` or the value of `b` changes, the value of `c` will be automatically updated. Such an assignment looks like giving an expression "= a + b" to a cell named `c` in spreadsheet programs, where `a` and `b` are cell names. Signals are very useful to describe data flows. However, it is not possible to regard all variables as signals since in imperative programming variables are used to statically store states, e.g. fixed values. When we bring this concept to imperative programming, we have to face the problem that how to distinguish signals from normal variables. How could we know whether `a` is just a normal variable or a signal, i.e. should we update `c` whenever the value of `a` changes? To address this issue, signals are usually introduced as a special kind of variables, especially a predefined class in object-oriented languages [8, 9, 10]. Such a class-like structure is used to wrap a fixed value or an expression for updating the value. For example, in REScala, a sophisticated Scala library supporting signals, we can define `a` and `b` as follows [11]:

```
val a = Var(2)
val b = Var(3)
```

Then we can compose them to another signal `c` as shown below:

```
val c = Signal { a() + b() }
```

Since a and b are instances of the Var class, and c is an instance of the Signal class, we have enough information on how we should update the value of c. We explicitly say that a and b are signals rather than normal variables. To get the current value inside a signal, we can use the following method:

```
c.now
```

It returns 5 in this case according to the expression between the braces following the Signal for c. If we use the statements shown below to change the value of a and then get the current value of c again:

```
a set 4
c.now
```

it will return 7 based on the latest values of a and b. As a result, there are two kinds of things for holding states in the language: normal variables and signals. Normal variables hold values statically, while signals hold values dynamically. Here Var is used to lift up a fixed value or a normal variable to a signal, while the now method can be regarded as lifting down a signal to a normal variable in order to get its fixed value. As to Signal, it is a mechanism to lift up a normal variable with its assignment to a signal.

Without signals programmers can use events to propagate value changes, though it might be slightly annoying in such a trivial case. In a typical event system, we need to prepare events for a and b, and manually bind the handler that updates the value of c to these events. For example, in EScala [12], events for a and b look like this:

```
evt ea[Unit] = afterExec(setA)
evt eb[Unit] = afterExec(setB)
```

where setA and setB are the setter functions for a and b. The two lines of code declare two events that will be triggered after the execution of the two setter functions, respectively. Note that Unit is a type in Scala, and declaring an event with Unit means that the event does not carry any value. We can prepare a handler function, for example hc, which contains the assignment "c = a + b":

```
def hc() { c = a + b; }
```

and bind this handler hc to the events for a and b, i.e. ea and eb:

```
ea += hc;
eb += hc;
```

Comparing with signals, programmers need to declare events for the variables and bind handler functions to those events. Although in the design of several event systems such as EScala and C#, events may carry values, basically events do not participate in calculation; events are used for handler bindings. This is quite different from signals, which participate in calculation. Event-driven programming is explicit. Programmers need to prepare events and handlers, then bind them manually. On the other hand, reactive programing is implicit. Programmers do not need to explicitly propagate value changes with additional statements.

This motivated us to propose a declarative push-pull automation based on our previous research result [13, 6]. Although the proposed automation mechanism eliminates the difference between signals and normal variables, there is some room for improving its design and the prototype implementation. First, the direction of triggering the reevaluation in current design is one-way. It always pushes changes to the variable at the left-hand side, i.e. the current value of c in this example, even if we do not need the current value inside it right now. It is possible to avoid unnecessary reevaluation by pulling the changes when we really need the current value as discussed in several research [14, 15]. Second, the way to enable the automation is not declarative. It means that a variable can be a normal variable at the beginning and become a signal later. Such a design is based on event systems and is very flexible, while it might be unnecessary when using them as signals; signals tend to be declarative in order to express data flows without taking runtime conditions into account. In this declarative push-pull proposal, programmers may choose between normal variables and signals when they declare variables, and specify the direction of triggering the reevaluation to make their intention clear and prevent bugs. Nevertheless, programmers do not have to lift up/down between normal variables and signals in the program.

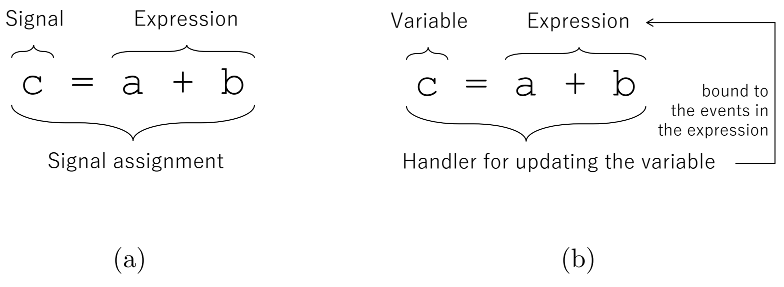

Figure 1: Signal assignments can be translated to events, handlers, and bindings.

3. A two-way automation: push and pull

To satisfy the need of implicitly using signals in imperative programming while avoiding unnecessary propagation of value changes, we propose a two-way model based on the automation mechanism in our previous research result [6]. In order to explain how we base this proposal on it and show the difference, we briefly explain the original one, discuss the direction of trigger and propagation in events and signals, and then explain our push-pull automation proposal. As to the implementation, it is explained in Section 4.

3.1. The original automation mechanism

The automation of handler bindings we proposed can expand event systems to support signals in imperative programming. To explain its idea here, we borrow the figures from those papers [13, 6] and modify them to fit our motivating example as shown in Figure 1. The mechanism is a kind of automation for event systems, which can translate the signal assignment in Figure 1(a):

```
c = a + b
```

where a, b, and c are all signals, to the handler bindings in Figure 1(b):

- a normal variable assignment for c,
- two events ea and eb for marking the value changes of a and b,
- and a handler hc bound to the above two events, which contains the assignment "c = a + b".

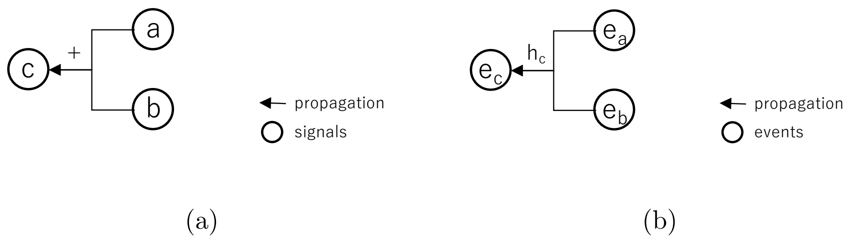

Figure 2: The value changes can be propagated with signals and events, respectively.

However, the automation mechanism only supports one-way trigger direction. When to trigger the propagation of value change is only determined by the events `ea` and `eb`. Even though we do not need the latest value of `c` right now, the reevaluation is triggered. Furthermore, if `ea` and `eb` occur successively and there is no side effect expected to be executed twice, merging the two events can avoid unnecessary evaluation. To distinguish from the new automation mechanism we are going to propose in this paper, we use the term "the original automation mechanism" to refer to our previous proposal [6].

3.2. The direction of propagation

To consider the direction of trigger, we may first think about how value changes of variables are propagated and the direction of propagation. We can regard signals as a way to propagate value changes. As shown in Figure 2(a), we use nodes to represent signals and use edges with an arrow to mark the direction of propagation. In the junction of edges, an operator is noted for showing how the value changes are composed. In Figure 2(a), the value changes of the signals `a` and `b` are propagated to another signal `c`, and the operator used to compose the two propagation is $+$.

On the other hand, events in a typical event system are used to mark changes in module and loose the coupling between modules. Events are flags for invoking the execution of functions without knowing those functions exactly. In imperative programming, value changes are important events since values represent states in a program. We can use events to propagate value changes as what signals do. In this example, in order to propagate the value change of `a` to

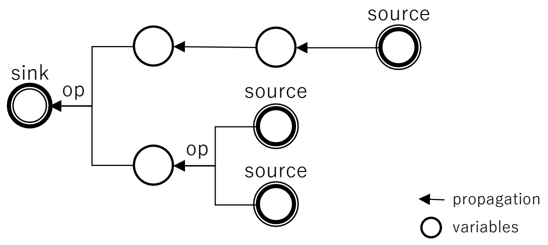

Figure 3: Value changes are propagated from source nodes to sink nodes.

c, we can first prepare an event ea for the variable a and bind the handler hc for updating the value of the variable c to ea. Then once the value inside the variable a changes, the event ea will be triggered, and the handler bound to ea, i.e. hc, will be executed for updating the value of c. With events programmers can avoid to hard-code the function calls for handling value changes and it is even possible to switch between different handling at runtime. When a handler function for the event such as hc modifies the values in other variables, in this example c, it might further trigger other events for those modified variables such as ec, and causes the execution of its handler functions. In other words, the value changes of certain variables are propagated to other variables as what signals do. We can use similar figures to represent such a propagation as shown in Figure 2(b). Note that now the nodes in the figure are events rather than signals, and they are composed by handlers instead of operators. Figure 2(a) can be translated to Figure 2(b) if we prepare a corresponding event for every variable and proper handlers for composing the propagation.

For either signals or events, we can consider such a series of propagation as a process of propagating value changes from *source* to *sink*. We use Figure 3 to unify the figures for showing the propagation of value changes with signals and events. The nodes marked with *source* are the signals or events that start propagating value changes, and the node marked with *sink* is the one we are observing, i.e. the signal or event representing a variable at the left-hand side in the assignment. Figure 3 shows that the direction of propagation is fixed:

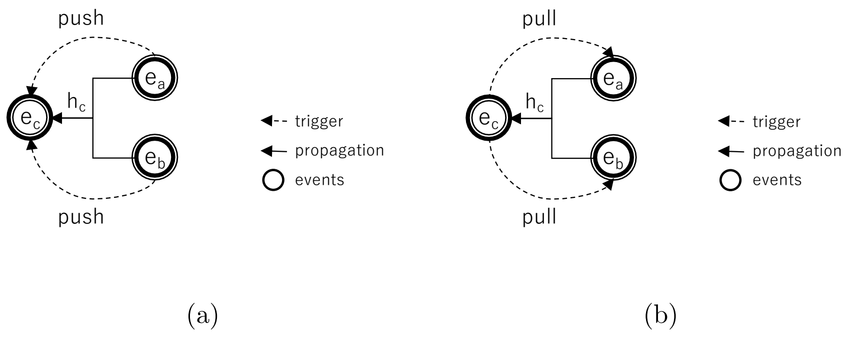

Figure 4: The direction of triggering the propagation can be push or pull.

values changes are always propagated from *source* nodes to *sink* nodes since it represents the direction of assigning the result in the assignment. When we construct the relation through an assignment, the direction of propagation is fixed: propagating value changes from the right-hand side to the left-hand side.

3.3. A push-pull automation mechanism

Traditionally the direction of triggering the propagation is the same as the one of propagation. It means that the timing of starting the propagation is the same as the timing of value changes. *Source* nodes gradually push the changes to *sink* nodes as shown in Figure 4(a). The original automation mechanism made it possible to automatically push value changes of *source* nodes to *sink* nodes. However, we can also let the timing of starting the propagation be the timing of retrieving the value of *sink* node, i.e. pulling the latest values inside *source* nodes when the value of *sink* node is needed as shown in Figure 4(b). The push and pull model is not new and has been discussed in several research, but here we use the model to think about the propagation of value change in the original automation mechanism.

This proposal further considers the direction of trigger in order to make programmers' intention clearer, while preserving automatic inference and implicit binding in the original automation mechanism. We extend the original automation to support both push and pull in a declarative way. Undoubtedly the direction of propagation is fixed, but the direction of trigger can be *push* or

```
 1 def push-pull(modifier, declaration, assignment):
 2     sink = assignment.lhs     # left-hand side
 3     expr = assignment.rhs     # right-hand side
 4     sources = find_sources(expr)
 5     if sources is empty:
 6         fallback_normal(declaration, assignment)
 7     else:
 8         h = create_handler(assignment)
 9         if modifier is 'push':
10             for s in sources:
11                 es = find_changing_events(s)
12                 for e in es:
13                     e.bind(h)
14         else:  # 'pull'
15             es = find_retrieving_events(sink)
16             for e in es:
17                 e.bind(h)
```

Figure 5: The pseudo-algorithm of the two-way automation mechanism.

pull, depending on how programmers specify in the declaration for the variable. The original automation we proposed [6] can be enabled by handler binding statements. In this proposal we limit the enabling to declaration, but let programmers to specify *push* or *pull* model by the corresponding modifiers:

('`push`' | '`pull`')   ⟨*declaration*⟩

The two-way automation mechanism can be enabled by declaring with `push` or `pull` modifier at the beginning. Strictly speaking, the declaration here is a declaration along with an assignment. Figure 5 shows how the two-way automation takes such a declaration and creates handler bindings for programmers. At runtime, value changes will be pushed or pulled to/from the *sink* node according to these handler bindings, respectively. The behavior of the push declaration (Line 9–13) is the same as the original automation in our previous proposal, which ensures every value change from *source* nodes is propagated steadily. On the other hand, the pull declaration will only ask for the propagation when someone needs the value of *sink* node (Line 14–17). This declaration looks like the signal assignment in other systems, but there is no need to explicitly distinguish signals from normal variables in its expression. In the push-pull au-

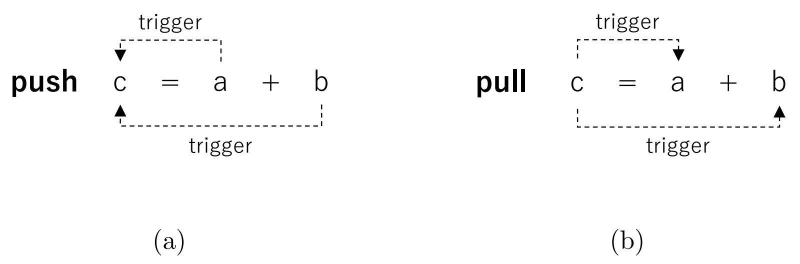

Figure 6: Declaring with the modifier `push` or `pull`.

tomation mechanism, enabling the automation in an event system is declarative rather than dynamic. The automation for an assignment can only be enabled in the declaration that declares the variable we are assigning; it is not allowed to dynamically enable by any kind of statements later. Furthermore, programmers have to clearly specify which model to use for the variable at the left-hand side in this declaration. The variable at the left-hand side of the declaration will be regarded as signals and cannot be reassigned later. This design decision is not only for fixing the trigger model for signals, but also for avoiding toggling between signals and normal variables later. To show the difference between the two models, we add the two modifiers before the assignment "c = a + b" respectively and mark the direction of trigger with dashed lines as shown in Figure 6(a) and Figure 6(b). Note that here type is omitted for simplifying the explanation. In Section 4, we concretely show how to use the push-pull automation mechanism in our prototype implementation.

3.4. source and sink in the push-pull automation

Programmers can use the variables that have been declared with `push` or `pull` at the right-hand side of a push/pull declaration to propagate value changes from source nodes to sink nodes as what it does in Figure 3. For example, the following declarations propagates value changes from the variable a to the variable c through the variable b:

```
a = 0
pull b = a + 1
push c = b + 1
```

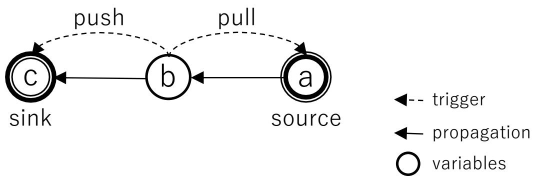

Figure 7: Propagating the value change of a to c with pull and push.

Table 1: The difference of using push and pull for b and c in Figure 7.

for b	**for c**	**when to update b**	**when to update c**
push	push	changing the value of a	changing the value of a or b
pull	push	retrieving the value of b	changing (retrieving) the value of b
push	pull	changing the value of a	retrieving the value of c
pull	pull	retrieving the value of b	retrieving the value of c

Figure 7 shows both the propagation and trigger direction among the three variables. Note that a is a normal variable rather than a signal no matter whether we declare it with push/pull or not—since push and pull are used to describe the trigger between the declared variable and the variables involved in its evaluation at the right-hand side, a declaration without any involved variable will fall back to a normal variable declaration. Table 1 shows the four possible combination if we consider using push or pull for b and c in Figure 7. The cases of using push or pull for both b and c extend the propagation of value change based on their evaluation strategies, respectively. Since the two modifiers are used to specify the edges rather than the nodes themselves, it is possible to pull values from signals declared with push; source nodes will always hold the latest values and wait for sink node's pulling. On the contrary, asking signals declared with pull to push their values as shown in Figure 7 is also possible. In that case, the value change of source nodes will not be pushed to sink node until their values are retrieved. However, applying push and pull to individual source nodes at the right-hand side of a signal declaration is not allowed.

There is also a risk of propagation loop in this proposal as in other reactive designs and spreadsheet programs. When the values of signals depend on each other, the value changes will be propagated endlessly. We can regard the problem as detecting cycle in a directed graph, where vertices are signals and edges are push/pull arrows. Since signals are specified by declaration, it is possible to detect cycles with DFS algorithms such as Tarjan's algorithm [16] at compile-time and ask programmers to fix. However, the cost of detection depends on the scale of programs and the strategy of module loading in the language. It might be realistic only if the signal networks are closed within the same module; allowing using signals among modules might make compilation complicated. This is also one of the reasons why we limit the declaration to fields in our prototype implementation.

4. PuPPy

This idea we proposed in Section 3 can be applied on variables and functions, but we limit the application to fields and methods to make the design clearer and simpler in an imperative object-oriented language. We implemented PuPPy as an extension to the Python language to show the feasibility of our idea. Fields on objects can be declared with the push and pull statements, which are normal declaration with the modifiers prepended. Note that the reason why we implemented it on Python rather than on our previous implementation, ReactiveDominoJ [6], is to show the proposed mechanism can be directly built on top of Python.

4.1. A quick overview

We implemented PuPPy as an extension to the Python [7] language to demonstrate our idea. PuPPy adds two new keywords to Python, `pupush` and `pupull`, to support the push-pull automation mechanism for objects. They stand for "PuPPy push" and "PuPPy pull", respectively. Here we chose them rather than just `push` and `pull` since the terms `push` and `pull` might be too popular to be reserved keywords—they are frequently used in the implementation of data structure such as stacks and queues[1]. Since there is no explicit field declaration in Python, we simply add two kinds of statements that can be used to define fields in a class:

('pupush' | 'pupull') ⟨*field assignment*⟩

⟨*field assignment*⟩ is a normal statement in Python for defining fields in a class. For example, defining a field dis with pupush in the definition of class Point, which represents a point on an orthogonal coordinate plane:

```
class Point:
      :
  x = 0
  y = 0
  pupush dis = math.sqrt(x ** 2 + y ** 2)
      :
```

The field dis is the distance from the origin. Although here the values of x and y are 0 at the timing of defining dis, the definition of dis is similar to a signal assignment. Whenever the value of x or y is set, the value of dis will be updated automatically. Note that x and y are fields in the same class; the fields on other objects are not taken into account. In some sense, it helps programmers to prevent the definition of dis from being evaluated immediately in imperative programming. Note that if there are two occurrences of assignment for the same field in a class, the definition will be updated after the later assignment is interpreted according to the semantics of Python. However, in PuPPy this case is ignored to simplify the implementation. We assume that in a usual object-oriented design such a case is rarely used. Similarly, pupull can be used to define the field dis. The only difference in looks is the keyword prepended to the definition of dis, but their evaluation strategies are different. In the case of pupull, the value of dis will only be updated when it is really fetched somewhere, i.e. when the value of dis is read in the evaluation of an expression. For example, when a statement that prints out the value of dis:

```
print(self.dis)
```

[1] In fact, we also encountered compilation problem in modifying CPython implementation if we simply use push and pull.

is executed, the expression assigned to dis will be evaluated again to return the latest value. In this example, the pull statement looks much meaningful since usually we do not want to evaluate dis twice for a movement: one for x and one for y. With the pull statement, we can always get the latest value of dis but avoid evaluating it several times. Although the push statement looks like useless in this example, it still plays an important role in imperative programming. Since we often use side effects in methods, there might be some statements that must be executed for the value changes. Suppose that in the expression assigned to the field, some side effects such as logging should not be skipped, then we have to use pupush instead of pupull to ensure every propagation will be done gradually and steadily.

4.2. The push and pull statements

The push statement is a statement that defines fields in a class, just like other assignments used to define class members. Writing a push statement outside of class definition is not allowed. An expression can be declared with pupush in the class definition to let the expression be reevaluated every time when the fields read in it are set. The syntax of the push statement is shown as follows:

'pupush' ⟨*field*⟩ '=' ⟨*expression*⟩

It begins with the keyword 'pupush' and looks like normal assignments. ⟨*field*⟩ is the field to be declared in the enclosing class, and ⟨*expression*⟩ is the expression assigned to the field. Unlike a normal assignment, what we assign to the field is the expression itself rather than the evaluation result of the expression. Although in imperative programming it is not possible to really postpone the evaluation, the push statement lets the expression be reevaluated every time when the fields read in it are set. In other words, it is not really a lazy evaluation but forces the expression to be reevaluated when we expect the result of reevaluation might be different. *A handler for executing the assignment again is created and bound to the setting events of the fields used in the evaluation.* The setting event for a field refers to the event which will be triggered when the value of the field is set. PuPPy only infers the fields at the right-hand side that are declared in the enclosing class. It means the scope of propagating value changes for a push/pull satement is lexically defined.

Alternatively, a field can be defined with pupull in the class definition to ensure the field will be up-to-date when it is used. The syntax of the pull statement is similar to the one of the push statement, but it begins with a different keyword:

'pupull' ⟨*field*⟩ '=' ⟨*expression*⟩

It also creates a handler for executing the assignment again, but binds the handler to the getting event of this field. The getting events for a field is the event that will be triggered when the value of the field is got. It means that the expression assigned to the field will be reevaluated only when the value of this field is fetched. Thus, any of the setting events for the fields used in the expression will not cause the reevaluation. Even though any change that might result in a different value for the evaluation of this expression is made, the handler will not be executed.

4.3. The implementation of PuPPy

The implementation of PuPPy[2] is based on CPython [17], the reference implementation of the Python programming language. It is an interpreter written in C. We added the push and pull statements to Python's grammar to support the push-pull automation mechanism.

Our approach to implementing PuPPy is basically a source-to-source transformation. PuPPy interpreter extends CPython to accept the push and pull statements, transforms them to plain Python code, and leaves code generation to CPython. The transformation is based on the idea of getter and setter in object-oriented design, which benefits from a built-in function named the property function [18] and the decorators proposed in PEP (Python Enhancement Proposal) 318 [19]. PuPPy transforms the code of class definition in which the push statement or the pull statement is used. The read and write access to

[2]It is available on our project page: http://psl.csie.ncu.edu.tw/puppy.

these fields are properly replaced with the access to the corresponding getters and setters.

5. Evaluation

In this section, we use several metrics to evaluate our prototype implementation. We conduct preliminary microbenchmarks to observe the performance overheads of transformed code. In order to reveal the benefits and drawbacks of using PuPPy, we then analyze typical usage of signals and events in several related designs for comparison. Note that we compare with signal class libraries and event systems rather than functional-reactive design since PuPPy is targeted at bringing signals to imperative object-oriented design.

5.1. Preliminary microbenchmarks

We show the result of conducting several kinds of preliminary microbenchmarks for measuring the performance overheads caused by our PuPPy extension. Note that there might be different approaches to implementing PuPPy and what we measured here is the performance of our prototype implementation. The version of CPython where we applied our PuPPy implementation is Python 3.7.0a2+[3], and configured with `--enable-optimizations` option. We ran all the preliminary microbenchmarks on both two platforms.

First, to ensure PuPPy does not affect the performance of the code without any push or pull statement, we prepare a class named `Point` containing the following assignment:

```
dis = math.sqrt(x ** 2 + y ** 2)
```

where `x` and `y` are the other two fields in the class. We measured the performance of accessing the field `dis` in an object of `Point` by the `timeit` method in `timeit` module:

```
timeit.timeit('p.dis', setup='from plane import Point; p=Point(3, 4)', number=1000000)
```

[3]The CPython implementation is on https://github.com/python/cpython.git, commit `4eaf7f949069882e385f2297c9e70031caf9144c`.

Table 2: The execution time of (i) getting the value in a field and (ii) calling a method.

	(i) getting the value in a field				**(ii) calling a method**		
	CPython	PuPPy	push	pull	event	push	pull
Platform A[a]	40.56	40.26	39.96	763.73	653.94	537.89	172.31
Platform B[b]	36.49	36.43	35.07	742.83	602.14	493.39	154.12

in milliseconds

[a]macOS High Sierra 10.13.4 with Intel Core i5 2.9GHz
[b]CentOS 6.9 with Intel Xeon E5-2620 v4 2.1GHz * 2

where the plane is the module containing the Point class. The function timeit is a simple way provided by the standard library in Python to time small bits of code, which avoids a number of common traps for measuring execution time [20]. It instantiates an object of Point and then gets its dis one million times. The results of running with the plain CPython interpreter and our PuPPy interpreter are shown in Table 2(i), marked with CPython and PuPPy, respectively. The result shows that on both platforms the two interpreters have same performance since our PuPPy interpreter does nothing for the code without the push and pull statements.

The next scenario we want to evaluate is the misuse of the push or pull declaration. When programmers use the push and pull statements in an unnecessary scenario, it might cause unexpected overheads of performance. Our PuPPy extension creates property attributes according to the statements pupush and pupull, even though the fields are never used as signals. For example, in the Point class mentioned above we insert pupull in front of the assignment of dis, but actually the values of the fields x and y never change. In this case, just retrieving dis as a normal field should be required, but the transformation introduces the overheads of accessing property attributes and reevaluating the expression. On the other hand, using pupush in this case also results in unnecessary transformation, but retrieving dis should cause no performance overheads. The result of running them by our PuPPy interpreter are respectively marked with push and pull in Table 2(i), which shows that the cost of running the pull statement is about twenty times higher than the push statement.

The third one is to compare with using events in the same scenario. To see the overheads caused by making a field up-to-date, we use plain Python to implement the most basic event functionality in C# and measure the execution time of updating a field[4]. We write a Circle class with such a naive event mechanism, in which a field named area will be automatically calculated when the other field r is set. To see the overheads due to making area up-to-date, we instantiate an object of Circle and measure the execution time of setR:

```
timeit.timeit('c.setR(6)', setup='from use_evt import Circle; c=Circle(5)', number=1000000)
```

where the use_evt is the module that contains the Circle class. The same behavior can be rewritten in PuPPy by simply defining the field area with pupush. The execution time of using events and the push statement are shown in Table 2(ii), marked with event and push, respectively. It is not surprising that the push version ran faster than the event version since it does not need to instantiate an event and iterate its handlers. The pull version is listed for comparison; it actually does nothing else but sets the value.

5.2. Compare with signal class libraries

In order to discuss the difference between PuPPy and signal class libraries, we list the essential elements and operations that are necessary for supporting signals, and then use them to compare PuPPy with three different kinds of signal class libraries.

5.2.1. Essential elements and operations for signals

As we discussed in Section 2, the elements in a typical signal class library include at least signals and normal variables. In addition, some signal class libraries introduce yet another special kind of variables between signals and

[4]How the simple event system is implemented and other details of the preliminary microbenchmarks can be found in the downloads on our project page: http://psl.csie.ncu.edu.tw/puppy#downloads.

Table 3: The construction and the conversion of the essential elements for supporting signals.

elements	construction	to signals	to source signals	to normal variables
signals	signal assignment	—	—	lifting down
source signals	source construction	—	—	lifting down
normal variables	variable definition	lifting up	lifting up	—

normal variables. Such a special kind of variables are similar to signals, but they are only used for holding values rather than expressions. They are used in the expression of a signal assignment to explicitly specify source nodes in the propagation. Unlike normal variables, they will not be evaluated to values immediately in signal assignments. For example, in Scala.React [4] and REScala they are created by Val and Var, respectively. In order to clarify the discussion, below we use source signals to refer to such a special kind of variables since they can only be used as source nodes in the propagation. On the other hand, normal signals can be used as both source nodes and sink nodes. It means that normal signals can be assigned an expression, and can also be used in the expression of another signal assignment.

As to the operations, we can consider the construction of these essential elements and the conversion among them as shown in Table 3. Note that the API functions for handling signals are not included in this table since here only the elements and conversion at language-level are listed. However, as one of the advantages of the class library approach, providing a rich set of functions at library-level for composing signals is also an important point; we leave the discussion to the next section. Note that signal class libraries need a way that is different from the signal assignment to construct source signals since source signals cannot be assigned an expression for being sink nodes. Also, signal class libraries may allow programmers to change the value inside source signals or not, depending on their design. As to the conversion among these essential elements, since source signals are kinds of signals, there are no difference in lifting up/down signals and source signals to normal variables.

REScala [11] supports signals with a dedicated class[5] named Signal. Furthermore, an additional class Var is given for explicitly specifying source nodes in signal assignment. On the contrary, PuPPy implicitly infers source nodes in a push/pull statement, and does not give a special kind of variables. As a consequence, in PuPPy the effective scope of a signal is limited to the enclosing class, while in REScala it is determined by the scope of the special kind of variables.

Flapjax [9] also provides a class library for signals, but is slightly different from REScala. The signals supported in Flapjax is called Behavior, following the naming in Fran[5]. Unlike REScala, Flapjax does not provide source signals and supports only signals and normal variables. Therefore, programmers cannot construct a signal whose value can be manually altered as in REScala. A signal can be constructed by calling functions given by the library, for example to construct a signal from the property of an HTML element by extractValueB. It can be considered as lifting up a normal value. In Flapjax, all expressions whose values depends on a signal also become signals, so programmers can further construct others by using signals in the assignments. On the other hand, programmers can use functions like insertValueB to lift down a signal to the property of an HTML element. Basically, all signals are implicitly used along with normal variables in the world of Flapjax, but they need explicit lifting up/down in order to interact with the outside, which can be done by calling functions given by the library. On the other hand, PuPPy even does not support any abstraction for signals and events, and only focuses on the propagation of value change in fields of a class.

SignalJ [15] is a simple extension of Java, which integrates signals with events by regarding events as signal updates. In SignalJ, variables can be declared as signals by prepending the signal modifier, and the usage of signals are the same as normal variables. Although here we discuss SignalJ along with other class libraries, there is no need to explicitly construct source signals and lift

[5]Strictly speaking, it is implemented with traits in Scala, but here we simply use the term "class" for the abstraction of class-level composition.

Table 4: The comparison of the four extensions for supporting signals.

	REScala	**Flapjax**	**SignalJ**	**PuPPy**
signal assignment	Signal{}	normal assignment	signal	pupush/pupull
source construction	Var()	—	—	—
lifting up	Var()	extractValueB(), etc.	—	—
lifting down	now	insertValueB(), etc.	—	—

up/down between variables and signals. The annotation approach in SignalJ hides the class library from programmers, but allows signals to accept some specific operators as invoking methods on objects, for example the subscribe operator. Every variable declared with signal are signals, which pulls value changes from its source signals in the assignment. The value change of a signal is considered as an event, and allowing to bind handlers with the subscribe operator for providing side effects. Both SignalJ and PuPPy use modifiers to distinguish signals from normal variables, but the way of event handling is different. The handlers in PuPPy are automatically created according to the assignment, while the ones in SignalJ can be arbitrarily specified with operators. As to the operators for event composition in SignalJ, we discuss along with REScala in the next subsection.

In PuPPy source nodes in a signal assignment are inferred automatically, so that there is no need to specify with something like Var. However, since PuPPy uses the modifiers to specify signal assignments, programmers need to explicitly add pupush or pupull at the beginning of the assignment. The comparison of the four extensions for supporting signals is summarized as Table 4. In PuPPy there are only normal variables, and it is the modifiers to let some of them work as signals in a specific scope. The benefit is that programmers can use normal variables as signals without any lifting up and lifting down. This avoids introducing types for signals and simplifies the language semantics [15] as in SignalJ. On the contrary, as a result of the lack of a dedicated abstraction for signals, the signals in PuPPy cannot be passed to another objects. PuPPy is close to SignalJ on the design of modifiers, but PuPPy supports push-based evaluation in addition to pull-based evaluation to distinguish between events

```
1 val s = Var[Int](0)
2 val s_MAP: Signal[String] = s map ((x: Int) => x.toString)
3 val o1 = s_MAP observe ((x: String) => println(s"Here: $x"))
4
5 val e = Evt[Int]()
6 val e_MAP: Event[String] = e map ((x: Int) => x.toString)
7 val o1 = e_MAP observe ((x: String) => println(s"Here: $x"))
```

(a)

```
1 import puppy as p
2 s = 0
3 pupush s_MAP = p.map(s, lambda x : str(x))
4 pupush o1 = p.map(s_MAP, lambda x : print(f"Here: {x}"))
```

(b)

Figure 8: The map example in REScala (a) and in PuPPy (b).

and signals. On the other hand, SignalJ allows to construct signal networks in both static and dynamic ways, but PuPPy cannot provide dynamic construction due to its simple semantics and limited scope.

5.2.2. Operators for composing signals

The operators for using signals play an important role in signal class libraries. REScala provides a rich set of combinators such as map and fold for composing signals and events, while SignalJ provides operators like subscribe for registering handlers to the events occurring on signals. Here we show how we can implement these operators in PuPPy. As shown in Line 2 of Figure 8(a), map function in REScala applies a function to the value carried by the signal s to generate another signal s_MAP. Then in Line 3 the operator observe is used to attach a handler; Line 5–7 show the same operation on events. In PuPPy, we can simply implement such a map function in a module:

```
def map(v, f):
    return f(v)
```

Then importing that module, here puppy, and using it along with the lambda expression in Python as shown in Line 2 of Figure 8(b). For such a usage of the observe operator, which is similar to registering a handler to the events occurring

```
val e = Evt[Int]()
val f = (x:Int,y:Int)=>(x+y)
val s: Signal[Int] = e.fold(10)(f)
```

(a)

```
e = 0
p = puppy.PuPPy(10)
pupush s = p.fold(e, lambda x, y : x + y)
```

(b)

Figure 9: The fold example in REScala (a) and in PuPPy (b).

Table 5: Rewriting the examples of REScala combinators with PuPPy.

combinators	from	to	rewritable?	accumulation?
latest	event	signal	yes	—
changed	signal	event	yes	—
map	signal/event	signal/event	yes	no
fold	event	signal	yes	yes
or	event	event	no	—
and	event	event	no	—
count	event	signal	yes	yes
last(n)	event	signal	yes	yes
list	event	signal	yes	yes
latestOption	event	signal	—	—
fold Match	event	event	yes	no
iterate	event	signal	yes	yes
change	signal	event	yes	yes
changedTo	signal	event	no	—
flatten	signal	event	no	—

on signals by the subscribe operator in SignalJ, we can use map function instead. 580 Figure 9(a) shows the usage of fold function in REScala, which creates a signal by folding events with a given function. In Line 3, a function f that sums up the values associated to the event is given along with an initial value 10. Since PuPPy does not implement signals with classes, we need an accumulator to store the previous value of the signal s. We can implement a class to hold the 585 acc field and the fold function:

```
def fold(self, v, f):
    self.acc = f(self.acc, v)
    return self.acc
```

As shown in Figure 9(b), an object instance p of such a class for accumulation 590 is instantiated with the initial value 10 and used to fold the values in e to s. Table 5 lists the combinators explained in REScala manual [11]. For every combinator, we write down the elements it converts between in from and to, and mark yes in rewritable? if the example of which can be rewritten in PuPPy[6]. The accumulation? indicates whether the function needs an accumulator or not; if yes, we need an object to hold the accumulated value as what we showed for the fold combinator; otherwise, its implementation can be a global function as the map combinator. Note that latest and changed are just assigning values between a field and another field declared with pupush since PuPPy unifies signals and events; therefore, no function is necessary. For others, basically we can rewrite them by storing its past value or accumulated value in an object and applying a lambda expression for conversion as in REScala. However, generally speaking, the way of associating the object for holding past values with a signal itself in REScala is better since it hides the objects from programmers. For the combinators or and flatten, similarly the or operator in SignalJ, which detect the occurrences of one among multiple events, we failed to rewrite since in PuPPy it is not able to exactly know which source signal triggers and get the value carried by it. For example, in our version of or combinator[7]:

```
pupush e1_OR_e2 = p.e_or(e1, e2)
```

here we expect e1_OR_e2 must be set to the value of e1 or e2, depending on which one is triggered. However, PuPPy provides no mechanism to exactly know which one is triggered. A workaround is to hold the previous values of e1 and e2 in the object instance p and returning the one that is different from its previous value, but it does not work when e1 or e2 is triggered again with the same value—we have no idea whether the value of e1 or the value of e2 should be returned. For the combinators such as and and changedTo in REScala and the when operator in SignalJ that take a predicate to decide triggering or not, we cannot rewrite them in PuPPy either since the sink signal will always be reevaluated no matter whether the predicate is true or not. In the following

[6]The code examples of implementing these combinators with PuPPy are available on the downloads of our project page: http://psl.csie.ncu.edu.tw/puppy#downloads.

[7]Since or/and are reserved keywords in Python, we use the names e_or/e_and instead.

example,

```
pupush e_AND = p.e_and(e, lambda x : x > 10)
```

whenever e is triggered, the value of e_AND will be set even though x is not greater than 10. We can store the value of x in the object instance p when the predicate is true, and use it to set e_AND again when the predicate is false. It might not be a significant problem except that it will wastefully calculate. However, we cannot stop executing the handlers bound to e_AND unless PuPPy allows to cancel event triggering. As to latestOption, it is a variant of latest function for the Option in Scala, so we ignore the implementation in Python.

Another interesting difference is that in PuPPy we cannot assign values to other fields except the sink signal since the lambda expression in Python does not allow to use assignments as a side effect[8]. For example, in REScala we can assign x to test:

```
val f = (x:Int)=>{test=x; x+1}
val s: Signal[Int] = e.iterate(10)(f)
```

where iterate returns a signal that holds the value computed by f on the occurrence of the event e, and assigning the value to test in f can be regarded as kinds of side effects. To rewrite it in PuPPy, we have to move the assignment to another lambda expression and apply to test separately:

```
pupush test = p.iterate(e, lambda x : x)
pupush s = p.iterate(e, lambda x : x + 1)
```

In other words, we can only set the value inside the sink signal in the functions given to combinators or observe. Furthermore, even if the value of a source signal is not used in the lambda expression, we still have to leave it in the right-hand side. For example, the following line in REScala:

```
e observe { _ => println("hello") }
```

should be rewritten to:

```
pupush s = p.map(e, lambda x : print("hello"))
```

[8]Python is going to support assigning to variables within an expression[21], then it might be possible to provide side effects inside lambda expressions.

Unless e is specified at the right-hand side, PuPPy has no idea how to trigger this reevaluation. Note that here map is a global function; in the case that observe needs an accumulator, we have to use map on an object instance instead.

5.3. Compare with pure event systems

For a pure event system, undoubtedly there are two essential elements: events and handlers, so the system must support operations on defining events and handlers. Furthermore, it needs to provide mechanisms for binding handlers to events and triggering events. Note that events are mainly defined for being triggered rather than holding values. In the design of most event systems, events can only temporarily carry values when they are triggered; they cannot be evaluated as normal variables.

EScala is based on Scala and provides rich event support, and we have shown how to construct events by evt and bind normal methods as its handlers in Section 2. Due to the lack of dedicated support for specify setting events, in order to propagate the value change of a field declared with var[9] in Scala, we have to prepare a setter method and declare an event on that method by afterExec. We may also declare ea as an imperative event for being triggered by calls. However, neither of the ways to declare the event ea is as implicit as signals. Another interesting point is that events do not hold values. Although events are declared with a type, it is used to define the type of the value used in event trigger. We cannot directly operate on the value of an event since it does not carry the value until being triggered.

C# supports imperative events and handlers by the keyword event and delegate, respectively. To implement a signal-like field, we can use delegate to define the type of handlers and then use event to declare events along with the handler type. As to the binding, it can be done by operators as in EScala. However, unlike in EScala, the event trigger in C# can only be imperative. A notable

[9]Note that in Scala var is for mutable variables while val is for immutable ones. Unlike in REScala we assign an object to a val, here we choose var to change the value in it later.

Table 6: The difference between ReactiveDominoJ and PuPPy.

	ReactiveDominoJ	**PuPPy**
direction of trigger	one-way (push)	two-way (push and pull)
how to enable automation	dynamic	declarative
scope of specifying automation	multiple statements	single statement
scope of inferring fields	cross objects	within objects

difference is that C# provides the ?. operator to conveniently ensure that we do not want to trigger the event without any handler bound to it. Using events in C# to implement signal-like field has the same drawbacks as using the ones in EScala.

On the other hand, in PuPPy, programmers can simply prepend the keyword `pupush` before the assignment to let a field be updated when the value it depends is changed. Unlike other pure event systems, there is no need to explicitly define the setting event and the handler. It also means that programmers do not have to worry about the relation between the field, the event, the handler, and the binding. Moreover, the type in PuPPy is hidden from programmers. As a consequence, there is no way to arbitrarily define events and bind handlers; event handling is implicitly processed for propagating value changes of fields only. For this reason, in PuPPy there is no additional expression such as `e()` and `e?.Invoke()` to trigger events except updating values in fields.

5.4. Compare with ReactiveDominoJ

As discussed in Section 3, the original automation mechanism can be mapped to the push declaration in the push-pull automation mechanism. Since PuPPy supports the push-pull automation, it provides the pull statement that is not available in ReactiveDominoJ, the implementation for the original automation mechanism. Furthermore, how to enable the automation is quite different. ReactiveDominoJ is more dynamic and flexible, while PuPPy is much simple and declarative; PuPPy even hides events and handlers from programmers. In addition, the scope of specifying the automation and applying the inference is also different. These differences are summarized in Table 6.

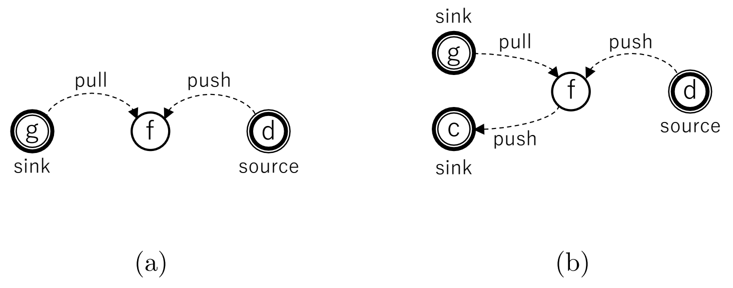

Figure 10: Two possible scenarios of mixing push and pull.

5.5. The advantage of hybrid push/pull design

A hybrid push/pull design lets programmers to choose between push and pull to fit their needs. Suppose that we have a device that periodically updates its value. For its GUI client, we can decide to update the graph immediately or allowing users to set the refresh interval. Furthermore, such a design also makes it possible to mix the usage of push and pull. As shown in Figure 10(a), we might hope a signal value from the device d can be steadily recorded by a file f, while the GUI client g can still refresh the graph at its own pace. It might be useful when we want to log every change from the device d in the file f, but avoiding frequently refreshing the drawing on g. Using push for both f and g will make every change in d propagated to g immediately. Another possible scenario is shown in Figure 10(b), where another client c needs to be continuously updated. In a system that supports only push or pull, it is not able to allow detailed configuration.

6. Related Work

The signals, also known as behaviors, play the major role in functional-reactive programming [5, 22, 23, 24, 25] and its abstraction heavily relies on functions. When porting the signal to the world of imperative object-oriented programming, library developers tend to wrap it up in a class as how they do for higher-order functions. Several projects are devoted to provide frameworks [8, 9, 26, 11, 27, 28] that support using functional-reactive programming along with imperative object-oriented design. Usually they develop a complete and powerful framework that supports signals along with event streams and gives a lot of functions for conversion between signals, event streams, and normal variables. Although programmers need to introduce a whole framework to benefit from signals, they can quickly enter the world of functional-reactive programming and start using signals with library function calls. On the other hand, several research such as active expressions [29] provide a mechanism to build signals by writing expressions. In both two designs, the boundary between time-varying values and fixed values is very obvious. The class for signals holds time-varying values, while normal variables store fixed values. Function calls and signal construction are the boundary between signals and normal variables. Although we can directly give signals to functions provided in the framework, we need to lift signals down to normal variables before giving to other functions. It might be helpful to the design of a complicated system, while being too heavy for a small system.

On the other hand, several research activities are devoted to developing a dedicated abstraction or even a new language for functional-reactive programming, for example FrTime [30] and KScript [31]. They construct the language with better support for functional-reactive programming, and make programming further declarative and seamless. Since these languages directly support functional-reactive programming, the semantics of them are based on functional programming. To an existing system with imperative object-oriented design, it is too expensive to migrate. SuperGlue [32] is one of the approaches to integrating signals with imperative object-oriented design. Such design limits the usage of signals to a smaller scope, for example being the properties of components. It lets programmers to implement an imperative object-oriented design but still benefit from signals. However, it can also be classified as the approach of proposing a new language.

The push and pull model has been mentioned by Elliott [14] to discuss data-driven and demand-driven evaluation in the implementation in the context of functional-reactive programming. It is also discussed in the event-based system such as ReactiveX [33] in the context of imperative object-oriented programming. It turns out that the two styles are useful in different scenarios and the performance might be affected by their usage [34, 15]. There are also research activities discussing distributed states handling [35]. By comparing the discussion in functional-reactive programming and imperative object-oriented programming, we can know that the push style is the basics of events while the pull style has the flavor of signals. We discussed the push and pull styles on our previous research result [6], and extended the automation mechanism to support both the two styles.

Since side effects are frequently used in imperative programming, how to limit the scope of read/write access is important. There are a lot of discussion on how to make programmers' intention clear and how to improve the encapsulation, and various support such as design patterns, class libraries, and language constructs are developed. The Observer pattern, one of the most famous patterns [1], can loose the coupling between objects. However, it is difficult to group pattern code and ensure the relation between the fields and methods in patterns. Several researchers suggested replacing them with dedicated abstraction [4, 36] or extracting them to one place for reuse [3]. The signals and slots [37] supported in Qt's meta-object system can also be regarded as an approach to eliminating code for implementing patterns. The getter-setter design is another example, which is encouraged in the domain of imperative object-oriented programming and has almost become a guideline of development in industry. However, it results in a lot of annoying code, and how to ensure the consistency among the field, the getter, and the setter is an issue. The property in C# is a solution to clearly declarative getters and setters. On the other hand, recently there are several imperative programming languages supporting functions with a library class [38] or a dedicated abstraction [39] in order to provide the flavor of functional programming. They let programmers to limit the scope of side effects and clarify their intention. These support help programmers to announce their intention declaratively, and thus undesired writing can be prevented. The push-pull automation mechanism might also be classified under such kind of support to imperative programming.

7. Conclusion

Both events and signals can be regarded as a mechanism to propagate value changes of variables; events are explicit while signals are implicit. We pointed out that both a series of events and signals can be considered as a process of propagating value changes from *source* to *sink*. Although the direction of propagation is fixed, the direction of trigger can be either *push* or *pull*. We analyzed why in existing systems it is necessary to distinguish between signals, source signals, and normal variables and then proposed a lightweight push-pull automation mechanism that helps programmers to implicitly use signals. The intention of using signals can be clarified by declaring with *push* or *pull*. The push-pull automation can be applied on variables and functions, but we focused on object-oriented design in our prototype implementation, *PuPPy*. As an extension to the Python language, *PuPPy* provides a solution to simply use variables as signals. We ran preliminary microbenchmarks for our prototype implementation, and the result shows that this extension does not impose significant overheads on the runtime. We discussed the essential elements and operations in signal class libraries and pure event systems, and used them to compare *PuPPy* with several representative designs. It shows that PuPPy is lightweight and has a simpler design devoted to value changes.

Acknowledgments

This research is supported by the Ministry of Science and Technology (Taiwan) as part of the project "Providing better abstraction and modular support at language-level for applications in scientific computing domain" [MOST 105-2218-E-008-016-MY2].

References

[1] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns, Addison-Wesley, 1994.

[2] E. W. Dijkstra, Selected Writings on Computing: A Personal Perspective, Springer-Verlag New York, Inc., New York, NY, USA, 1982.

[3] J. Hannemann, G. Kiczales, Design pattern implementation in Java and AspectJ, in: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA'02, ACM, New York, NY, USA, 2002, pp. 161–173. doi:http://doi.acm.org/10.1145/582419.582436.

[4] I. Maier, M. Odersky, Deprecating the Observer Pattern with Scala.react, Tech. rep. (2012).

[5] C. Elliott, P. Hudak, Functional reactive animation, ICFP'97, ACM, 1997, pp. 263–273. doi:10.1145/258948.258973.

[6] Y. Zhuang, S. Chiba, Expanding event systems to support signals by enabling the automation of handler bindings, Journal of Information Processing 24 (4) (2016) 620–634. doi:10.2197/ipsjjip.24.620.

[7] Python Software Foundation, The python programming language, https://www.python.org/.

[8] A. Courtney, Frappé: Functional reactive programming in Java, PADL'01, Springer-Verlag, 2001, pp. 29–44.

[9] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, S. Krishnamurthi, Flapjax: a programming language for Ajax applications, OOPSLA'09, ACM, 2009, pp. 1–20. doi:10.1145/1640089.1640091.

[10] G. Salvaneschi, G. Hintz, M. Mezini, REScala: Bridging between object-oriented and functional style in reactive applications, Modularity'14, ACM Press, 2014.

[11] Software Technology Group, TU Darmstadt, Rescala manual, http://www.rescala-lang.com/manual/.

[12] V. Gasiunas, L. Satabin, M. Mezini, A. Núñez, J. Noyé, EScala: modular event-driven object interactions in Scala, AOSD'11, ACM, 2011, pp. 227–240. `doi:http://doi.acm.org/10.1145/1960275.1960303`.

[13] Y. Zhuang, S. Chiba, Enabling the automation of handler bindings in event-driven programming, in: 2015 IEEE 39th Annual Computer Software and Applications Conference, Vol. 2, 2015, pp. 137–146. `doi:10.1109/COMPSAC.2015.48`.

[14] C. M. Elliott, Push-pull functional reactive programming, in: Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell, Haskell '09, ACM, New York, NY, USA, 2009, pp. 25–36. `doi:10.1145/1596638.1596643`.

[15] T. Kamina, T. Aotani, Harmonizing signals and events with a lightweight extension to java, Programming Journal 2 (3) (2018) 5. `doi:10.22152/programming-journal.org/2018/2/5`.

[16] R. Tarjan, Depth first search and linear graph algorithms, SIAM JOURNAL ON COMPUTING 1 (2).

[17] Python Software Foundation, CPython on GitHub, https://github.com/python/cpython.

[18] Python Software Foundation, Built-in Function — property(), https://docs.python.org/3/library/functions.html#property.

[19] Python Software Foundation, PEP 318 – Decorators for Functions and Methods, https://www.python.org/dev/peps/pep-0318/.

[20] Python Software Foundation, timeit — Measure execution time of small code snippets, https://docs.python.org/3.7/library/timeit.html.

[21] Python Software Foundation, PEP 572 – Assignment Expressions, https://www.python.org/dev/peps/pep-0572/.

[22] Z. Wan, P. Hudak, Functional reactive programming from first principles, PLDI'00, ACM, 2000, pp. 242–252. `doi:10.1145/349299.349331`.

[23] Z. Wan, W. Taha, P. Hudak, Real-time FRP, ICFP'01, ACM, 2001, pp. 146–156. `doi:10.1145/507635.507654`.

[24] H. Nilsson, A. Courtney, J. Peterson, Functional reactive programming, continued, Haskell'02, ACM, 2002, pp. 51–64. `doi:10.1145/581690.581695`.

[25] Z. Wan, W. Taha, P. Hudak, Event-driven FRP, PADL'02, Springer-Verlag, 2002, pp. 155–172.

[26] I. Maier, M. Odersky, Higher-order reactive programming with incremental lists, ECOOP'13, Springer-Verlag, 2013, pp. 707–731. `doi:10.1007/978-3-642-39038-8_29`.

[27] The Sodium Project, Sodium - Functional Reactive Programming (FRP) Library for multiple languages, https://github.com/SodiumFRP/sodium.

[28] Facebook Inc., React - A JavaScript library for building user interfaces, https://reactjs.org.

[29] R. H. Stefan Ramson, Active expressions: Basic building blocks for reactive programming, Programming Journal 1 (2) (2017) 12. `doi:10.22152/programming-journal.org/2017/1/12`.

[30] G. H. Cooper, S. Krishnamurthi, Embedding dynamic dataflow in a call-by-value language, ESOP'06, 2006, pp. 294–308.

[31] Y. Ohshima, A. Lunzer, B. Freudenberg, T. Kaehler, KScript and KSWorld: A time-aware and mostly declarative language and interactive GUI framework, Onward! '13, ACM, 2013, pp. 117–134. doi:10.1145/2509578.2509590.

[32] S. McDirmid, W. C. Hsieh, SuperGlue: component programming with object-oriented signals, ECOOP'06, Springer-Verlag, 2006, pp. 206–229. doi:10.1007/11785477_15.

[33] ReactiveX, An API for asynchronous programming with observable streams, http://reactivex.io.

[34] G. Salvaneschi, M. Mezini, Reactive behavior in object-oriented applications: an analysis and a research roadmap, AOSD'13, ACM, 2013, pp. 37–48. doi:10.1145/2451436.2451442.

[35] F. Myter, T. Coppieters, C. Scholliers, W. De Meuter, I now pronounce you reactive and consistent: Handling distributed and replicated state in reactive programming, in: Proceedings of the 3rd International Workshop on Reactive and Event-Based Languages and Systems, REBLS'16, ACM, New York, NY, USA, 2016, pp. 1–8. doi:10.1145/3001929.3001930.

[36] Y. Zhuang, S. Chiba, Method slots: supporting methods, events, and advices by a single language construct, AOSD'13, ACM, 2013, pp. 197–208. doi:10.1145/2451436.2451460.

[37] The Qt Project, Signals & Slots, http://doc.qt.io/qt-5/signalsandslots.html.

[38] Microsoft Corporation, Func(T, TResult) Delegate (System), https://msdn.microsoft.com/en-us/library/bb549151(v=vs.110).aspx.

[39] Oracle Corporation, OpenJDK: Project Lambda, http://openjdk.java.net/projects/lambda/.