Implicit Invocation of Traits

Thomas Pawlitzki
Lehrgebiet Programmiersysteme
Fakultät für Mathematik und Informatik
Fernuniversität in Hagen
D-58084 Hagen, Germany
Thomas.Pawlitzki@fernuni-hagen.de

Friedrich Steimann
Lehrgebiet Programmiersysteme
Fakultät für Mathematik und Informatik
Fernuniversität in Hagen
D-58084 Hagen, Germany
steimann@acm.org

ABSTRACT

We propose the introduction of a special kind of traits that implement methods implicitly invoked when an event of a given type occurs. Events are announced explicitly in the source code at their place of origin, and classes publishing events, as well as traits subscribing to them, are explicitly marked as such. The result is greater independence of publisher and subscriber (when compared to other implementations), as well as an explicit interface between the two. An implementation in Scala is briefly sketched.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*polymorphism, control structures*

General Terms

Languages, Design

Keywords

implicit invocation, traits, modularity

1. INTRODUCTION

Implicit invocation is a mechanism by which methods are called without being referenced at the call site [10, 14]. Implicit invocation is the underlying mechanism of publish/subscribe systems and event driven programming. In contemporary object-oriented programming languages, implicit invocation is usually realized using the Observer pattern [5]: it maps the implicit invocation, the event of which is being notified, to explicit invocations of notification methods in registered observers (or listeners, or subscribers). True implicit invocation, however, keeps the invoker completely ignorant of (and thus decoupled from) the invoked. More recently, aspect-oriented programming (AOP), AspectJ [1] in particular, has popularized a special form of implicit invocation, called implicit invocation with implicit announcement. It promotes the invocation of unnamed methods (called advices) before, after, and around the execution of certain statements, and this without the creation and dissemination of explicit events announcing the execution. While implicit announcement establishes coupling without explicit interfaces (see, e.g., [12] for a critical account of AOP and modularity), tying events to the execution of statements has some merits: it allows one to add behaviour the implementation of which depends on the concrete system composition. It is thus comparable to—yet significantly different from—the dynamic binding of object-oriented programming [12].

Traits [11] are partial specifications of objects (behaviour and sometimes also state) that can be mixed into classes. Traits are abstract in the sense that they cannot be instantiated—to obtain instances exhibiting a trait's behaviour, classes enhanced with traits must be instantiated. Deviating from interfaces, traits do not only specify behaviour, they also provide implementation. They are thus units of code reuse. In this paper, we suggest the combination of traits and implicit invocation so that behaviour specified in a trait is implicitly invoked by the occurrence of an event of a given type—rather than objects (or classes) subscribing to that event type, objects enhanced with a trait are automatically activated through one of the trait's methods being called, in response to the occurrence of a corresponding event. Our proposal of implicit invocation of traits (abbreviated as IIT) involves a special kind of classes, called events, and a special kind of traits, called handler traits. Events are explicitly announced; they can be parameterized with variables from the context in which they occur, and their occurrence can be associated with the execution of a block of statements. Analogous to [13], this gives us implicit invocation with parameter passing through clearly marked interfaces, promoting both independent development and code reuse.

The remainder of this paper is organized as follows. In Section 2, we motivate our approach by presenting a brief example of implicit invocation and its implementation based on the Observer pattern, contrasting it with our solution using IIT, and comparing this solution to related work. In Section 3, we present the language constructs involved in IIT and how they work together. Section 4 sketches a prototype implementation of IIT as a Scala library.

2. MOTIVATION

Implicit invocation is a popular architectural style for building loosely coupled software systems. Under implicit invocation, components communicate via events published by one component and consumed by others, its subscribers. One

SAC'10 March 22-26, 2010, Sierre, Switzerland.

```
1  interface Observer {
2    void update(Cell cell, Value value);
3  }
4  abstract class Cell {
5    Value value;
6    abstract void recalculate();
7    List<Observer> observers;
8    void notifyObservers() {
9      for(Observer o: observers) {
10       o.update(this, value);
11     }
12   }
13 }
14 class Expression { Value evaluate() {...} }
15 class Equation extends Cell {
16   Expression expression;
17   void recalculate() {
18     value = expression.evaluate();
19     notifyObservers();
20   }
21 }
22 class Table implements Observer {
23   void update(Cell cell, Value value) {...}
24 }
```

Figure 1: Implementation using the Observer pattern in Java.

```
1  event class ValueChange { Value value; Cell cell; }
2  handler trait Update subscribes ValueChange {
3    abstract void refresh(Cell, Value);
4    subscribe (ValueChange vc) {
5      refresh(vc.cell, vc.value)
6    }
7  }
8  abstract class Cell {
9    Value value;
10 }
11
12
13
14 class Expression { Value evaluate() {...} }
15 class Equation extends Cell publishes ValueChange {
16   Expression expression;
17   void recalculate () {
18     value = expression.evaluate();
19     publish ValueChange(this,value);
20   }
21 }
22 class Table uses Update {
23   void refresh(Cell cell, Value value) {...}
24 }
```

Figure 3: Alternative implementation using event types and handler traits.

key feature of implicit invocation is that publishers remain ignorant of their subscribers [6], thus allowing system extension unanticipated at the publishers' design time. Implicit invocation has originally been used as an extension mechanism for integrated development environments [10], and later been generalized to interactive systems, distributed systems, real-time monitoring, etc.[2].

2.1 Example

To motivate our approach, we use a simple spreadsheet example. A spreadsheet consists of a set of cells that can contain values, equations, etc. The contents of the cells are displayed in various forms, namely tables, diagrams, etc. When the content of a cell changes, the displays may need to be updated to reflect the change. An implementation of this simple mechanism using the Observer pattern is shown in Fig.1; note that in this implementation the invocation of the observers is implicit where the event occurs (the calling of `notifyObservers()` in line 16), but explicit in `notifyObservers()` (the calling of `update(.,.)` in line 7 on the particular observers).

Using the Observer pattern, the implementation of the example not only involves considerable boilerplate code for registration and notification (which is not included in Fig.1), it also establishes an explicit dependency of the class `Cell` and its descendants (the publishers) on their subscribers (here abstracted through an interface `Observer`; see Figure 2). The event itself is coded as a (synchronous) method call with parameters to the event implemented as parameters to the method. What we would rather see (and what we suggest in this paper) is a reification of events as instances of event types, and a subscription to event types rather than to publishers, so that instead of publishers and subscribers depending on (abstractions of) each other, they both depend on event types as interfaces between the two. Furthermore (and borrowing from the aspects of AspectJ), we would like to allow the grouping of handlers of different event types in one logical unit of reuse, called a handler trait. To see how this should work, we have rewritten the example of Fig.1 in Fig.3, using an ad hoc extension of Java's syntax (whose semantics should be intuitive enough for the time being, and which will be detailed in the following sections). Note that there is no explicit invocation of the handler trait—it occurs implicitly through the publishing of the event in line 12. The resulting dependencies are illustrated in Fig.4.

2.2 Related Work

Typed based publish/subscribe. The example in Fig.4 is strongly reminiscent of typed based publish/subscribe (TPS), as described in [3]. TPS uses the types of events primarily as an event filtering mechanism: subscribers subscribing to events of certain types will not be notified by events of other types. By contrast, we establish event types and declarations to publish and to subscribe them as explicit interfaces between publishers and subscribers whose adherence to can be checked by a compiler. Also, by assigning event handlers to traits, the handling of specific (groups of) events implemented in a trait can be reused across different classes, effectively introducing a behavioural layer between events and their consumers that is not available in TPS à la [3].

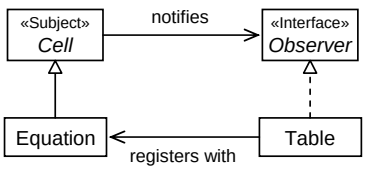

Figure 2: Dependencies between subject and observer established by the Observer pattern. (Notation ist UML)

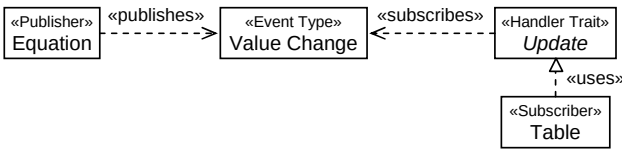

Figure 4: Dependency of publisher and subscriber on the event type, as established by IIT (Notation ist UML)

AspectJ. As mentioned in the introduction, the aspect oriented programming language AspectJ [1] comes with unnamed methods (called advices) that get invoked upon the execution of certain points in the program (called join points). As in IIT, advices can be grouped in behavioural units of code reuse (called aspects). However, because the designation of join points (the events) occurs implicitly in AspectJ (through so-called pointcuts), invocation is not only implicit, but also implicitly announced [15]. This means that for someone inspecting a join point at source level, it is not obvious whether its execution means an event and thus leads to the (implicit) invocation of advice (let alone which variables of the join point's context are passed as parameters). By contrast, we require events to be clearly marked in the code (by the instantiation of an event type) and all context variables to be passed as parameters be explicitly included in the event creation expression. Replacing the implicit announcement of AspectJ with an explicit one not only introduces a mutually decoupling interface between the implicit invoker and the invoked, it also allows us to dispense with the whole-program analysis required by AspectJ's compilation mechanism (called weaving), regaining separate development on the class level. Last but not least, since event handlers are eventually provided by classes, our approach is symmetric (both event sources and sinks are classes or, rather, objects), which AspectJ is not (all sinks are aspects).

Classpects. Classpects [9] is an approach to AOP that removes the asymmetry of AspectJ, by separating advice from its binding expressions and placing both in ordinary classes. Resulting is a language whose event handlers are plain methods (which are named and therefore can be overridden in subclasses), bound to runtime events (join points) by separate binding declarations. As in IIT, event handlers are executed on the object level (on instances of the hosting classes). However, classpects adopt the implicit announcement strategy of AspectJ (using pointcuts), inheriting the modularity problems that we want to avoid.

Ptolemy. Like IIT, Ptolemy [8] builds on explicitly announced events and uses event types to transport context information to the implicitly invoked event handlers. Event handlers are named methods that are bound to (sets of) event types using special binding declarations. Rather than copying context information into the fields of an event using a constructor (as we suggest for IIT), Ptolemy uses closures to gain access to the context of an event. On the other hand, Ptolemy does not support event subtyping as we do, therefore missing out an opportunity for further decoupling (handlers for a given type can accept events of subtypes; see Sections 3.1 and 3.3). Also, Ptolemy differs from our IIT in that it makes it possible that only some instances of a class act as observers (making handler methods useless ballast for unregistered objects), but these observe events from all subjects; whereas with our IIT, all instances of a subscribing class are automatically observers, but can choose individually which subjects they wish to observe (by setting a filter; see Section 3.4).

IIIA for Java. In our own prior work on Implicit Invocation with Implicit Announcement (IIIA) for Java, we have introduced join point types as types of events that can be both explicitly and implicitly announced [13]. Implicit announcement of events requires pointcuts that are defined within (and restricted in scope to) each publishing class, and which are interpreted as polymorphic type predicates (polymorphic because each class has to define its own pointcut, analogous to how each concrete class has to provide its own implementation of an inherited abstract method). Although theoretically, polymorphic pointcuts would allow separate compilation, using the pointcut implementation of AspectJ (and its associated weaving technology) does not, so that classes cannot be compiled modularly. Since two extensive case studies we conducted showed that implicit announcement (as enabled by pointcuts) was rarely advantageous over explicit announcement, we decided to dispense with implicit announcement altogether and replace aspects as keepers of event handlers with a construct that allows for separate compilation, such as traits. The result is the subject of this paper.

3. IMPLICIT INVOCATION OF TRAITS

As indicated by Fig.4, IIT involves handler traits and event types as new language constructs. The following explains how these are defined and integrated into a host language.

3.1 Event Types

Event types are types whose instances' sole purpose is to convey the occurrence of events parameterized by information from the context in which the events occurred. The creation of an event instance caused by a `publish` statement (line 12 in Fig.3) implicitly causes the invocation of all event handlers subscribing to this type of event (unless additional filtering criteria are provided; see below); it corresponds to the explicit announcement of the event.

Event types can be thought of (and implemented) as special kinds of classes with fields and implicitly defined constructors for setting the fields during instantiation. In a way, they are similar to Java's exception types; in particular, although they can have methods and their instances can be stored in variables just like all other objects, there is usually only little use in doing this—their main purpose is the signalling of an event, which is inherently transient.

Event types can have subtypes which are again event types; an instance of an event type is an (indirect) instance of all its super-types and substitutable for them. This is of particular interest for event handling, where a handler defined for an event type will accept instances of its subtypes. Event types inherit all properties from their supertypes.

3.2 Publishing Events

As indicated in the previous subsection (and in line 12 of Fig.3), to publish an event for IIT takes little more than instantiating the corresponding event type with the parameters from the context to be conveyed as part of the event—the only extra we require is that the type of events a class publishes is announced in the header of the class (using the `publishes` keyword). This is again analogous to Java's exception handling, which requires a `throws` clause in the header of a method throwing (checked) exceptions.

Inheriting from AspectJ [1] (and ultimately from CLOS), we allow events to be associated with the execution of (blocks of) statements, so that the event handler may choose to react before the execution, after it, or both (by providing corresponding handler methods; see below). Syntactically, we express this by attaching a block to the event type instantiation, as in

```
value = publish ValueChange(this, value) {
  expression.evaluate();
}
```

Note that the `publish` expression returns a value which is the value returned by the block. As far as publishing is concerned, standard event creation corresponds to event creation with an empty block.

3.3 Event Handling

While different classes likely react to events of the same kind differently, experience shows that there is usually also some boilerplate code associated with handling events of a given type[1], and also that events of different types are subscribed to by the same group of classes. Therefore, we tie event handling to traits that can be used by the classes showing interest in the events.

So-called handler traits are special traits with three predefined handler methods, named `subscribe`, `before`, and `after`. All three methods have a single parameter whose type must be an event type. If a handler trait subscribes to more than one event type, the handler methods are overloaded. In case of event subtyping, the implicit invocation of a handler method is bound to the method definition with the most specific event type (which, due to the single inheritance of event types, is always uniquely determined). Handler traits can have subtraits extending them; the rules of inheritance are basically those of the host language (with handler methods of same parameter type overriding one another). For reasons of symmetry, handler traits declare which event types they subscribe to in their headers, using the `subscribes` keyword. Event subscribing is inherited by the classes using the traits. While traits can provide complete event handlers, they cannot be instantiated. For this, they must be used by classes. A class using a handler trait inherits its handler methods (and everything else the trait defines); if a handler trait invokes abstract methods (as in line 18 of Fig.3) overridden in the class, the class is directly involved in handling the event. Treatment of multiply inherited handler methods with same parameter type is the same as in the host language.

3.4 Event Filtering

Until this point, event handling is defined completely on the type level: handler methods are defined in traits which are used by classes. The publishing of an event of a given event type will therefore implicitly invoke the handler methods on all instances of all classes using traits subscribing to the event type. This is in contrast to event publishing, which—although also declared on the type level—is tied to the execution of code and thus (unless the code is declared static) to (the context of) individual objects.

To tie the handling of events to single objects (as is inherently the case for implementations of the Observer pattern, and also for Classpects [9], EventJava [4] and Ptolemy [8]), handler traits are equipped with a filtering function which decides whether a given event instance should be accepted. The filter can be set for each instance of a subscribing class individually, by invoking, on the instance, a method `setFilter(.)` that takes a filtering expression (a function object). For instance, if a given display subscribing to `ValueChange` events is interested only in negative values, the statement

```
display.setFilter((ValueChange e) => {
  return e.value < 0;});
```

(using Scala syntax for defining function literals [7]) must be added. Filters also allow one to tie an observer to a single subject: for instance, if the subject is referred to by a variable `cell`, the statement

```
display.setFilter((ValueChange e) => {
  return e.cell == cell;});
```

makes sure that the instance referred to by `display` will be notified by `ValueChange` events originating from the cell referred to by `cell` only.

[1]Note that this boilerplate code cannot be attached to the event class, since this would mean that to get involved in event handling, subscribing classes either have to subclass the event class (which makes no sense) or need to register with it, leading to a re-introduction of the Observer pattern.

3.5 Event Chaining

The handling of an event can give rise to new events, and therefore a handler trait can publish events (including ones of the type that it handles; note once more the similarity to Java exception throwing and handling). The following gives an example of this:

```
handler trait CheckValueChanged subscribes
PotentialValueChange, ActualValueChange publishes
    ActualValueChange {
  Value cache = null;
  before (PotentialValueChange vc) {
    cache = vc.value ;
  }
  after (PotentialValueChange vc) {
    if (cache != vc.value)
      publish ActualValueChange(vc.value, vc.cell);
  }
  subscribe (ActualValueChange vc) {
    refresh (vc.value);
  }
  abstract void refresh(Value v);
}
```

in which `PotentialValueChange` and `ActualValueChange` are subtypes of `ValueChange`.

3.6 Event Dispatching

While publish/subscribe communication is traditionally asynchronous, implementations based on the Observer pattern are typically not. Also, the intuitive semantics of before and after handlers suggests that corresponding event handlers are actually invoked before and after the event, i.e., the execution of the wrapped code block. Therefore we defined our IIT such that implicit invocations triggered by events not wrapped around code blocks are dispatched asynchronously (to handler methods named `subscribe`), while those tied to code blocks are dispatched synchronously (to handlers named `before` or `after`).

4. IMPLEMENTATION IN SCALA

As a proof of concept, we have created a Scala [7] library that allows us to explore the possibilities of implicit trait invocation with syntax and semantics only marginally different from that suggested by the previous sections. The library, as well as further examples of its use, can be found under `http://www.fernuni-hagen.de/ps/prjs/IIT`; here, we sketch only its major contributions.

4.1 Event Classes and Publishing

Event classes must extend the predefined `iit.Event` class (where `iit` is the name of our package for IIT). `Event` is empty and basically serves type checking (to make sure that handler methods declare only subtypes of `Event` as their formal parameters). An event class (including standard constructors) is thus defined, in Scala syntax, as in

```
class ValueChange (val cell: Cell, val value: Value)
    extends iit.Event
```

In order to publish event instances the publishing class has to be enhanced with the trait `iit.Publisher`. This trait defines two methods, one for publishing an event asynchronously and one for publishing it synchronously. Both methods forward to a (singleton) dispatcher object (described in Section 4.2) and require as first argument the event creation expression (not an event!). In addition, the method for synchronous publishing requires a code block which is executed between execution of the before and after handler methods of subscribed handlers.

```
trait Publisher {
  def publishAsynchronously(eventCreation: => Event)
  { Dispatcher.publishAsynchronously(eventCreation) }
  def publish[T](eventCreation: => Event)(proceed: =>
      T): T =
  { Dispatcher.publish[T](eventCreation)(proceed) }
}
```

The `publish` method for instance can then be called from the publishing classes using the syntax

```
value = publish[Value](new ValueChange(this, value)){
  expression.evaluate()
}
```

4.2 Handler Traits, Subscribing and Event Dispatching

Handler traits must extend the predefined trait `iit.Handler`. Handler methods must be named `subscribe` (for asynchronous event handling), `before`, or `after` (both for synchronous event handling). The handled event type must be the parameter type of the handler method. The default of a handler method is do nothing.

Upon instantiation of a class using a handler trait, the instance is automatically registered with the `iit.Dispatcher` object for the event types it provides handlers for (`Dispatcher` determines this reflectively). Registration occurs automatically through a call of `iit.Dispatcher.register()`, inherited from the constructor of trait `iit.Handler`.

```
abstract trait Handler {
  assert(Dispatcher.register(this))
}
```

Once the dispatcher receives an event of a given type (as the result of calling one of the publish methods; see above), it checks for all subscribed handler instances whether the event is accepted by a potential filter. If no filter is defined or if the filter for this event type accepts the event instance, it is passed to the handler by calling the appropriate handling method.

```
object Dispatcher {
  def register(handler: Handler): Boolean = { ... }
  def publish[T](eventCreation: => Event)(proceed: =>
      T): T = {
    var e = (eventCreation)
    // select handlers with passing filters
    val handlers = ...
    // determine most specific before() and dispatch
    handlers.foreach(h => dispatchBefore(h, e))
    val ret: T = proceed
    e = (eventCreation) // re-create in new context
    // determine most specific after() and dispatch
    handlers.foreach(h => dispatchAfter(h, e))
    ret
  }
}
```

Note how this implementation depends on closures: `proceed` is a closure representing the code block passed to the `publish` statement and the `eventCreation` expression is also passed as closure, so that the event can be re-created (with new parameters from the context) after the execution of `proceed`.

5. CONCLUSION

Our approach to IIT is modelled after typed publish/subscribe (TPS) as advocated in [3]; it adds to it by defining event handlers as traits, and by allowing implicit invocation to be associated with a block of code, letting handlers act as a kind of method wrappers (similar to the advice of AspectJ). It also draws on our own prior work [13] on handling implicit invocation with implicit announcement in a modular, type-safe manner. Using traits subscribing to event types as suggested in this paper makes (1) the link between event sources and sinks explicit as the event types are part of the declaration headers of publisher and subscriber, and (2) the event handling code reusable as traits can enhance classes with event handling capabilities independently from their class hierarchy. The applications of IIT are those of publish/subscribe, including all applications of the widely used Observer pattern.

6. REFERENCES

[1] AspectJ. `http://www.eclipse.org/aspectj/`.

[2] J. Dingel, D. Garlan, S. Jha, and D. Notkin. Reasoning about implicit invocation. In *SIGSOFT Software Engineering Notes*, pages 209–221, 1998.

[3] P. Eugster. Type-based publish/subscribe: Concepts and experiences. *ACM Trans. Program. Lang. Syst.*, 29(1):6, 2007.

[4] P. Eugster and K. Jayaram. EventJava: An extension of java for event correlation. *Proc. ECOOP '09*, 2009.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[6] D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *Proc. VDM '91*, pages 31–44, London, UK, 1991. Springer-Verlag.

[7] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima, 2008.

[8] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In *Proc. of ECOOP '08*, pages 155–179, 2008.

[9] H. Rajan and K. J. Sullivan. Classpects: Unifying aspect- and object-oriented language design. In *Proc. ICSE '05*, pages 59–68. ACM Press, 2005.

[10] S. P. Reiss. Connecting tools using message passing in the Field environment. *IEEE Softw.*, 7(4):57–66, 1990.

[11] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behaviour. In *In Proc. ECOOP '03*, pages 248–274. Springer, 2003.

[12] F. Steimann. The paradoxical success of aspect-oriented programming. In *Proc. OOPSLA '06*, pages 481–497, 2006.

[13] F. Steimann, T. Pawlitzki, S. Apel, and C. Kästner. Types and modularity for implicit invocation with implicit announcement. *ACM TOSEM*, accepted for publication in 2011.

[14] K. Sullivan and D. Notkin. Reconciling environment integration and component independence. In *SIG-SOFT Softw. Eng. Notes*, pages 22–33, 1990.

[15] J. Xu, H. Rajan, and K. Sullivan. Understanding aspects via implicit invocation. In *ASE, IEEE Computer*, pages 332–335. Society Press, 2004.