WDS'07 Proceedings of Contributed Papers, Part I, 85–90, 2007. *ISBN 978-80-7378-023-4 © MATFYZPRESS*

Dynamic Traits

V. Žďára
Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic.

Abstract. Traits were introduced by Schärli *et al.* as reusable units of behaviour. Recently, the stateful version of traits was presented. In our work, we have reused stateful traits for dynamic composition. Dynamic traits can be dynamically bound on an object reference and thus specialize the object behaviour dynamically and contextually. Especially interesting feature of dynamic traits appears to be dynamic overloading mechanism, which can simplify transformations on structured data. This paper presents principles of dynamic traits, motivation examples and the language of dynamic traits.

1. Introduction

Traits, as presented in [21; 20], are parametrized behaviours composed of methods. Traits *require* a set of methods that are used by traits but not implemented. On the other hand, traits *provide* a set of methods. There were defined several operations on traits: the symmetric concatenation, the method exclusion and the method aliasing. From traits, classes can be built as compositions of traits, fields and a glue code. The glue code resolves possible conflicts of methods and binds getter and setter methods to the fields. Stateful version of traits [6] adds fields directly to traits. By default, the trait state is private but it is possible to access it during the trait or class composition through the special operator @@ that maps one or more fields to a new field. A shared field can be made by mapping the corresponding fields to one new field.

Traits can be perfectly used for dynamic composition due to its behaviour parametricity (the *require* methods set). If a trait is bound to an object reference, calling of required methods can be delegated (or forwarded) to the object. Our aim is to study this kind of dynamic composition of traits, which brings new interesting challenging issues concerning techniques of delegation, dynamic overloading and memory model.

The basic model of dynamic traits that we present here is more relaxed than the traditional object model. Figure 1 shows a schematic picture of an object in the traditional object model (to the left) and an object in the model of dynamic traits (to the right). In our model, an object is an open structure of dynamically bound traits. Moreover, the binding is context sensitive because the dynamic traits are bound on references instead of the identity. It is then possible to have more than one view to an object.

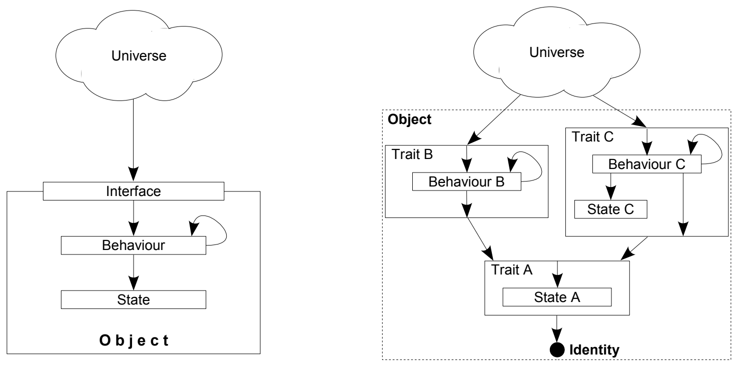

Figure 1. The traditional object model vs. the model of dynamic traits.

2. Motivation Examples

In fact, the idea of dynamic traits bound to objects is very similar to design patterns *Adapter* or *Wrapper* [14] or inner classes in *Java* [2], which are frequently used in the object-oriented programming. In this section, we present two examples from software engineering where the traditional class-based design is not ideal. The first example describes model-based software systems and the problems they are suffering from. The second example describes possible uses of dynamic traits in the data processing and the data transformation programming.

2.1. Model-based Software Systems

Many applications share the following architectural pattern: The application is composed of several modules sharing a common model. The model typically consists of pure data or data with some common behaviour like constraints or a code handling special object relations, etc. Weakness of this design is that not every module requires only the common behaviour of the model but also requires its own module-dependent view. To provide this, the programmer of the module must create suitable adapters to adapt the shared model for the module-dependent view. This solution does not conform with the pure object design and it rises problems with object identities. On the other hand, holding all module-dependent behaviour in the shared model is not possible in large software systems.

Well-known real examples of such software systems are JEE applications. A typical JEE application consists of web modules, entity beans and sessions beans. The entity beans represent the business model of the application, the session beans implement the business logic and the web modules are responsible for presentation. The session beans are natural candidates for to be implemented as dynamic traits.

2.2. Data-oriented programming

Data-oriented programming covers a wide range of data processing and data transformation applications. Such an application performs one or more tasks on streams of data. Typical examples are compilers, XSLT transformations etc. Due to the similar reasons as those discussed in the previous subsection, the object-oriented programming is not suitable for this kind of applications because data flows through several relatively independent transformation tasks, each requiring its specific view to the data. As we will show in section 4, transformation tasks can be elegantly represented by dynamic traits.

Current general-purpose programming languages with some support for data processing (e.g. [4] [7]) are focused on providing type and syntactic support for data. Behaviour is organized in standard way, e.g. as functions in CDuce or as classes in C_ω, which are in fact collections of functions. Both lack the possibility to use transformation behaviour in the pure object-oriented manner.

3. The Dynamic Traits Language

In this section, we review the essential features of a Java-like language enriched with the dynamic traits. Currently, we are working on experimental extensions of Groovy [23], which uses the following syntax and semantics.

Trait: We adopted traits with some minor incompatibilities from the original model [21]. Our traits do not support the method exclusion and the method aliasing. Instead, we support method hiding [17] as an alternative to the exclusion. For example, we can define the following traits.

```
trait Point {
  int x = 0;
  int y = 0;
}
trait Radial requires Point
  int getDiameter() { return Math.sqrt(x*x + y*y); }
  double getAngle() { return Math.atan(y/x); }
}
trait RadialPoint extends Point, Radial {  }
```

In the example, we have defined three traits. The first trait, `Point`, represents a 2-dimensional point represented by the cartesian coordinates. The trait `Radial` contains the radial view to `Point`; it is preferable to define the radial behaviour outside of `Radial` because the radial view is not an essential feature of `Point` and is used rather occasionally. The third trait, `RadialPoint`, inherits from the previous traits.

It is possible to use the traits as given in the following code:

```
Point p = new Point();
Point@Radial rp = p@Radial;
rp.x = 2;
rp.diameter;
```

```
p@Radial.diameter;
```

In the first line, new point is created. In the second line, new reference with the radial view is used as `rp`. In the third line, the x-axis position of the point is changed. In the forth and fifth line, the property `diameter` is given. While the first call is made through `rp`, the second is made through the adjusted `p`. In the following paragraphs, we review some important features of the language.

Identity: Each object is uniquely identified by its identity. The object state and the behaviour can change during the object's lifetime but the identity is still one and the same. In our model, when comparing references, it holds that all references to one identity are always equal. For example, if `p` is an existing reference then the expressions `p == p@TraitA` and `p@TraitA == p@TraitB` are both true.

Binding Operator: For the dynamic composition of traits an operator `@` is defined. When applying a trait on a reference (e.g. `p@Radial`), the reference is enriched with the trait (`Radial`). If the trait defines a state then it is created and initialized at this moment. From this point, it is possible to use the trait view (`Radial`) as well as the origin view (`Point`) with the reference. The resulting type of the composition will be `Point@Radial`. The operator `@` is, in the context of types, a special type operator. The operator merges the types `Point` and `Radial` and overrides all conflicting methods and fields in `Point` with the ones in `Radial`. Thus, the resulting type `Point@Radial` is compatible with the second type, `Radial`, but is not necessarily compatible with the first one — it depends on whether some parts of the type are overridden with their incompatible versions.

Dynamic Overloading: A powerful feature of the dynamic traits is the dynamic overloading of traits, i.e. a trait can be multiply defined for different required interfaces. This feature is similar to tuple classes implemented in [16] or to dynamic overloading of methods implemented, for example, in Cecil [9] or MultiJava [10]. Using dynamic overloading of traits instead of tuple classes or methods has several advantages: the dynamic lookup is made less frequently (not for every method call), a stateful trait is dynamically chosen and assigned to an object instead of stateless one-purpose method. An example of the module `Validator` presented in section 4 shows how traits can be effectively used to implement a task on a tree structure of nodes of different types. In [10] authors show how *MultiJava* can solve the problem with managing *Visitors* and *Nodes* in compilers. We think that our model of dynamic traits bound on the nodes can solve the problem in easier and cleaner way.

Releasing Operator: For comfortable work with composed references, we have defined a releasing operator `!`. The operator removes the most recently bound trait from the reference. For example, let `p` has the type `Point@Radial` then the expression `p!` has the type `Point`. This operator can be used to remove module specific behaviour from a reference. Note that the released trait is actually removed and not only hidden.

Information Hiding: The model of dynamic traits supports more flexible information hiding. First, objects are not encapsulated as in the standard model but they are rather open structures of traits. Second, traits as the basic building blocks support method or interface hiding. This is possible because of the traits do not induce a subtyping hierarchy. Moreover, we added an extra sealing operator `#`. The operator works in the same way as the operator `@` with the exception that it hides the original object. For example, we can forbid to access fields of the point `p` by `p#Radial`. The type of this expression is `Radial` instead of `Point@Radial` and it is not possible to down cast to that type.

3.1. Dispatching of Messages

When a message cannot be handled in a trait, it is sent to the underlying object. The message is sent through the chain of bound traits until an appropriate method body is found. There are two ways of handling such a method call: delegation or forwarding. In the first case, the method is executed in the context of the trait receiving the message. In the second case, the method is executed in the context of the trait containing the method. We decided to use dispatching by forwarding because it is safe. Moreover, we do not need delegation because we have trait inheritance.

Generally, dispatching by delegation is not safe. Let us consider the following example. In systems using object delegation, compatibility (subtyping) must be ensured when extending objects. That is why all parent self calls are redirected back to the called object. Thus, if we allowed object delegation, similar constraints would have to hold in our model. Then, the following program would contain unsafe code.

```
abstract trait RealFun {
  abstract Real getArg();
  Real eval() { return getArg(); }
  Real evalDefault() { return self@EvalForZero.eval(); }
}
trait EvalForZero requires RealFun {
  Real getArg() { return 0; }
}
trait LogFun inherits RealFun {
  PositiveReal getArg() { return 1; }
  Real eval() { return log(getArg()); }
}
```

The program contains a definition of three traits: the trait `RealFun` is the base trait for real functions of one argument, the trait `LogFun` is an implementation of the function *log* and the trait `EvalForZero` is used to eval the real function at zero. The trait `LogFun` restricts the argument (method `getArg`) to positive real values only to make the evaluation of *log* safe. But the trait `RealFun` does not know about this restriction and enriches the self-reference with `EvalForZero` in the method `evalDefault`. Thus, if we ran the following code `new LogFun().evalDefault()` a runtime exception would occurred because of `log(0)`. To provide a safe solution of the object delegation, we would have to dramatically restrict the operation `@`, which we do not want. The presented example is similar to the example in [1], Chapter 8, p. 109.

Another reason to abandon the object delegation is the absence of code security by the broken encapsulation. For example, let an object has a method `hasRole` returning `true` or `false` depending on the current user has or has not a role authorizing him/her to call the object's secured services. In the systems with the object delegation, it is easy to specialize `hasRole` to override the security constraints. In our model, the secured services will always call the right method body.

4. An Example

In this section, we will show a more complex example of the dynamic traits. The example defines a validator for a data model containing diagrams of nodes connected by edges. The following code shows how a validator could look like.

```
public trait Validator
    requires Container s {
  public void validate() throws ValidationException {
    for (Item e : s.children) {
      e@This.validate();
    }
  }
}
public trait Validator
    requires Node s {
  public void validate() throws ValidationException {
    // do node validation
  }
}
public trait Validator
    inherits Container@This c, Node@This n
    requires Diagram s {
  public void validate() throws ValidationException {
    // first do diagram validation
    n.validate(); // do node validation
    c.validate(); // validate children
  }
}
public trait Validator
    requires Edge s {
  public void validate() throws ValidationException {
    // do edge validation
  }
}
```

We have used dynamic overloading for `Validator`. The first trait defines a general validation code for containers. The second and the fourth traits define the validation code for nodes and edges, respectively, and the third trait inherits from the first and the second traits and defines the validation code for diagrams. In the first trait, the code `e@This.validate()` binds itself (actually, it can be any of the traits of name `Validator`) to the child nodes and runs the validation of them. The example shows how it is possible to implement a task on an object graph by a set of traits. This solution does not affect the classes of the model, which is not possible when using a *Visitor* class and when a kind of "visit contents" method must be imported into the classes. Moreover, the traits are modular and can be added to or removed from the program without recompiling the rest of the code.

5. Discussion

The model of dynamic traits can be compared to several existing approaches. We can compare partial parts of our model like traits, message dispatching and the open structure of objects but, by the time of writing this paper, we have not found an approach similar to ours as a whole.

Traits were recently studied in [20; 21; 13; 19; 17; 6]. Most of the papers consider traits as building units for static composition of classes. In [19] authors study dynamic substitutions of traits in runtime. This idea is similar to object reclassification studied in [11]. Our model is different in that it allows to specialize an object contextually by modifying the reference to the object, while *Chai*$_3$ presented in [19] changes the object. In our model, schema of dependence is inverse: dynamic traits depend on the underlying object while an object in *Chai*$_3$ depends on its substitutable trait.

There were presented several approaches of combining classes and delegation, e.g. [15] [12]. The author of [15] defines delegatees in classes. Delegatees are references to a parent object in the chain of delegation. In [12] the author defines a dynamic inheritance through pattern variables as superpatterns in *gbeta*. Both delegatees and pattern variables represent rather inverse type of delegation than our model does.

It also is worth to mention two today's popular scripting languages JavaScript [24] and Groovy [23], which support dynamic modification of classes and some ways of delegation. In Javascript, classes can be created by class constructors (in fact they are JavaScript functions) and inheritance is implemented by prototype objects, which serve as parent objects to whom messages are delegated in case they cannot be handled by the called object. By modifying prototype objects, behaviour of all derived objects is changed. Groovy rather supports classes in the style of Java. Moreover, it adds metaclasses, which are special handlers of methods and property getters and setters, that can be assigned in runtime to classes. Thus, objects of a class can be enriched with new behaviour. The main difference between dynamic traits and JavaScript or Groovy is that trait binding operators change behaviour only of one object and only contextually while changing an object or a prototype object in Javascript or setting a metaclass to a class in Groovy changes behaviour of all objects of the class instead. This "all objects modification" results in hard typability—both JavaScript and Groovy have dynamic type system only, even though there are attepts at static type inference [3].

The open structure of objects used in our model is similar to *split objects* [5]. Split objects are composed of pieces representing different viewpoints to objects. The pieces of an object form a tree and unhandled messages are delegated from children to their corresponding parents. Pieces are slots which can contain properties and methods. However, the model of split objects is too general (it is derived from *Self* [22]) to compare advantages or disadvantages especially because of the absence of a static typechecking and safety.

Finally, our model uses the dynamic overloading of traits which is similar to multi-methods studies in [9] [10] [18] [16] or *tuple classes* [16]. The purpose of multi-methods is to provide multiple dispatch and/or provide more flexible method overriding [8] to methods. The tuple classes are more similar to our approach. They can define methods on tuples but they do not define a state and cannot be bound permanently to an object. Moreover, modifying a tuple class requires type checking of the whole program, our traits do not, as we believe.

6. Conclusion

We have presented the model of dynamic traits, which was designed on our experience and on the recent studies in the object-oriented programming. We are working on a prototype compiler of Groovy extended with the dynamic traits. In parallel, we study formal issues of the dynamic traits. Particularly, we have defined a core calculus of state, behaviour and references (from the lack of space, we did not present it in this paper). Our current aim is to extend the core calculus with multi-traits, which works in the similar way as multi-methods.

References

[1] Abadi M., Cardelli L.: Theory of Objects. Springer, 1996.

[2] Arnold, K., Gosling, J.: The Java programming language. Addison-Wesley, 1996.

[3] Anderson, Ch., Giannini, P., Drossopoulou, S.: Towards Type Inference for JavaScript. In *ECOOP, Lecture Notes in Computer Science*, Vol. **3586**, pp. 428-452, Springer, 2005.

[4] Benzaken, V., Castagna, G., Frisch, A.: CDuce: a white paper. Working document, November 12, 2002. `www.cduce.org`

[5] Bardou D., Dony Ch.: Split Objects: A Disciplined Use of Delegation within Objects. *ACM SIGPLAN Notices*, **31, 10**, 122–137, 1996.

[6] Bergel, A., Ducasse, S., Nierstrasz, O., Wuts, R.: Stateful Traits. In *Proceedings of the International Smalltalk Conference*, ESUG Academic Track 2006, Prague, Czech Republic September, LNCS, Springer-Verlag, 2006.

[7] Bierman, G., Meijer, E., Schulte, W.: The essence of data access in C_ω. In *ECOOP, Lecture Notes in Computer Science*, Vol. **3586**, pp. 287-311, Springer, 2005.

[8] Castagna, G.: Covariance and Contravariance: Conflict without a Cause. *ACM Transactions on Programming Languages and Systems*, 17(3), pp. 431-447, May 1995.

[9] Chambers, C.: Object-Oriented Multi Methods in Cecil. 1992.

[10] Clifton, C., Leavens, G.T., Chambers, C., Millstein, T.: MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. Technical Report TR #00-06a, Iowa State University, Iowa, 2000.

[11] Drossopoulou, S., Damiani, F., Dezani-Ciancaglini, M., Giannini, P.: Fickle: Dynamic Object Reclassification ECOOP, Lecture Notes in Computer Science, Vol. 2072, pp. 130-149, Springer, 2001.

[12] Ernst, E.: Dynamic inheritance in a statically typed language. *Nordic Journal of Computing,* **Vol.6**, 72–92, 1999.

[13] Fisher, K., Reppy, J.: Statically Typed Traits. Technical Report TR-2003-13, University of Chicago, 2003.

[14] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[15] Kniesel, G.: Implementation of Dynamic Delegation in Strongly Typed Inheritance-based Systems. Institut für Informatik III, Universität Bonn, Report, 1995.

[16] Leavens, G. T., Millstein, T. D.: Multiple dispatch as dispatch on tuples. *OOPSLA'98: Proceedings of the 13th ACM SIGPLAN conference on object-oriented programming, systems, languages and applications,* (New York, NY, USA), ACM Press, 1998, pp. 374-387.

[17] Reppy, J., Turon, A.: A foundation for trait-based metaprogramming. International Workshops on Foundations of Object-Oriented Languages, 2006.

[18] Salzman, L., Aldrich, J.: Prototypes with Multiple Dispatch: An Expressive and Dynamic Object Model. In *ECOOP, Lecture Notes in Computer Science*, Vol. **3586**, pp. 312-336, Springer, 2005.

[19] Smith, Ch., Drossopoulou, S.: *Chai*: Traits for Java-like Languages. In Proc. of *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, Glasgow, UK, July 25-29, 2005.

[20] Schärli N., Ducasse S., Nierstrasz O., Wuyts R., Black A.: Traits: The Formal Model. *Technical Report CSE* **02-013**. OGI School of Science & Engineering, 2002.

[21] Schärli N., Ducasse S., Nierstrasz O., Wuyts R., Black A.: Traits: Composable units of behaviour. *Technical Report CSE* **02-013**. OGI School of Science & Engineering, 2002.

[22] Ungar D., Smith R.: Self: the power of simplicity. *LISP and Symbolic Computation: An International Journal,* **4(3)**, 1-20, 1991.

[23] Groovy: `http://groovy.codehaus.org`

[24] JavaScript: ECMAScript Language Specification, Standard ECMA-262, 3rd Edition, December 1999. `http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf`