VRIJE UNIVERSITEIT BRUSSEL
FACULTY OF SCIENCE
DEPARTMENT OF COMPUTER SCIENCE

Context-Oriented Meta-Programming

Thesis submitted in partial fulfilment of the requirements for the degree of Licenciaat in de Informatica

By: Stijn Timbermont

Promotor:
Prof. Dr. Theo D'Hondt

Advisors:
Dr. Pascal Costanza
Dr. Jessie Dedecker
Kris Gybels

Augustus, 2006

Abstract

In current information systems, context information has an increasing influence on the functionality of an application and on how the application should be executed. The systems must be able to adapt their behavior to the context in which they are running. This adds a new dimension to the complexity of these systems but current programming languages do not offer the necessary means to modularize the context-dependent behavior. Recent research has lead to Context-Oriented Programming, a new programming paradigm that addresses this problem by defining a new unit of modularization that allows to group those parts of a program that are affected when the behavior of the program should be adapted to a different context.

However, context-oriented programming only focuses on the application logic itself and does not yet provide means to adapt the way how the application is executed. How a program is executed is defined in the semantics of the used programming language, also called the meta-level. In this dissertation we present how context-oriented programming at the meta-level can be used to express language semantics that depend on the context in which a program is executed and illustrate the usefulness of this approach based on our own implementation, called TinyContext.

Acknowledgments

I would like to take this opportunity to express my gratitude towards all the people who supported me tremendously throughout the writing of this dissertation and without whose help I would have never finished it.

Prof. Dr. Theo D'Hondt for promoting this dissertation.

Dr. Pascal Costanza for coming up with the subject and for guiding me along every step through implementation and writing to proof-reading.

Dr. Wolfgang Demeuter, Dr. Jessie Dedecker and Kris Gybels for proof-reading my work and for their valuable comments.

The researchers at the Programming Technology Lab for listening and giving comments during the thesis presentations.

My brother Toon and my sister Leen for proof-reading and for supporting me.

My fellow students, friends and family for being there.

And last but not least, my parents for supporting me all these years and for giving me the opportunity to obtain a higher education.

Contents

Chapter 1

Introduction

In this dissertation we present how context-oriented programming at the meta-level can be used to express language semantics that depend on the context in which a program is executed and illustrate the usefulness of this approach based on our own implementation, called TinyContext.

In current information systems, context information has an increasing influence on the functionality of an application and on how the application should be executed. This context information can be anything, ranging from technical information such as battery life, bandwidth or the availability of resources such as printers to functional information such as the geographical information of a portable device, the time of the day or the properties of the user. The need to adapt the system to the context of the execution adds a new dimension to the complexity of the information systems. Dealing with this added complexity requires the right tools but current programming languages lack the necessary support.

Indeed, developing software systems that are able to adapt their behavior based on the context in which they are running often proves to be hard because current programming languages do not offer the necessary means to modularize those parts of a program that should depend on the context. This leaves the programmer no other option than to spread out context-dependent code in the entire application or to use modularizations that are not well-suited for context-dependent behavior.

Recent research has lead to Context-Oriented Programming (Costanza and Hirschfeld, 2005), a new programming paradigm that addresses this problem by defining a new unit of modularization that allows to group parts of those program that are affected when the behavior of the program should be adapted to a different context. In Chapter 2 we elaborate on context-oriented programming.

However, context-oriented programming only focuses on the application logic itself and does not yet provide means to adapt the way how the application is executed. How a program is executed is defined in the semantics of the used programming language, what is also called the meta-level of a software system. This indicates that context-oriented programming should also be available at the meta-level and a collection of constructs is required to support meta-programming that takes into account the context. Modifying or extending the semantics of a programming language can already be achieved with metaobject protocol (Kiczales et al., 1991), but adapting the semantics to the context of execution experiences the same problems as those for the application logic. This will be establishes in Chapter 3.

In this dissertation we will combine context-oriented programming and meta-programming. In order to do so, we will present TinyContext, an experimental environment that allows us to use the constructs for context-oriented programming not only for base-level programs, but for meta-programs as well.

1.1 Organization of the Text

In Chapter 2, we introduce context-oriented programming. We start with explaining the motivation behind context-oriented programming by working out a scenario where we need to define context-behavior and we identify the problems that occur with current programming languages. Then we show how context-oriented programming solves these issues and present ContextL, the first language extension that provides explicit support for context-oriented programming.

In Chapter 3 we start with a general description of meta-programming and we continue with demonstrating how meta-programming with a metaobjects protocol allows the programmer to modify and extend the language semantics to his own needs. However, defining context-dependent behavior at the meta-level poses the same problems as those encountered at the base-level. This suggests to use context-oriented programming at the meta-level as well.

In Chapter 4 we present our first contribution TinyContext, the experimental environment that will be used to implement our approach. TinyContext is based on TinyCLOS, a kernelized version of CLOS, implemented in Scheme, with a simple but powerful metaobject protocol. TinyContext is an adaptation of TinyCLOS in order to support context-oriented programming. The language constructs for context-oriented programming are based on a subset of the features found in ContextL. TinyContext is designed in such a way that it will have the necessary ingredients to implement our approach for context-oriented meta-programming.

In Chapter 5 we combine Chapter 2 and Chapter 3 by using context-oriented programming on the metaobject protocol of TinyCLOS. This results in a number of language constructs for TinyContext that allow the programmer to modularize context-dependent meta-behavior and to adapt the semantics of the programming language to the context of execution.

In Chapter 6 we give three examples of how context-oriented programming can be used. The show that context-oriented meta-programming is useful and effective for adapting the semantics of the language to the context, as well as for defining new language constructs for defining context dependencies.

In Chapter 7 we compare our approach with aspect-oriented programming because both approaches seem to have an overlapping range of applications. This is especially because aspect-oriented programming and meta-programming are good approaches for expressing non-functional concerns.

In Chapter 8 we present our conclusions and we identify some areas for future work.

Chapter 2

Context-Oriented Programming

This chapter introduces *Context-Oriented Programming* based on the work of Costanza and Hirschfeld (2005).

Context-Oriented Programming is a new programming paradigm that provides techniques for modularizing behavior that might depend on the context in which the program is executed.

We start in Section 2.1 with explaining the motivation behind context-oriented programming. We work out a scenario where context-dependent behavior has to be defined and we identify the problems for doing so with current programming languages. We continue in Section 2.2 by explaining how context-oriented programming addresses these problems. Section 2.3 presents ContextL, the first programming language extension that provides explicit support for context-oriented programming.

2.1 Motivation

In this section we introduce the motivation behind context-oriented programming. First we introduce a simple working example (in Section 2.1.1) where we try to add some context-dependent behavior (in Section 2.1.2). We show how the usage current programming languages leads to several problems.

2.1.1 Scenario

One of the key ideas of object-oriented programming is that objects themselves know how to behave. A typical example to introduce this idea is a person object that responds to the message *display* (Figure 2.1). Instead of asking the person object for its name and

```
class Person {
  String name, address;
  void dispay() {
    println("Person");
    println(name);
    println(address);
  }
}
```

Figure 2.1: Simple introductory class

address and then displaying that information, we just tell the person object to display itself by sending it a message. The person object owns the knowledge of how it should be displayed. This is called the *Tell, Don't Ask* principle.

> "Procedural code gets information then makes decisions. Object-oriented code tells objects to do things." (Sharp, 1997)

This means that you should tell objects what you want them to do and let the objects decide, based on their internal state, how they respond; you should not ask for their internal state, make decisions based on that state and then produce some behavior yourself. Doing so violates the encapsulation of the object.

Bock (2000) illustrates this nicely with the tale of the paperboy and the wallet. The paperboy comes by your front door, rings the doorbell and asks for payment for a job well done. You turn around and let the paperboy pull out your wallet out of your back pocket. The paperboy takes two bucks from your wallet, puts them in his own wallet and finally puts your wallet back in your back pocket.

This tale shows how procedural programming breaks the encapsulation of the customer. In object-oriented programming, the paperboy should simply tell the customer to pay two dollars instead of taking the wallet. In fact, the paperboy should not even have any knowledge about the wallet of the customer. Instead, the customer should deal with his own wallet when the paperboy tells him to pay some money.

We now extend our simple person example with students and professors that respectively follow and teach courses. A class diagram for this example is shown in Figure 2.2. A student has an extra field for the year in which he is in, and a list of the courses that he is following. A professor has a field for the faculty where he teaches, a field for his salary, and a list of the courses that he teaches. A course has just a title.

The code for displaying people, students, professors and courses is defined in the classes themselves. For example, displaying a student might be defined as follows. First the person information is displayed (the super call), then the student information (the year and the followed courses) is displayed.

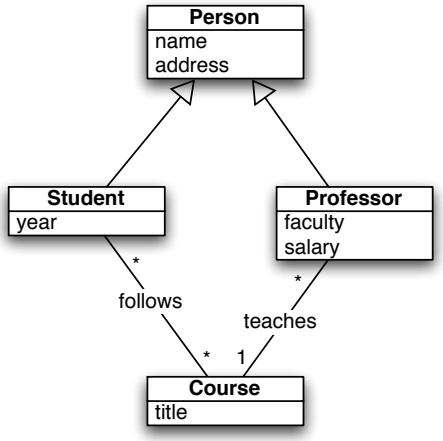

Figure 2.2: Class diagram

```
void display() {
  super.display();
  println("Student");
  println(year);
  for (Course course : follows) {
    course.display(fullInfo);
  }
}
```

2.1.2 Context-Dependent Behavior

Now consider that we want to display a person differently depending on the context in which the person has to be displayed. In other words, we want to define different views on the same objects and choose a particular view based on context information. This context information can be anything that might have an influence on the behavior of a program. Typical examples of "context" are geographical information, time of the day, battery life of for example a laptop, other devices that are nearby, etc.

In our person example, context information might be *who* is requesting the data. Suppose the class hierarchy for people and courses is part of a bigger application that provides access to the database of university for a range of possible clients. For example, an anonymous visitor might want to have a look at the various courses that are offered and is allowed to see the names of the professors that teach the courses, but a visitor may not see any information about the students that follow those courses, nor see any personal information about the professors. However, somebody who is part of

the academic community and is logged into the system does have the right to see the list of students that follow a particular course but personal information should still be hidden. On the other hand, somebody working in the administration of the university should have full access and see all the personal information as well.

With the current mainstream programming languages, there are two techniques for realizing this.

- The context information is passed around to the objects and they use conditionals (`if` or `switch` statements) on the context information in order to decide how they have to behave.
- The behavior for the different contexts is factored out into separate classes and one of these classes is chosen based on the available context information.

In the following paragraphs we investigate both options in more detail.

Conditionals If the methods use conditionals, they need information to test on, so we have to pass some parameter(s) to `display` that indicate(s) something about the context. The various method implementations for `display` in the different classes can use this information to decide what should be displayed and what not. How code for this approach looks like is demonstrated in Figure 2.3. The conditional statements and the variable indicating the context are highlighted.

This approach has several disadvantages and shortcomings.

- First of all, if statements in order to achieve polymorphic behavior is not good object-oriented programming style. One of the problems with if tests is that the number of possible cases is fixed to those that are actually used in the conditional statement. If there is suddenly some alternative case, every test in all the different classes have to be manually modified in order to support the new case.
- Another consequence of using if tests is that all the code for a particular case is spread out in different classes. There is no overview of the context-dependent behavior. In the person example, finding out what changes when fullinfo is true requires to look at all the methods for displaying. It would be better if the all the code for a particular context situation is grouped together.
- A different problem lies in how the context information is passed around. In our example, all clients of `display` must pass some arguments indicating how the object should be displayed. This mechanism works well as long choosing the displaying strategy and invoking `display` happens at the same place. If that is

```
class Person {
  String name, address;

  void display(boolean personalinfo) {
    println("Person");
    println(name);
    if (personalinfo) {
      println(address);
    }
  }
}
```

```
class Professor extends Person {
  String faculty;
  double salary;

  void display(boolean personalinfo) {
    super.display(fullinfo);
    println("Professor");
    println(faculty);
    if (personalInfo) {
      println(salary);
    }
  }
}
```

```
class Course {
  Professor teacher;
  List<Student> students;

  void display(boolean studentinfo, boolean personalinfo) {
    println("Course");
    println(title);
    teacher.display(personalinfo);
    if (studentinfo) {
      for (Student student : students) {
        student.display(personalinfo);
      }
    }
  }
}
```

Figure 2.3: Context-dependent behavior with if statements

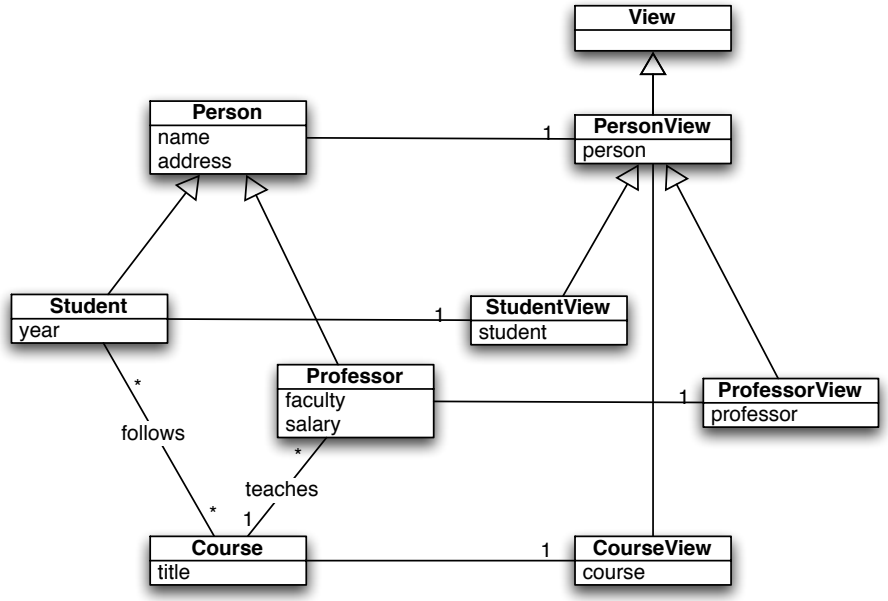

Figure 2.4: Spreading out context-dependent behavior to different classes

not the case (e.g. `display` is invoked indirectly), the variables describing the context must be passed along to every intermediate object, even if these objects have no interest in this context information. This problem is already visible in our example: how a course should be displayed does not directly depend on whether personal information should be displayed (a course does not have personal information). Still, the variable `personalinfo` must be passed to a course, because displaying a course might display student objects as well.

- A similar problem lies in the fact a class only receives the same context information as its superclass. In our person example, the context information that is passed to the class `Student` is restricted to the context information passed to the class `Person` (in this example: only one variable personalinfo). If the way that a student is displayed can depend on some other factor, it must be passed to every person as well, even if only `Student` uses it.

Separate View Objects Another approach for defining context-dependent behavior is to factor it out to different classes. For our example, this means that the displaying code is factored out into view objects. A class diagram of this new design is shown in Figure 2.4. Now the person classes are no longer aware of any displaying behavior. This allows us to define multiple views on the same object, one for each context.

This approach immediately solves the first two problems with if statements: new views

can be defined whenever needed, without modifying any other view, and all the code for a certain context is grouped together in one class. However, this approach introduces new disadvantages.

- In this design, the *Tell, Don't Ask* principle is no longer respected. Because the code for displaying is no longer associated with the person objects, each class in the person hierarchy requires a specialized view class. The person objects must then make their internal state available to those views. In other words, instead of telling the person objects to display themselves, the views pull out the necessary information out of the person objects.
- The conceptual simplicity of the original design is highly complicated by the added classes.

2.2 Context-Oriented Programming

The goal of context-oriented programming is to allow a programmer to define and group context-dependent behavior on objects - the views - without actually removing that knowledge from the object - respecting the *Tell, Don't Ask* principle. This is achieved by factoring out the context-dependent behavior into layers instead of separate classes. Different versions of the same operation can coexist in different layers, but they still are associated with the same class. In our person example there might be for example a layer that groups together all the methods that display the full info of a person.

Once the context-dependent behavior is grouped in layers, we can activate the appropriate layer based on the context information. This will adapt the running program by adding the definitions in the layer. Sending the same message to an object can now produce different behavior.

We can summarize the concepts of context-oriented programming as follows.

Grouping in Layers Context-dependent behavior is grouped in layers. Each layer represents a particular case and can group definitions for different classes. In our person example, there will be a layer that groups all definitions for displaying all the information instead of just some of it.

Dynamic Activation Layer activation should be dynamic. Once a layer is activated, not only the code in the layer activation itself, but also the code that is called indirectly, should be adapted. This way of layer activation solves the problem of having to pass layer information to objects that do not use it.

Thread Locality Activating a layer in one thread should not interfere with other threads. Each thread has its own context of execution and activation of a layer in one thread should not automatically activate it in any other thread.

2.2.1 Layers

Layers provide a new unit of modularization that complements classes and methods. A layer is basically an identity with no further properties of its own. However, new definitions can be placed in a layer by explicitly referring to one. Consider our person example. Suppose that there are three different ways to display the objects. Each way has its own layer that groups the methods for displaying the objects of the different classes. Note that these method definitions are still associated with the classes as well. Whenever we make a decision about how person objects should be displayed, we activate the appropriate layer. Whenever a person object is sent the `display` message, the one of the activated layer is used.

All *normal* definitions that do not specify a layer are automatically placed in an implicitly predefined root layer. That way, all definitions belong to exactly one layer and by consequence, a program is partitioned in layers. When the program starts running, only the root layer is active. At any certain point in the execution of the program, the actual program that is running is the combination of all active layers.

2.2.2 Layer Activation

Layers are activated dynamically in the control flow of a program. This means that a layer is activated before execution of some piece of code and automatically deactivated when the code has finished. The activation is *dynamic* because not only the code inside the layer activation will use the definitions of the activated layer, but also all the code that is invoked indirectly. Consider the person example. At some point in the program, we decide to use the layer that will print all information of person objects. We activate the layer and execute some code in the context of that layer. If the code sends `display` messages to person objects, they will be displayed by using the method defined in the activated layer. If the code in the layer activation does *not* display person objects, but just invokes some other behavior that might display person objects, the methods defined in the activated layer are still used.

Layer activation can be nested. In that case all active layers are combined and all the definitions of the activated layers become part of the running program. If a layer is activated when it was already active, it is only deactivated again when control flow returns from the *first* time that that layer was activated. If two layers have definitions

for the same operation, and they are both activated, the order in which the layers were activated decides which one is used.

Layer activation has similar properties as dynamic scoping (see Appendix A for more information about dynamic scoping). The *context of execution* of a program is implicitly defined as the combination of all active layers and behaves as a *dynamically scoped variable*:

- Changes to the context of execution (layer activation) happen in the control flow of a running program, similar to defining new bindings for a dynamically scoped variable.
- The context of execution is restored when the control flow returns from the layer activation. In the same way, the value of a dynamically scoped variable is restored when the control flow returns from the definition of the new binding.
- Each thread has its own context of execution: activating a layer in one thread does not activate it in other threads. Dynamically scoped variables are thread-local as well.

2.3 ContextL

ContextL is the first language extension that provides explicit support for context-oriented programming. We will present the most important constructs of ContextL by using the introductory *person* class as an example. Since ContextL is an extension of CLOS, we define the person again, but now in CLOS code.

```
(defclass person ()
  ((name :initarg :name
         :accessor person-name)
   (address :initarg :address
            :accessor person-address)))
```

This piece of CLOS code defines the class `person` with no superclasses and two slots (CLOS terminology for fields or instance variables) `name` and `address`. The slot `name` can be initialized with `:name` and accessed with `person-name`. The same goes for the slot `address`.

2.3.1 Layered Functions

Now we can define some behavior on this new class (CLOS is based on the notion of *generic functions*). The goal is to show how we can have different views on our

person class by defining the behavior in layers. This is achieved by defining a *layered function* and then adding some *layered methods* to it.

```
(define-layered-function display-object (object))
```

This code defines a generic function that has support for context-oriented programming. It takes one parameter object. In order to make this function to perform something useful, we have to define methods on it. First we define the default displaying behavior in the root layer. In ContextL, this root layer is denoted with t. The only parameter object is specialized on the class person.

```
(define-layered-method display-object
  :in-layer t
  ((object person))
  (format t "Person~%")
  (format t "Name: ~A~% (person-name object)))
```

This method will only print the name of the person. Now we define the layer full-info-layer and place a method in that layer that will print not only the name but also the address.

```
(deflayer full-info-layer)

(define-layered-method display-objects
  :in-layer full-info-layer
  :after ((object person))
  (format t "Address: ~A~% (person-address object)))
```

This method adds behavior to the previous defined method for the class person that is to be executed :after the previous one has been executed. Because this :after method is defined in full-info-layer, it will only be executed if that layer is active. This is demonstrated in the following transcript.

```
> (defvar *stijn*
    (make-instance 'person
      :name "Stijn Timbermont"
      :address "Lokerenbaan 166, Zele"))
> (display-object *stijn*)
Person
Name: Stijn Timbermont
> (with-active-layers (full-info-layer)
```

```
    (display-object *stijn*))
Person
Name: Stijn Timbermont
Address: Lokerenbaan 166, Zele
```

2.3.2 Layered Classes

Not only behavior can be defined in layers. Class definitions as well can be confined to a specific layer. For example, we might confine address information of a person to the `full-info-layer`. First we define the class `person` without a slot for an address, as follows.

```
(define-layered-class person ()
  ((name :initarg :name
         :accessor person-name)))
```

Note that the class is defined by using `define-layered-class` in order to support context-oriented programming (similar to `define-layered-function`).

Now we define a class for representing an address inside `full-info-layer`.

```
(define-layered-class address
  :in-layer full-info-layer
  ((street :initarg :street
           :layered-accessor address-street)
   ;; number, zip code, city, country
   ... )
```

Having this class confined to `full-info-layer` does not have a useful effect yet. The class can still be instantiated from anywhere. Layered classes become interesting when we define a class in a layer when it was already defined in another layer. In that case, the class is not replaced, but the slot definitions of the new class definitions are added to the original class.

```
(define-layered-class person ()
  :in-layer full-info-layer
  ((address :initarg :address
            :layered-accessor person-address)))
```

So in this example, `person` still has its original slot `name`, but additionally, it gets the slot `address` in `full-info-layer`. Since the accessor function `person-address`

and all the accessor functions of the class `address` are declared as `:layered-accessor`, the slots are actually only visible when the `full-info-layer` is active. When it is not active, an error is raised when these accessor functions are called. This allows the programmer to restrict the view of slots to certain layers.

2.4 Summary

In this chapter we introduced Context-Oriented Programming.

We started with explaining the motivation behind context-oriented programming by working out a scenario where context-dependent behavior had to be defined. As a concrete example, we used a class hierarchy representing people, students, professors and courses. The goal was to define different views on these classes in such a way that we could choose a view based on the context in which we wanted to display the person objects.

We observed that current programming languages lack the means to do so. Using `if` statements violates one of the fundamental principles of object-oriented programming, namely to avoid conditionals for expressing polymorphic behavior. Separating out the views into different classes breaks the encapsulation of the objects person objects: their internal state has to be exposed to the views.

Context-Oriented Programming addresses these problems by providing a new unit of modularization that can group context-dependent behavior without separating it from the different classes.

Then we presented ContextL, the first programming language extension that provides explicit support for context-oriented programming.

Chapter 3

Engineering the Meta-Level

In this chapter we explain meta-programming and demonstrate some meta-programming systems. We start in Section 3.1 with a general definition of meta-programming. We continue in Section 3.2 by showing how a metaobject protocol can offer meta-programming facilities in a language. Because especially the CLOS metaobject protocol is important for the rest of the text, we explain it in more detail in Section 3.3. In Section 3.4 we examine how a metaobject protocol can be used to express context-dependent behavior at the meta-level.

3.1 Meta-Programming

In this section we explain meta-programming and introduce the terminology. The following paragraphs are based on (Maes, 1987b) and (Maes, 1987a) and are adapted from (Gybels, 2001).

A *program* describes a *computational system* that reasons about some kind of problem domain. For example, the domain of a banking application consists of accounts, clients, transactions, etc. They are called the *entities* of the domain and they are represented by structures in the computational system. The program prescribes how they can be manipulated.

A special kind of computational systems are those whose problem domain consists of computational systems themselves. Such systems are called *meta-systems*. The program of a meta-system is a *meta-program*. The entities of the domain are then representations of computational systems. If only the program is manipulated, *meta-programming* boils down to writing programs that manipulate programs as data. The programming language that is used for meta-programming is called the *meta-language*.

The programming language that the meta-program reasons about is called the *base-language*. A program written in that language is a *base program*.

Meta-programming is much used for implementing tools for programming languages and software development. A compiler or an interpreter are common examples of meta-programs. Some programming languages such as C and Lisp provide support for code generation with macros. Another example is code generation from UML models.

A reflective system is a computational system that reasons about itself: part of the domain of the system is the system itself. It includes structures representing (aspects of) itself, making up its self-representation. Examples of the use of reflection are: keeping performance statistics, debugging, tracing, self-optimization, self-modification, etc. Reflection does not participate directly in solving problems in the actual domain of the system. Instead, it provides means to organize the internals of the system.

3.2 Metaobject Protocols

Metaobject Protocols are interfaces to the language that give users the ability to incrementally modify and extend the language (Kiczales et al., 1991). Most languages have well-defined and fixed semantics and users are expected to treat them as immutable. The idea of a metaobject protocol is to "open a language up", allowing the users to modify or extend the design and implementation to their needs. In addition, these modifications and extension the the language can be expressed within the language itself.

Languages that support meta-programming by means of a metaobject protocol, reify *metaobjects* such as classes and methods and define *protocols* that offer the programmer hooks to the interpreter or compiler in order to change the semantics of the metaobjects. For a programmer that only uses the language, a class definition is a declaration of the properties of a certain type of objects. and a method declaration defines the behavior on the objects of his class. For the meta-programmer, both the class definition and the method definition are creations of metaobjects as instances of two different metaclasses. The operations defined on the metaclasses tell how the metaobjects are created and how they behave at run-time. For example, how instances of (normal) classes should be created are defined in their respective metaclasses. Because the metaclasses are also just classes, they support subclassing, just as the base-level classes. By subclassing a metaclass and overriding the methods that are defined on them, the default semantics can be modified or extended.

Metaobject protocols come in three categories. The difference lies in the moment when the metaobjects are reified and made available for meta-programming (compile-time, load-time or run-time).

- Compile-time metaobject protocols or open compilers: they reify the metaobjects only during the compilation phase. The meta-programmer can modify and extend the compilation of the metaobjects (Lamping et al., 1992; Chiba, 1995; Tatsubori et al., 1999; Rodriguez, 1991). Once compiled, no meta-level information is available.
- Load-time metaobject protocols: the metaobject are reified right before being loaded into the execution environment (Chiba, 2000). This is for example useful to apply security policies to code that comes from an untrusted source (for example over the internet) or to adapt applications that were not meant to be deployed in the host environment.
- Run-time metaobject protocols: the metaobjects are also reified at run-time. This kind of metaobject protocols also provide powerful reflection support. (Cointe, 1987a; Kiczales et al., 1991; Paepcke, 1993)

The distinction between a compile-time and a load-time metaobject protocol is however less relevant in our situation because they both operate on source code. In the case of load-time metaobject protocols, bytecode or other intermediate representations can also be regarded as source code, especially when they contain symbolic information such as names of classes and methods.

In the next sections we will look at three different meta-programming systems. The first one, ObjVLisp (Cointe, 1987a), was among the first object systems where classes and methods are available as metaobjects and where the classes of the metaobjects are no different that base-level classes. The next one, OpenC++ (Chiba, 1995), is a compile-time metaobject protocol for C++. The last one, the CLOS metaobject protocol (Kiczales et al., 1991), has a similar class hierarchy to ObjVLisp but provides the programmer much more control over the semantics of the language. Because the CLOS metaobject protocol is more important for the rest of the text than the other two, it is discussed in a separate section (Section 3.3).

3.2.1 ObjVLisp

Cointe (1987b) presents an object model (based on Smalltalk) in which classes and instances are unified. There is only one kind of object in the entire model. The only thing that distinguishes a class from an object, is that a class is capable of creating new objects (it responds to the `new` selector). A metaclass is simply a class that instantiates other classes.

The ObjVLisp model is fully described by six postulates:

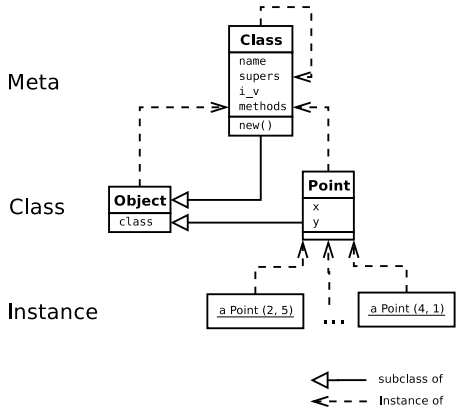

Figure 3.1: Architecture of the ObjVLisp model

1. Everything in the language is an object. An object represents a piece of knowledge (data) and a set of potentialities (procedures)
   `object = < data, procedures >`

2. The only control structure is message passing
   `(send object message args1 ... argsN)`

3. Every object belongs to a class that specifies its data (instance variables) and its behavior (methods). Objects will be dynamically created from this model and they are called instances of the class. All instances of a class have the same structure and shape, but they differ through the values of their instance variables.

4. A class is also an object, generated from a class, called a metaclass. Because of postulate three, each class has an associated metaclass which describes its behavior as an object. The initial primitive metaclass is the class `Class`, built as its own instance.

5. A class can be defined as a subclass of one or many other classes. This subclassing mechanism allows sharing of instance variables and methods and is called inheritance. The class `Object` represents the most common behavior shared by all objects.

6. Class variables of instances of the same class are instance variables of that class.
   `class variable [object] = instance variable [object's class]`

Figure 3.1 shows the basic architecture of the ObjVLisp model. There are actually only two classes: `Object` and `Class`. The class `Object` is the root of the inheritance tree. Everything in the model is an instance of `Object` or one of its subclasses. The metaclass `Class` is the root of the instantiation tree. There are two things special

about `Class`: it is a subclass of `Object`, what makes it a "normal" class as well - classes and metaclasses are only different in what kind of instances they produce - and it is an instance of itself. It must correctly describe its own structure. `Class` declares five instance variables and also has five values for those instance variables:

- `class = Class` (the class of the instance; inherited from `Object`)
- `name = 'Class'` (the name of the class)
- `supers = (Object)` (the list of superclasses)
- `i_v = (name, supers, i_v, methods)` (the list of instance variables)
- `methods = (new ...)` (the methods)

This model allows a programmer to define its own metaclasses and instantiate classes with it that have a different semantics than the instances of `Class`. Metaclasses may control:

- the algorithm used for method lookup
- the internal representation of objects by using different primitives for allocating objects in the `new` operation
- the access to methods; for example caching method lookup
- access to instance variable values by distinguish between private and public variables

3.2.2 Open C++

Chiba (1995) presents a metaobject protocol for C++, called Open C++. The goal of OpenC++ is to allow programmers to implement customized language extensions such as persistent or distributed objects, or customized compiler optimizations such as inlining of matrix arithmetic. These can be implemented as libraries and then used repeatedly. The advantage of providing control over the *compilation* of programs rather than over the run-time environment in which they execute is that there is no run-time speed or space overhead compared to ordinary C++. The design is based on the CLOS metaobject protocol (Kiczales et al., 1991) and on some previous compile-time metaobject protocol approaches (Rodriguez, 1991; Lamping et al., 1992; Chiba and Masuda, 1993).

As a motivating example, Chiba uses a simple linked list that should be made persistent. The persistency should be implemented as a library The persistency should happen transparently tough. A simple annotation to the class should be sufficient.

```
persistent class Node {
  public:
    Node* next;
    double value;
}

Node* get_next_of_next(Node* p) {
  Node* q = p->next;
  return q->next;
}
```

This little example defines a class `Node` with a value and a pointer to a next node. The simple function `get_next_of_next` takes a node as an argument and returns the node that is two places further in the linked list.

The keyword `persistent` in front of the class definition makes `Node` use the metaclass `PersistentClass`. This metaclass modifies the compilation of references to instance variables in order to verify whether the object is already loaded from the medium on which it is stored (and if not, load it). This is achieved as follows.

```
Expression
PersistentClass::CompileReadDataMember(
                    Environment env,
                    String member_name,
                    String variable_name) {
  return MakeParseTree(
            "(Load(%s), %e)",
            member_name,
            Class::CompileReadDataMember(
              ...));
}
```

It should be noted that the code in `CompileReadDataMember` *generates* code (under the form of a parse tree). Once all the code has been passed trough the OpenC++ compiler, only base-level C++ code remains.

3.3 CLOS Metaobject Protocol

Kiczales et al. (1991) define a metaobject protocol for CLOS. First they start with explaining how a CLOS implementation is built up. To do so, they implement a subset of CLOS (called Closette) in CLOS itself.

Basically, a CLOS program consists of `defclass`, `defgeneric` and `defmethod` forms. Behind the scenes, the execution of these forms creates internal representations of the classes, generic functions and methods. This information is used later on during method lookup, instantiation and initialization of objects, and so on. The execution of the defining forms and the processing of metaobjects is divided in three layers (the term "layer" does not refer to the layers in context-oriented programming).

- The macro-expansion layer provides the syntactic sugar that the user gets to see (for example the `defmacro`)
- The glue layer maps names to the metaobjects
- The functional layer provides all the run-time support for the language and deals directly with first-class metaobjects. This is where the behavior of classes, generic functions and methods is implemented. The metaobject protocol concentrates on this layer.

For example, the following definition

```
(defclass color-rectangle (color-mixin rectangle) (...))
```

macro-expands to

```
(ensure-class 'color-rectangle
  :direct-superclasses (list (find-class 'color-mixin)
                             (find-class 'rectangle))
  :direct-slots (list ...))
```

The functions `ensure-class` and `find-class` are from the glue layer. The implementation of these functions may create metaobjects or invoke operations of the lowest implementation layer.

The following example is the definition of the metaclass `standard-class` using ordinary CLOS code.

```
(defclass standard-class ()
  ((name :initarg :name
         :accessor class-name)
   (direct-superclasses :initarg :direct-superclasses
                        :accessor class-direct-superclasses)
   (direct-slots :accessor class-direct-slots)
   (class-precedence-list :accessor class-precendence-list)
   ...))
```

All user-defined classes (such as `color-rectangle` defined above) that do not specify otherwise, will be instances of this metaclass. Those instances will contain a name, a list of direct superclasses, a list of slots, a list of all superclasses (direct or indirect), etc.

The first step in exposing everything behind the scenes to the user is letting him inspect its own classes, generic functions and methods. This allows to implement for example a class browser and other tools in standard CLOS. Using this part of the metaobject protocol is called *introspection*. A program can look into its own structure, class hierarchy, methods, etc.

The next step is however the one that allows the user to modify and extend the language, called *intercession*. By using the standard object-oriented techniques of subclassing and specialization, the user can define its own metaclasses and create (and afterward use) a specialized language with a specialized implementation.

The following meta-program allows to define classes that monitor their slot access[1].

```
(defclass monitored-class (standard-class) ())

(defmethod slot-value-using-class :before
           ((class monitored-class) instance slot-name)
  (note-operation instance slot-name 'slot-value))
```

This meta-program defines a new metaclass by subclassing `standard-class` and adds a before method to the generic function of the metaobject protocol that is responsible for getting the value of a slot. Base-level classes that have `monitored-class` as their metaclass will use this semantics for their slot-access.

Functional and Procedural Protocols

In order to guarantee that every slot operation of `monitored-class` in the previous example is notified, it is essential that it is specified in the slot access protocol that `slot-value-using-class` is called *each and every time* that a slot is accessed.

This requirement contrasts with another desirable property of meta-level operations: memoizability. For example, `compute-class-precedence-list` should *not* be invoked every time that the class precedence list is needed. Instead, it should only be invoked the first time when the information is needed. Each subsequent time, the same value can be reused.

[1] Only part of the meta-program is given. In order to modify slot access properly, three other slot operations have to be adapted, among others one for slot assignment.

According to Kiczales et al. (1991), the first paragraph describes a *procedural* protocol and the second paragraph a *functional* protocol.

In a functional protocol, such as `compute-class-precedence-list`, a function is called to compute a result, which is then used by other parts of the system in order to produce the intended behavior. In other words, calling a functional protocol can affect the behavior of a system, but is does not produce that behavior directly.

Because of this, certain restrictions can be placed on the result that a functional protocol should produce. The specification will typically place limitations or requirements on the result of a functional protocol in order to ensure memoizability. For example, the class precedence list of a class should not change after it is computed with `compute-class-precedence-list`. This assures that when the precedence list is needed, the previously computed list can be used again instead of recomputing it. Moreover, because the result has to remain constant, other parts of the system can take advantage of that knowledge. For example, the positions of all the slots (both direct and inherited slots) can be precomputed when the class-precedence list is known.

Procedural protocols, such as `slot-value-using-class`, are called to perform some action and to directly produce some part of the total system behavior. The specification of a procedural protocol will typically place fewer restrictions on the activity of the function, but put more restrictions on when it is to be invoked.

Because the effect of a procedural protocol is direct, such protocols tend to put more power in the hands of the programmer. Their results can however not be memoized.

3.4 Context-Dependent Meta-Behavior

Now we examine how context-dependent behavior at the meta-level can be defined by using a metaobject protocol.

As discussed in the previous section, metaobjects form the base program and the metaclasses define the semantics of the metaobjects. In order to choose the semantics based on the context, there are two concepts that are important:

Association of a metaobject with its metaclass. For example, a generic function that represents some base-level operation has its associated metaclass that defines how the generic function invocation works.

Composition of metaclasses. Since metaclasses are built with the same language constructs as the base level classes, they can use the same composition techniques. For example, combining two metaclasses in CLOS can be done by creating a

new metaclass that inherits from both metaclasses. Metaobjects associated with the new metaclass will then use the composition of the two superclasses.

So in order to modify the semantics of the metaobjects based on the context, there are two options:

- Allow metaobjects to change their associated metaclass at run-time and possibly make new compositions of metaclasses at run-time.
- Express the context dependencies in the implementation of the metaclasses and leave the association and composition fixed.

Based on the requirements for context-oriented programming discussed in Section 2.2, the first option can be eliminated immediately because the modifications would not be thread-local. Changing the metaclass of a metaobject is a global operation that would influence other threads as well. The second option however, will suffer from the same problems as those described in Section 2.1.2. For example, the behavior of a language extension could be wrapped in `if` statements that check whether some additional code should be applied or not. This indicates that context-oriented programming should be applied at the meta-level.

3.5 Summary

We started this chapter with a general definition of meta-programming. We continued with demonstrating how a metaobject protocol offers meta-programming facilities to a language in such a way that the programmer can modify and extend the language by defining new metaclasses. Because the CLOS metaobject protocol is more important for the rest of the text, this was explained in more detail.

Finally we considered how a metaobject protocol can be used to define context-dependent meta-behavior. The first option to modify the association and composition of metaobjects at run-time was eliminated because this would fail to meet the requirement of thread-locality. This means that context-oriented programming, presented in Chapter 2, should be used on the metaobject protocol itself. This is explored in Chapter 5. In order to do so, we have created an experimental environment, presented in Chapter 4.

Chapter 4

TinyContext

In this chapter we present TinyContext, an experimental environment that will be used in Chapter 5 to apply context-oriented programming at the meta-level.

TinyContext is based on TinyCLOS (Kiczales, 1992), a simple object-oriented programming language implemented in Scheme. More information about TinyCLOS can be found in Appendix B. In Section 4.1 we go over the basic features of TinyCLOS in order to make the reader familiar with TinyCLOS code.

In Section 4.2 we present TinyContext, our extension of TinyCLOS with explicit support for context-oriented programming. The features of TinyContext are a subset of those found in ContextL, presented in Section 2.3.

4.1 TinyCLOS

In this section we demonstrate TinyCLOS, the basis for TinyContext. The first subsection shows how classes can be created and used. The second subsection introduces the concept of generic functions to define behavior on classes.

TinyCLOS is a simple object-oriented language implemented in Scheme. According to the author (Kiczales, 1992), it is a "kernelized" version of the Common Lisp Object System (CLOS). There are a lot of features in CLOS that are not present in TinyCLOS, but the meta-programming facilities of TinyCLOS allow the programmer to define them himself. TinyCLOS and its metaobject protocol are similar to Closette, the language defined in (Kiczales et al., 1991) to introduce metaobject protocols.

The reason for choosing TinyCLOS over CLOS or Closette is twofold. The goal of TinyContext is to allow to use context-oriented programming at the meta-level, that is,

inside the metaobject protocol. This means that (parts of) the metaobject protocol will have to be redefined. Using the CLOS metaobject protocol for this would introduce a much higher degree of complexity. Much of these complexities would be of a practical nature and would not be relevant for our work. The second reason is that working in Common Lisp is more complex than working in Scheme. This is one of the reasons that TinyCLOS was created in the first place.

4.1.1 Classes

Consider the following class definition in TinyCLOS.

```
(define <person>
  (make-class (list <object>)    ; Direct superclasses
              '(name address)))  ; Direct slots
```

In this example, the class `<person>` is defined. In TinyCLOS, the name of a class is by convention wrapped in angle brackets: < ... >. It has one superclass (like CLOS, TinyCLOS supports multiple inheritance), the predefined class `<object>`, and two instance variables, called *slots*: a name and an address. The following example defines the class `<student>`, a subclass of `<person>`.

```
(define <student>
  (make-class (list <person>)    ; Direct superclasses
              '(school)))        ; Direct slots
```

A student is a person with an extra slot for his school. The class `<student>` has one *direct* superclass and one *direct* slot, but during the initialization of the class, the list of all *indirect* superclasses is calculated as well. The final list of slots for `<student>` will then be the combination of all the direct slots of all the superclasses.

As demonstrated in the next example, creating an instance of a class is done through `make`. Its first argument is the class of which it has to create an instance. The rest of the arguments are initialization values for the slots.

```
(define stijn
  (make <student>
        'name "Stijn Timbermont"
        'address "Lokerenbaan 166, 9240 Zele"
        'school "Vrije Universiteit Brussel"))
```

4.1.2 Generic Functions

In TinyCLOS, classes only define state and no behavior. Methods are not associated with classes but with *generic functions*. The following example demonstrates how a generic function can be created.

```
(define display-person (make-generic))
```

Generic functions are created with `make-generic`, without any arguments. They can be invoked just like ordinary Scheme functions and methods can be added to them. When just created, invoking a generic function will produce an error message because there are no methods to choose from.

So if we want the previously created generic function to do something useful, we have to add methods to it. This is demonstrated in the following example.

```
(add-method display-person
  (make-method (list <person>)
               (lambda (call-next-method object)
                 (display "Name: ")
                 (display (slot-ref object 'name))
                 (newline))))
```

This examples does two things at once: create a method and then add it to the generic function `display-person`. Creating a method requires two parameters: a list of *specializers* and a procedure with the actual method body. When a generic function is invoked, it will choose a method by matching the types of the arguments with the specializers in order to find the *most specific* method and apply its body to the arguments. In other words, the methods whose specializers match best with a list of actual parameters will be executed when a generic function is invoked.

In the example above, the body of the method has a special parameter called `call-next-method`. Invoking this procedure can be compared to a super send in an object-oriented programming language that uses message passing. When a generic function is invoked, not only the best matching, but all the applicable methods are computed. The one that matches best is invoked first, but if that method invokes `call-next-method`, the second best is invoked. If there are no more methods left, invoking `call-next-method` will raise an error.

The following transcript demonstrates how the class and the generic function that we defined above can be used.

```
> (define stijn
    (make <student>
          'name "Stijn Timbermont"
          'address "Lokerenbaan 166, 9240 Zele"
          'school "Vrije Universiteit Brussel"))
> (display-person stijn)
Name: Stijn Timbermont
```

4.2 TinyContext

In this section we introduce TinyContext, our extension of TinyCLOS with support for context-oriented programming. The constructs described in this section are based on ContextL, presented in Section 2.3.

First we recapitulate the concepts of context-oriented programming that were introduced in Section 2.2.

Grouping in Layers Context-dependent behavior is grouped in layers. Each layer represents a particular case and can group definitions for different classes.

Dynamic Activation Based on context information, the programmer can activate a layer and execute some code in the context of that layer. Both code that is called directly and code that is called indirectly should use the definitions of the activated layer.

Thread Locality Activating a layer in one thread should not interfere with other threads. Each thread has its own context of execution and activation of a layer in one thread should not automatically activate it in any other threads.

4.2.1 Layers

Layers are the primary concept of context-oriented programming. A layer is basically an identity with no further properties of its own. However, new definitions can be placed in a layer by explicitly referring to one.

Layers can be created with `make-layer`, just like this.

```
(define full-info-layer (make-layer))
```

Layers can be activated in the control flow of a running program, as follows.

```
(with-layer full-info-layer
  ... contained code ...)
```

This way of layer activation has the effect that the layer is only active during execution of the *contained code*, including all the code that the *contained code* calls directly or indirectly. During this execution, all definitions of the activated layer become part of the running program. For example, a method defined in `full-info-layer` will now be taken into account for method dispatch. When control flow returns from the layer activation, the layer is automatically deactivated again.

Layer activation can be nested. In that case all active layers are combined and all the definitions of the activated layers become part of the running program. The definitions of the last layer that was activated have precedence over the definitions of previously activated layers. For example, if two methods with the same signature, but defined in different layers become part of the running program, method dispatch will give precedence to the one defined in the layer that was activated last. In short we say that the last activated layer has precedence over all other active layers. When an already activated layer is activated again, it just gains precedence. The layer is only deactivated when control flow returns from the *first* time that that layer was activated.

4.2.2 Layered Functions and Methods

In the previous sections we have define the classes `<person>` and `<student>`, the generic function `display-function`, one method for `display-function` and the layer `full-info-layer`. Now it is time to define some methods in that layer. This can be achieved by defining a *layered method*. The only difference with a normal method is that it names a layer where the definitions has to be placed in. In fact, normal methods are layered methods placed in the root layer. Therefore, the term *layered method* will only be used for methods defined in another layer than the root layer.

In TinyContext, layered methods can be added to any generic function (the generic function does not have to be defined with some special construct). Therefore, we use the term *layered function* to stress the fact that there are layered methods added to the generic function, or they will be added later on, even tough there is no technical difference between a layered function and a generic function.

The following example defines a layered method in `full-info-layer` and adds it to `display-person`. The purpose of this method is to also print the school of a student, but only when `full-info-layer` is active.

```
(add-method display-person
```

```
            (make-layered-method
              full-info-layer
              (list <student>)
              (lambda (call-next-method object)
                (call-next-method)
                (display "School: ")
                (display (slot-ref object 'school)))))
```

As long as `full-info-layer` is not active, this method definition will not be taken into account for method dispatch and displaying a student will just display his name. When `full-info-layer` is activated, this method becomes part of the running program and displaying a student will then use this method and display not only his name but also his school. This is demonstrated in the following transcript.

```
> (display-person stijn)
Name: Stijn Timbermont
> (with-layer full-info-layer
    (display-person stijn))
Name: Stijn Timbermont
School: Vrije Universiteit Brussel
```

In the example above, the new method has a different list of specializers from the method that was already present. When `full-info-layer` is active, normal method dispatch can be used to choose between the methods. However, it is possible to add methods in different layers with the same specializers list. In that case, method dispatch will use the layer precedence rule to choose which method to invoke.

Consider the following example.

```
(add-method display-person
            (make-layered-method
              full-info-layer
              (list <person>)
              (lambda (call-next-method person)
                (call-next-method)
                (display "Address: ")
                (display (slot-ref person 'address))
                (newline))))
```

In this example, a layered method is defined in `full-info-layer` and the argument `person` is specialized on `<person>`. There is already such a method defined in the root layer, so layer precedence will be used in method dispatch.

The new layered method will display not only the name of a person, but also his address. The following transcript shows the total effect of the method in the root layer and the two method in `full-info-layer`.

```
> (with-layer full-info-layer
    (display-person stijn))
Person
Name: Stijn Timbermont
Address: Lokerenbaan 166, 9240 Zele
School: Vrije Universiteit Brussel
```

4.2.3 TinyContext versus ContextL

The main difference between TinyContext and ContextL is that ContextL is a language *extension* while TinyContext is a *modification* of the default language. ContextL uses the metaobject protocol of CLOS to define new metaclasses and override the operations defined on them. For example, a layered function in ContextL is an instance of a new metaclass (a subclass of the standard generic function metaclass) that changed method lookup in order to take into account which layers are active and which are not. In TinyCLOS, the notion of layered functions is "pushed into the language". Now the metaobject protocol is not used, but modified. The reason for this is that this allows to use context-oriented programming in the metaobject protocol itself. This technique will be used in the next chapter.

Another difference is that ContextL has more features than TinyContext. TinyContext only supports context-dependent behavior (layered functions) and no context-dependent state (layered classes).

4.3 Summary

In this chapter we presented TinyContext, the experimental environment that will be used in the rest of this dissertation. The goal of TinyContext is to allow us to apply context-oriented programming at the meta-level. TinyContext allows us to do so because the language constructs for context-oriented programming are already available into the default language (unlike ContextL, that is an *extension* of CLOS).

TinyContext is based on TinyCLOS, a "kernelized" version of CLOS. TinyCLOS has a lot less features, but the metaobject protocol of TinyCLOS is powerful enough to allow the programmer to add them when necessary. More information about TinyCLOS can be found in Appendix B.

TinyContext will be used in the next chapter to apply context-oriented programming on the metaobject protocol. This is possible because in TinyContext, every generic function is a layered function as well. This is not the case in ContextL, where only generic functions that are defined with a special construct can be layered.

Chapter 5

Context-Oriented Meta-Programming

In this chapter we combine context-oriented programming presented in Chapter 2 and meta-programming presented in Chapter 3 by applying context-oriented programming on the metaobject protocol. TinyContext, the experimental environment presented in Chapter 4 is used for this. TinyContext is based on TinyCLOS, so the metaobject protocol of TinyCLOS is used.

The meta-programming facilities of the TinyCLOS metaobject protocol allow the programmer to change and extend the semantics of the base language by defining new metaclasses and overriding the operations on them. For instance, the metaclasses define how generic functions are applied and how the slots of an object are accessed. However, the operations defined for the metaclasses are all used during the *creation* of the metaobjects and not when the metaobjects are used during the execution of the actual base-level program. Consider slot access and generic function invocation.

- The getters and setters for the slots of an object are created when the class is defined. Accessing or assigning a slot of an object will then use the procedures that were created during the definition of that class.
- Each generic function has its own apply function, responsible for method dispatch. It is (re)computed every time a method is added to the generic function. The metaobject protocol performs a partial evaluation by examining the available methods. Invocation of a generic function are forwarded to its apply function.

In other words, the protocols for slot access and generic function invocation are all functional protocols.

TinyContext already provides constructs for defining context-dependent behavior at the base-level. More concretely, layered methods can be added to existing layered functions. If the context-dependent behavior is to be added to an ordinary function, it has to be converted into a layered function first. This adds the extra indirection of method dispatch.

In order to make the meta-level operations such as slot access and generic function invocation context-dependent, they have to be converted to layered functions as well. In order to limit the overhead of this extra indirection, only the operations that need to be layered should be converted.

Section 5.1 shows how to define context-dependent meta-level behavior by defining new metaclasses that use the constructs for context-oriented programming. This will allow us to use layer activation to change slot access and generic function invocation depending on the context of the running program.

In Section 5.2 we will use context-oriented programming on the generic functions of the metaobject protocol itself. This allows to change the initialization of metaobjects by changing the context in which they are defined. In combination with redefinition of metaobjects, this allows to modify the metaobjects at run-time.

The context-dependent meta-behavior must still be anticipated by the programmer: he has to use the right metaclasses when defining the base-level classes and methods. Section 5.3 will show how using context-oriented programming for the operations of the metaobject protocol itself can add the necessary hooks to the meta-objects during the execution of the base program.

5.1 Context-Dependent Meta-Behavior

In this section we will provide constructs that allow the use of context-oriented programming for slot access and generic function invocation. This will be accomplished by normal meta-programming techniques with a metaobject protocol: we define new metaclasses and override some operations defined on them.

The goal is to turn some of the procedures created by the metaobject protocol during the definition of the classes and generic functions into layered functions. This will not have an immediate effect on the base program, but it allows us to add layered methods to those meta-level operations.

5.1.1 Layered Apply Function

Every generic function has its own apply function. It is an ordinary procedure that takes all the arguments and uses their types to choose the method it has to invoke. It

is (re)computed by the metaobject protocol every time that a method is added to the generic function. The following meta-program defines a new metaclass and overrides the generic function of the metaobject protocol that is responsible for creating the apply function. The actual behavior is not changed, but is wrapped in a generic function.

```
(define <layered-apply-generic>
  (make <entity-class>
        'direct-supers (list <generic>)))

(add-method compute-apply-generic
  (make-method (list <layered-apply-generic>)
               (lambda (cnm generic)
                 ;; convert the result of (cnm)
                 ;; to a layered function
                 ...)))
```

The new metaclass is called `<layered-apply-generic>`. Generic functions that are created with this metaclass will have an apply function that is not a normal procedure but a layered function. The generic function of the metaobject protocol that is responsible for creating the apply function is called `compute-apply-generic`. It is overridden for the new metaclass in order to turn the actual apply function into a layered function.

We can define a generic function with this new metaclass as follows.

```
(define double
  (make <layered-apply-generic>))

(add-method double
  (make-method (list <number>)
               (lambda (cnm n)
                 (* n 2))))
```

At first sight, this generic function will behave exactly the same way as standard generic functions. It can be invoked as a standard generic function and it has the same result.

```
> (double 3)
6
```

The difference with a standard generic function is that the meta-level operation for applying the generic function is now a layered function instead of an ordinary procedure.

Suppose a function `(log generic name)` that takes two arguments: a generic function that has to be logged and the name of the function that has to be used for logging invocations. The function `log` will add a layered method for the `logging` layer to the apply function of the generic function.

```
(log double 'double)
```

Invoking `log` on `double` has the effect that when the `logging` layer is not active, double will behave as normal, but when the `logging` layer is active, all invocations of `double` will be printed to the screen. This is demonstrated in the following transcript.

```
> (double 3)
6
> (with-layer logging
    (double 3))
(double 3)
(double 3) => 6
6
```

This means that we can actually change the semantics of invoking `double` during the execution of a program, by activating a layer. Logging invocations is not a very exciting example, but it makes it easy to demonstrate the effect. Chapter 6 gives a more useful application of these constructs.

5.1.2 Layered Slot Access

Changing slot access is in many ways similar to changing generic function invocation. It also involves defining a new metaclass, but this time one for creating classes instead of generic functions. When defining a class, the metaobject protocol creates getters and setters for each slot of the new class. By overriding the generic function of the metaobject protocol that is responsible for this, the semantics of accessing a slot can be changed. Automatic persistency is a common application of this feature.

In order to make slot access context-dependent, the procedures have to be converted to layered functions. This is accomplished with the following meta-program, similar to the one for layered apply functions.

```
(define <layered-getters-n-setters-class>
  (make-class (list <class>)
              (list)))
```

```
(add-method compute-getter-and-setter
  (make-method (list <layered-getters-n-setters-class>)
               (lambda (call-next-method class slot allocator)
                 ;; convert the actual getter and setter
                 ;; to layered functions
                 ...)))
```

Classes defined with metaclass `<layered-getters-n-setters-class>` will have getters and setters that are wrapped in layered functions. This has no immediate effect, as demonstrated in the following example.

```
(define <point>
  (make <layered-getters-n-setters-class>
        'direct-supers (list <object>)
        'direct-slots (list 'x 'y)))

(define p (make <point>))

> (slot-set! p 'x 3)
> (slot-ref p 'x)
3
```

This example defines a class `<point>` with one superclass `<object>` and two slots `x` and `y`. Accessing and assigning a slot can be done with respectively `slot-ref` and `slot-set!`. The operations simply invoke the right getter or setter defined in the class of the objects.

In the previous section we described a function that adds logging support to a generic function. In the same way, we can define a function that lets slot access be printed on the screen if the `logging` layer is active. The function `(log-getter-n-setter class slot)` does exactly that: it adds layered methods to the getter and setter for accessing `slot` in instances of `class`. The following transcript demonstrates this for the slot `x` of the class `<point>`.

```
> (log-getter-n-setter <point> 'x)
> (with-layer logging
    (slot-ref p 'x))
getting slot 'x' => 3
3
> (with-layer logging
```

```
    (slot-set! p 'x 4))
setting slot 'x' to '4'
> (slot-ref p 'x)
4
```

This transcript clearly shows that we can change the semantics of slot access by activating a layer. When `logging` is activated, every slot access of `x` is printed on the screen. When `logging` is not active, the old slot access is used. This example demonstrates logging, but other (more useful) changes in semantics are possible.

5.2 Layered Metaobject Protocol

If a programmer defines some classes and generic functions, without mentioning metaclasses, the metaobject protocol is still implicitly used. Consider the following example.

```
(define <point>
  (make-class (list <object>)
              (list 'x 'y)))
```

In this class definition, no metaclass is mentioned. The class `<point>` is intended to use the *default* semantics. The definition above is equivalent to the following definition.

```
(define <point>
  (make <class>
        'direct-supers (list <object>)
        'direct-slots (list 'x 'y)))
```

Now the metaclass is explicitly mentioned. The metaclass `<class>` is defined by TinyCLOS, but is conceptually at the same level as user-defined metaclasses. In the same way, `make-generic` uses the predefined `<generic>` and `make-method` uses `<method>`. All the operations of the metaobject protocol that can be overridden for user-defined metaclasses have a default implementation for these metaclasses.

In TinyCLOS, the only way to override the operations of the metaobject protocol is by defining a new metaclass and define new methods for the generic functions of the metaobject protocol. In TinyContext however, the generic functions of the metaobject protocol are also layered functions. By adding layered methods to these generic functions, it is possible to change the *default* semantics of the language, depending on the context.

5.2.1 Generic Function Invocation

Consider the example of implicitly forcing the arguments of an invocation. A promise is a delayed computation. Forcing a promise will execute the actual computation. Defining a metaclass for generic functions that automatically force their arguments can be done in TinyCLOS as follows.

```
(define <forcing-generic>
  (make <entity-class>
        'direct-supers (list <generic>)))

(add-method compute-apply-generic
  (make-method (list <forcing-generic>)
               (lambda (cnm generic)
                 (let ((original (cnm)))
                   (lambda args
                     (apply original
                            (map (lambda (arg)
                                   (if (promise? arg)
                                       (force arg)
                                       arg))
                                 args)))))))
```

This example defines a metaclass `<forcing-generic>`. Instances of this metaclass are generic functions that have a different semantics from standard generic functions. Before method dispatch, the generic function will loop over its arguments, check for promises among them and force them.

The following transcript shows the use of this metaclass.

```
(define double (make <forcing-generic>))
(add-method double
            (make-method (list <number>)
                         (lambda (cnm n)
                           (* n 2))))

> (double (delay (+ 1 2)))
6
```

In this example, the computation of `(+ 1 2)` is delayed. The argument passed to `double` is not `3` but a promise of a calculation that has not been executed yet. How-

ever, because the apply function of `double` forces all of its arguments, `double` executes as expected.

In TinyContext, implicit forcing can also be defined in a layered method that is added to `compute-apply-generic`, as follows.

```
(define implicit-forcing (make-layer))

(add-method compute-apply-generic
            (make-layered-method implicit-forcing
                                 (list <generic>)
                                 (lambda (cnm generic)
                                   ;; Same body
                                   ...)))
```

In this example, the same extension in semantics is defined. The difference is that now it is not for a different metaclass, but for a different layer. If the `implicit-forcing` layer is active, the default semantics of generic function invocation will include the forcing of arguments.

The apply function of a generic function is computed each time a method is added. This means that by adding a method when `implicit-forcing` is active, an apply function that forces the arguments will be created. This is demonstrated in the following transcript.

```
(define double (make-generic))

(with-layer implicit-forcing
  (add-method double
              (make-method (list <number>)
                           (lambda (cnm n)
                             (* n 2)))))

> (double (delay (+ 1 2)))
6
```

Note that now the generic function `double` has the standard metaclass `<generic>`. Adding a method will invoke `compute-apply-generic` and because the `implicit-forcing` layer is active, our new definition will be used.

It is still inconvenient that we have to add a method in order to change the default semantics. It would make much more sense to just recompute the apply function of a

generic function. The reason that there is no such function defined in the TinyCLOS metaobject protocol, is because such a function is useless if the result only depends on the generic function and its internal state (the list of methods). However, by applying context-oriented programming on the metaobject protocol itself, the semantics of the language can depend on the context in which the meta-objects are defined. This makes it useful to add a function that simply recomputes the apply function of a generic function because the computation can be context-dependent.

Based on this, we can use the full power of a layered metaobject protocol. Consider the following function definition.

```
(define (implicitly-force generic)
  (with-layer implicit-forcing
    (recompute-apply-generic generic)))
```

This function will just activate the `implicit-forcing` and recompute the apply function of a generic function. This function can be used on any generic function with the effect that the old value is wrapped in an apply function that forces the arguments.

5.2.2 Slot Access

The same technique can be applied on slot access Changing the semantics for slot access is achieved by overriding `compute-getter-and-setter`, a generic function defined in the metaobject protocol, that is called for each slot of a class when the class is defined. Adding a layered method to `compute-getter-and-setter` will change the default semantics for slot access for classes that are defined when the layer is active.

It also makes sense to provide a function that recomputes the getters and setters of a class, similar to the function for recomputing the apply function of a generic function. This allows us to change the slot access of any class, even after it is defined.

5.3 Unanticipated Context-Dependencies

Section 5.1 showed how metaclasses that use the constructs for context-oriented programming can be used to define context-dependent behavior at the meta-level. Section 5.2 showed how applying context-oriented programming to the metaobject protocol itself gives the possibility to change the semantics of meta-objects after they were created, by recomputing the operations they define in different layers. Now we can combine these two features to be able to express context-dependent meta-behavior that was

not anticipated in the base-level program. This will result in the constructs that will actually be used by the meta-programmer to define context-dependent meta-behavior.

5.3.1 Generic Function Invocation

Consider the `log` function of section 5.1.1: it has the requirement that the generic function that has to be logged must be an instance of `<layered-apply-generic>`. However, as we have seen in section 5.3.1, the meta-level semantics can be changed for any generic function by redefining the generic functions of the metaobject protocol in a layer. Combining those two features, we can define a function `(ensure-layered-apply generic)` that ensures that `generic` has a layered apply function. Then it is straightforward to define the utility function `specialize-apply-function` also adds a layered method to the layered apply function.

The following code shows the pattern that can be used to express unanticipated context-dependent meta-behavior.

```
(define x (make-layer))

(define (y generic ...)
  (specialize-apply-function generic
                             x
                             (lambda (cnm . args)
                               ...)))
```

In this pattern, `y` is a function that will ensure that `generic` has a layered apply function and then add a layered method for the layer `x` to it. `specialize-apply-function` takes three arguments: the actual generic function, the layer in which the meta-behavior is to be defined and a procedure with the actual meta-behavior, used to create the layered method.

Now we can show the real implementation of the `log` function described in the previous section.

```
(define logging (make-layer))

(define (log generic name)
  (specialize-apply-function generic
                             logging
                             (lambda (cnm . args)
                               (log-invocation name
```

```
                                                    cnm
                                                    args))))
```

Invoking `log` on any generic function ensures that it has a layered apply function and adds a layered method for logging invocations to it. The function `log-invocation` will print the logging information to the screen before and after the invoking the second parameter (which will perform the actual computation).

5.3.2 Slot Access

Similar constructs are provided to define unanticipated context-dependencies in the access and assignment of slots. The pattern for defining semantics for getters is as follows.

```
(define x (make-layer))

(define (y class slot-name ...)
  (specialize-getter class
                     slot-name
                     x
                     (lambda (cnm object)
                       ...)))
```

The pattern for defining semantics for setters is as follows.

```
(define x (make-layer))

(define (y class slot-name ...)
  (specialize-setter class
                     slot-name
                     x
                     (lambda (cnm object new-value)
                       ...)))
```

5.4 Summary

In this chapter we combined context-oriented programming presented in Chapter 2 and meta-programming presented in Chapter 3 by applying context-oriented programming

on the metaobject protocol. TinyContext, the experimental environment presented in Chapter 4 was used for this.

We started in Section 5.1 with adding the run-time hooks that we need to adapt slot access and generic function invocation to the context. This was done by converting the responsible protocols from functional into procedural protocols. Instead of ordinary procedures, they become layered functions. By adding layered methods to these layered functions, generic function invocation and slot access can effectively be modified at run-time by activating a layer.

The next step was to use context-oriented programming directly on the generic functions of the metaobject protocol. This allows to define meta-programs in layers instead of in new metaclasses. The metaobject protocol is invoked when metaobjects are created and initialized, so this means that their semantics can now depend on the context in which they are defined. In combination with run-time redefinition of metaobjects, this means that the behavior of specific metaobjects can be altered at run-time, without actually changing their metaclass.

Section 5.3 combined Sections 5.1 and 5.2 in order to create general purpose language constructs to define context-dependent meta-behavior.

Chapter 6

Examples

In this chapter we demonstrate the constructs presented in the previous chapter in more detail. First we start with combining logging invocations and caching result values. This simple example allows us to clearly see what happens, especially when combining two layers and activating them in different order.

The next example demonstrates a change in the semantics of function application such that algorithms that deal with lists can be adapted to deal with infinite lists as well.

The third example shows how a language construct to intercept patterns in the history of execution of a program can benefit from context-oriented programming. By returning a layer that contains layered methods on meta-level operations, the language construct can directly be used to define context-dependent behavior as well.

6.1 Logging and caching

To illustrate the previous chapter we will demonstrate how the constructs can be used. In the following paragraphs we elaborate on the logging example of the previous chapter. Next to that, we will provide means to build a cache of the results of previous invocations. Then we will apply and combine these two extensions to the `fibonacci` function, implemented with tree-recursion, as follows.

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1))
         (fib (- n 2)))))
```

Since the constructs cannot be used for ordinary Scheme functions, we have to replace them with an equivalent generic function. That can be achieved with `ensure-generic-function`: generic functions are returned immediately, but ordinary procedures are used to create a generic function with a default method that calls the original procedure.

```
(set! fib (ensure-generic-function fib))
```

If it is not desired to replace the original functions, it is of course possible to define a new one.

```
(define new-fib (ensure-generic-function fib))
```

Note that this step can be compared to the `redefdynfun` construct (Costanza, 2003) described in section A.2.3. Replacing an ordinary function by a generic function does not change the behavior immediately, but it adds the hooks that are needed for context-oriented meta-programming. For instance, now it is possible to use the `log` function described in the previous chapter on `fib`. This is demonstrated in the following transcript.

```
> (log fib 'fib)
> (fib 4)
3
> (with-layer logging
    (fib 4))
(fib 4)
|  (fib 3)
|  |  (fib 2)
|  |  |  (fib 1)
|  |  |  (fib 1) => 1
|  |  |  (fib 0)
|  |  |  (fib 0) => 0
|  |  (fib 2) => 1
|  |  (fib 1)
|  |  (fib 1) => 1
|  (fib 3) => 2
|  (fib 2)
|  |  (fib 1)
|  |  (fib 1) => 1
|  |  (fib 0)
|  |  (fib 0) => 0
```

```
|  (fib 2) => 1
(fib 4) => 3
3
```

Invoking `log` on `fib` adds support for logging to `fib`. For the first invocation of `fib`, the `logging` layer is not active, so the invocations are not logged. In the second invocation, the `logging` layer is active. Now the logging will take place. Because the Fibonacci function is implemented with tree-recursion, not only one invocation is logged, but all of them. The functions responsible for the actual logging keep track of the nesting depth of the recursion in order to represent the recursion clearly on the screen.

The next step is to define a function that will save the arguments and the result of a invocation into a cache, in order to look it up for later invocations. Note that the pattern described in Section 5.3.1 is used again.

```
(define caching (make-layer))

(define (cache generic)
  (specialize-apply-function
    generic
    caching
    (lambda (cnm . args)
      (lookup generic args
                      (lambda ()
                        (let ((result (cnm)))
                          (save generic args result)
                          result))))))
```

The function `lookup` takes three arguments: the generic function, the arguments and a procedure that is called of there is no entry in the cache for the generic function and the arguments. This means that the actual computation only takes place if the result is not yet in the cache, and because it is then immediately added to the cache, subsequent invocations with the same arguments will not be recomputed.

Adding caching support to `fib` goes as follows.

```
(cache fib)
```

Note that the caching only takes place when the `caching` layer is activated. Together with logging, we can now clearly see what happens and also examine the effect of nested layer activation and layer precedence.

There are two combinations possible:

- First logging is activated, then caching is activated:

```
(with-layer logging
  (with-layer caching
    (fib 4)))
```

This means that caching has precedence, and only when the method for the caching layer invokes `call-next-method`, the logging will take place. This results in the following output:

```
(fib 4)
|  (fib 3)
|  |  (fib 2)
|  |  |  (fib 1)
|  |  |  (fib 1) => 1
|  |  |  (fib 0)
|  |  |  (fib 0) => 0
|  |  (fib 2) => 1
|  (fib 3) => 2
(fib 4) => 3
3
```

It is clear that every time `fib` is invoked and the result is found in the cache, the invocation is not logged.

- First caching is activated, then logging is activated:

```
(with-layer caching
  (with-layer logging
    (fib 4)))
```

Now logging has precedence, so first the invocation will be logged and after that the result is lookup up in the cache (and calculated if not found). This means that in the output there are more invocations visible[1]:

[1]The cache was cleared before each example. Otherwise, this invocation would have returned immediately because the result was already in the cache.

```
(fib 4)
|  (fib 3)
|  |  (fib 2)
|  |  |  (fib 1)
|  |  |  (fib 1) => 1
|  |  |  (fib 0)
|  |  |  (fib 0) => 0
|  |  (fib 2) => 1
|  |  (fib 1)
|  |  (fib 1) => 1
|  (fib 3) => 2
|  (fib 2)
|  (fib 2) => 1
(fib 4) => 3
3
```

These examples showed that the order in which the layers are activated has an effect on the behavior, especially in this case, because caching does not always invoke the next method.

6.2 Delay / Force

Functional languages often provide a construct for *delaying* a computation, resulting in a *promise* that can be *forced* in order to obtain the actual value. Because a promise is often forced several times, its result is only computed once. Subsequent forcing of the promise simply returns the previously computed result. Consider the following transcript in Scheme

```
> (define p (delay (begin (display "computing p")
                          (+ 1 2))))
> p
#<struct:promise>
> (force p)
computing p
3
> p
#<struct:promise>
> (force p)
3
```

`p` contains a promise that, when forced, will print a message and return the sum of one and two. After being forced, `p` still contains a promise, but when forced again, the message is no longer printed: the previously computed result is returned.

In this example we will provide constructs for implicitly delaying computations and forcing arguments, depending on the context of execution.

Implicit delaying When a function is implicitly delayed, invoking it will return a promise instead of immediately computing the result. Note that the evaluation of the arguments is not delayed.

Implicit forcing An implicitly forcing function looks for promises in its arguments and forces them.

Implicit delaying is defined with `specialize-apply-function` as follows.

```
(define implicit-delaying (make-layer))
(define (implicitly-delay generic)
  (specialize-apply-function generic
                             implicit-delaying
                             (lambda (cnm . args)
                               (delay (cnm)))))
```

The function `implicitly-delay` will add support for delaying to the generic function that is passed to it. When the generic function is then called, a promise will be returned only if the `implicit-delaying` is active. Otherwise, the result is computed as normal. This is demonstrated in the following transcript.

```
> (define double
    (ensure-generic-function
     (lambda (n)
       (* n 2))))
> (implicitly-delay double)
> (double 5)
10
> (with-layer implicit-delaying
    (force (double 5)))
10
```

This examples defines a function that takes a number and returns the double. `ensure-generic-function` transforms it into a generic function and adds support for implicit delaying. On the

first invocation, `implicit-delaying` is deactivated and `(double 5)` immediately returns `10`. On the second invocation, the `implicit-delaying` is activated so `(double 5)` returns a promise that has to be forced.

Along the same lines, implicit forcing of the arguments can be defined.

```
(define implicit-forcing (make-layer))
(define (implicitly-force generic)
  (specialize-apply-function
   generic
   implicit-forcing
   (lambda (cnm . args)
     (apply cnm (map (lambda (arg)
                       (if (promise? arg)
                           (force arg)
                           arg))
                     args)))))
```

When implicitly forcing, invoking a generic function iterates over all the arguments, checks for promises and forces them. The forced values are then used to invoke the actual generic function.

Now that we have both implicit delaying and forcing, we can combine the two. Delayed evaluation can be used to provide support for infinite data structures. By using context-oriented meta-programming, it is possible to let an algorithm delay its computations when dealing with infinite data structures and compute them immediately when dealing with finite data structures.

The basic data structure in Scheme is a linked list, implemented with pairs. The two operations to retrieve the first and the second element of a pair are respectively called `car` and `cdr`. In terms of lists, `car` gives the first element and `cdr` gives a lists without its first element. Finally there is a special value denoting an empty list, called `null` and a predicate `null?` to check whether a certain value is an empty list.

The idea to support infinite lists is that an infinite list is in fact the *promise* of a list. The `cdr` of such a list is then again a promise. This way, the actual list is only computed as far as needed. Of course, somewhere during the application, the promises have to be forced. The primitives can only operate on real lists, not on promises. In order to support infinite lists, we have to redefine the primitives such that they force their argument.

```
(define mycar (ensure-generic-function car))
```

```
(define mycdr (ensure-generic-function cdr))
(define mynull? (ensure-generic-function null?))

(implicitly-force mycar)
(implicitly-force mycdr)
(implicitly-force mynull?)
```

Now we have our own versions of the primitives that will force their argument if the `implicit-forcing` layer is active. If not, these functions behave in the exact same way as the original primitives. Replacing the original primitives with these new ones is not possible because of bootstrapping issues (the implementation of TinyContext itself uses lists as well).

Now it is time to actually construct such an infinite list. The following function creates one with increasing numbers, starting from an initial number `n`.

```
(define integers-from
  (ensure-generic-function
   (lambda (n)
     (cons n (integers-from (+ n 1))))))
```

Note that there is no conditional statement to halt the recursion at some point. Under normal circumstances, invoking `integers-from` would result in an infinite loop. However, by delaying the computation of the list, `integers-from` will simply return a promise of a list.

```
(implicitly-delay integers-from)
```

The following transcript shows an example of an infinite list, created with `integers-from`. `mylist-ref` is a function that will return an element of a list at a certain position (the first element has position zero). `mylist-ref` is equivalent to the Scheme function `list-ref`, but is uses the new definitions for the primitive list operations. This is to make sure that the infinite list is forced when necessary.

```
> (with-layer implicit-delaying
    (with-layer implicit-forcing
      (let ((lst (integers-from 1)))
        (mylist-ref lst 3))))
4
```

Note that both layers are activated: `implicit-delaying` to make sure that the computation infinite list is delayed and `implicit-forcing` to ensure that the primitives force the computation of the infinite list (only as far as necessary).

A common operation on lists is `map`. `map` creates a new list by iterating over a given list and applying a given function to each element. Those results then form the new list. Here as well, we have to redefine this operation in order to use the new primitives.

```
(define mymap
  (ensure-generic-function
   (lambda (fun lst)
     (if (mynull? lst)
         null
         (cons (fun (mycar lst))
               (mymap fun (mycdr lst)))))))
```

This way of implementing `map` would normally not be able to deal with infinite lists. It would never reach the end condition and never stop iterating over the list. The solution lies again in making the computation of `map` delayed. Note, however, that this is independent of the definition of `map` itself.

```
(implicitly-delay mymap)
```

Now we can define for example a function that increments each element of a list. The function for incrementing a number will also print some text on the screen. This allows us to follow what happens.

```
(define (increment-all lst)
  (mymap (lambda (x)
           (display "inrementing ")
           (display x)
           (newline)
           (+ x 1))
         lst))
```

This function works as expected on finite lists. As long as the layers for delaying and forcing are not activated, no computation is delayed and nothing has to be forced.

```
> (increment-all (list 1 2 3 4 5))
inrementing 1
```

```
inrementing 2
inrementing 3
inrementing 4
inrementing 5
(2 3 4 5 6)
```

If we want to work with infinite lists, we have to active the layers for delaying and forcing.

```
> (with-layer implicit-delaying
    (with-layer implicit-forcing
      (let ((lst (increment-all (integers-from 1))))
        (mylist-ref lst 3))))
inrementing 1
inrementing 2
inrementing 3
inrementing 4
5
```

In this transcript, `incremente-all` is applied on an infinite list, but because the computation is delayed, this does not create an infinite loop. It is only when the number on position `3` is asked that the four first elements of the list are computed.

This example shows clearly that we can change the semantics of the language. Operations that were not defined with infinite lists in mind could nevertheless deal with them by changing the way they are applied on their arguments.

There is another way to support infinite lists: instead of delaying those functions that produce an infinite list, the computation of the arguments passed to the primitive operations that construct the list could be delayed. This is called *call-by-need* (as opposed to *call-by-value*). When using call-by-value, the arguments passed to a function are first evaluated and only then the function is applied to the resulting values. In the case of call-by-need, the arguments are computed at the moment when they are actually needed. In the case of lists, this would mean that the `cons` operation would use call-by-need, and thus not evaluating the two arguments. When the elements of the pair are asked (with `car` or `cdr`) the actual values are computed. Implementing call-by-need is not possible in TinyContext because generic function invocation relies on the function application of Scheme (the base language for TinyContext), and Scheme provides no call-by-need semantics.

6.3 Tracematches

Tracematches is an extension to AspectJ that allows the programmer to trigger the execution of extra code by specifying a regular pattern of events in a computation trace (Allan et al., 2005). The matching patterns can contain free variables so events can be matched not only by the kind of event, but also by the value associated with the free variables.

The following example shows a tracematch that ensures the safe use of iterators. It is usually the case that after modifying a datasource, an iterator on that datasource that was created before the modification should no longer be used because the iterator's internal state might be corrupted. By looking at the history of the usage of both the datasource and the iterator, the tracematch can detect situations where an iterator is used when its underlying datasource has been modified, and throw an exception.

```
tracematch (Iterator i, Datasource ds) {
  sym create_iter after returning(i):
      call(Iterator Datasource.iterator())
   && target(ds);
  sym call_next before:
      call(Object Iterator.next())
   && target(i)
  sym update_source after:
      call(* Datasource.update(..))
   && target(ds);

  create_iter call_next*
      update_source+ call_next
  {
    throw new ConcurrentModificationException();
  }
}
```

The tracematch contains 4 parts:

- A set of *free variables*. In this example, the free variables are `i` for the iterator and `ds` for the datasource.
- A set of *symbols*. They declare the events of interest. Because Tracematches is based on AspectJ, the symbols are defined with AspectJ pointcuts.
- A regular expression describing the *pattern* that has to be matched with the history of the execution.

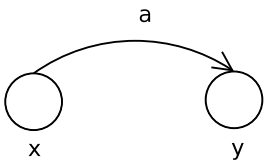

Figure 6.1: Transition from state x to state y for symbol a.

- The *code* that has to be executed when there is a match. When the code is executed, the free variables will be bound, so the code can use them.

In this example, there are three symbols of interest here: the creation of an iterator on a particular data source (`create_iter`), asking the next element from the iterator (`call_next`) and update operations on the datasource (`update_source`). When there is a creation, followed by some iteration steps, one or more update(s) and then another iteration step, an exception is thrown.

Not only the symbols play a role in the matching: the values of the free variables are also taken into account. For example, consider a datasource A and an iterator B that operates on A. If during the process of using B, some other datasource C is updated, this should not throw an exception. In other words, all occurrences of the same variable in the symbols must equal a single value in order to match the pattern. The semantics of a tracematch are then defined by all the sets of consistent bindings for which the pattern has a match in the execution trace of the program. If trying to match an event causes symbols to bind variables to a different value, they are ignored. This means that tracematches can not only capture traces in the control flow of the program, but can also capture the behavior of specific objects or groups of objects.

Allan et al. (2005) also provide a reference implementation, expression in AspectJ. Since the pattern is expressed with a regular expression, it can be converted to a finite state automaton. Each node has a *constraint*, expressed in propositional logic. The constrains of each node responds to the following rule: the execution trace can reach a certain node if and only if the constraints evaluates to true. The bindings themselves are also included in the constraint. For example, a constraint of a certain state might be as follows: `(and (= x 1) (not (= y 2)))`. This means that the execution trace can reach that node for bindings were `x` is equal to `1` and `y` is not equal to `2`..

Now we can look at how the constraints should be updated. Each event that matches a symbol causes the nodes of the finite state automaton to update their constraints according to the values of the free variables. Suppose two nodes `x` and `y` and a transition from `x` to `y` for symbol `a` (figure 6.3). If an event matches `a`, the constraints of the nodes `x` and `y` are update according to the following semantics.

- "If x was reachable before the event, y is now reachable if the free variables match the values of the event." The new constraint of y becomes conjunction of the old constraint of x and an expression that evaluates to true if *all* the variables used in the symbol a are *equal* to the run-time values used in the event.
- "If x was reachable before the event, x is still reachable as long as the free variables do not match the values of the event." The new constraint of x becomes the old constraint of x and an expression that evaluates to true if *some* of the variables used in the symbol a are *unequal* to the run-time values used in the event.

The only remaining question is when the code of the tracematch is actually executed. The finite state automaton has one or more final states, each having a constraint associated with them. Each time that they change (because of an event), they are evaluated. For each set of bindings that can be found in such a way that the constraints evaluates to true, the code is executed.

6.3.1 Trace matching with TinyContext

Now we will implement something similar in TinyContext. The goal of doing so is not to have an implementation of Tracematches in TinyContext, but to show that by using the constructs for Context-Oriented Meta-Programming to implement something like trace matching, the resulting language construct can be used to define context-dependent behavior.

Our implementation in TinyContext does not fully support all the functionality of Tracematches but enough to express the same example as the one given in the previous section. The only difference is that the pattern is not a regular expression but just a sequence of symbols.

```
(define conc-mod
  (let ((create (make <tmAfterSymbol>
                      'generic create-iterator
                      'args (list 'datasource)
                      'result 'iterator))
        (next (make <tmBeforeSymbol>
                    'generic iterator-next
                    'args (list 'iterator)))
        (modify (make <tmAfterSymbol>
                      'generic modify-source
                      'args (list 'datasource))))
```

```
(make-tracematch (list 'datasource 'iterator)    ; free variables
                 (list create next modify)       ; symbols
                 (list create next modify next)  ; pattern
                 (lambda (datasource iterator)   ; code
                   (error "Concurrent modification exception on"
                          datasource))))
```

First the symbols are defined. The `create` symbol should intercept invocations of the generic function `create-iterator` *after* the iterator is created. The argument passed to `create-iterator` - the datasource on which an iterator has to be created - must be bound to the free variable `datasource` and the resulting iterator to the free variable `iterator`. The `next` symbol should intercept invocations of `iterator-next` *before* execution and bind the passed argument to `datasource`. Finally, the third symbol `modify` should intercept invocations of `modify-source` and bind the passed argument to `datasource`.

With the symbols defined, the actual tracematch can be expressed. As in the original Tracematches, a tracematch consists of four parts: a number of free variables (in this case `datasource` and `iterator`), a number of symbols (the objects defined earlier), the pattern that has to be matched and the code that has to be executed in case of a match. In this implementation, the pattern can only be a sequence of symbols, so in the example given, the pattern is "the creation of an iterator, followed one iteration step, then a modification of the datasource and finally another iteration step". If such a pattern is about to take place in the execution of a program, an error should be raised before the final iteration step. This is expressed in the code. The values of the free variables are also passed the code.

Now we are ready to see how context-oriented meta-programming complements a language feature like tracematches. Defining a tracematch implicitly defines a new layer, that is returned by `make-tracematch`. In the example above, `conc-mod` contains the layer defined by the tracematch. If it is activated, the relevant events will be matched, if it is not activated, the tracematch is ignored.

Consider the following sample application.

```
(define (application)
  (let* ((source (list 1 2 3))
         (other-source (list 4 5 6))
         (it (create-iterator source))
         (other-it (create-iterator other-source)))
    (iterator-next it)
    (modify-source source)
```

```
    (iterator-next other-it)
    (iterator-next it)))
```

The application creates two datasources (two lists) and creates iterators on both of them. There is one iteration step on the first iterator and the first datasource is modified. Then there is an iteration step on the second iterator. This should have no influence on the first iterator and datasource, so this should certainly not raise an error. Finally, there is one iteration step on the first iterator. This is the moment where the tracematch should raise an error, because the underlying datasource has been modified.

The functions `create-iterator`, `iterator-next`, etc. have been implemented in such a way that they print some output on the screen so that we can see what happens. Consider the following transcript.

```
> (application)
Creating iterator on (1 2 3)
Creating iterator on (4 5 6)
Getting next element of (1 2 3)
Modifying (1 2 3)
Getting next element of (4 5 6)
Getting next element of (1 2 3)
```

Currently, the tracematch is not taken into account, so the application runs as normal. But when we activate the layer `conc-mod`, the last iteration results in an error.

```
> (with-layer conc-mod
              (application))
Creating iterator on (1 2 3)
Creating iterator on (4 5 6)
Getting next element of (1 2 3)
Modifying (1 2 3)
Getting next element of (4 5 6)
Error: Concurrent modification exception on (1 2 3)
```

6.4 Summary

This chapter gave three examples of context-oriented meta-programming. The first one was to demonstrate how it to use the constructs from the previous chapter. It provides also a good opportunity to show the effect of layer precedence when combining two layers that have definitions for the same operations. The second one showed how

algorithms that operate on finite lists could be used to operate on infinite lists when necessary by changing the semantics of function application. The third one showed that by using context-oriented meta-programming to implement a language feature for matching patterns in the trace of a program, the tracematch could be activated with a layer.

Chapter 7

Comparison with Aspect-Oriented Programming

In this chapter we compare context-oriented meta-programming with aspect-oriented programming because the two approaches have an overlapping range of applications.

We start by explaining aspect-oriented programming and the key concepts.

7.1 Aspect-Oriented Programming

Aspect-Oriented Programming (Masuhara and Kiczales, 2003) is a recent programming paradigm that provides language constructs for better modularization of crosscutting concerns. Two concerns crosscut each other if they cannot be modularized with standard language constructs such as classes, methods, modules and so on. Well-known examples of crosscutting concerns are synchronization, transaction management, exception handling, etc. Because they cannot be modularized, they are spread out over in the entire source code of the software system. The phenomenon of concerns being contained in several modules is called *scattering*. These modules then deal with several concerns instead of just one. This is called *tangled code*. The goal of aspect-oriented programming is to remove the tangled code of crosscutting concerns from the base program and to modularize the crosscutting concerns in an *aspect*, a new unit of modularity.

Aspects are then woven into the base program by an aspect weaver. An aspect consists of two parts: the implementation of the crosscutting concern itself, and a description of where the concern actually crosscuts the other concerns of the system. The aspect

weaver is then responsible to add the functionality of the crosscutting concern to the other modules of the system.

The three main concepts of aspect-oriented programming are

Joinpoint Joinpoints are reified events in the execution of a program. The aspect weaver has a set of different types of joinpoints such as sending a message or creating an instance of a class. It is at these joinpoints that an aspect can add or modify the functionality.

Pointcut A pointcut is a description of the set of joinpoints where the functionality should be added or modified. A pointcut is written in a pointcut language. Since the domain of the pointcut is the joinpoint model, the pointcut language can be regarded as a meta-language.

Advice The advice contains the actual implementation of the extra behavior. It also specifies how it should be applied on a joinpoint: before, after or around the base-program's behavior. Before- and after-advices only add functionality to the base program, around-advices can change the functionality of the base program

7.2 AOP and Context-Oriented Programming

Context-Oriented Programming, presented in Chapter 2, is a new programming paradigm that provides techniques for modularizing behavior that might depend on the context in which it is being executed. The motivating example was to have different views on a group of objects without using conditional statements everywhere in the code and without separating the code for displaying the objects into different classes.

As explained in section 2.2, the three main concepts of context-oriented programming are grouping of context-dependent behavior in layers, dynamic activation and thread-locality.

It is clear that these concepts are different from the ones that form the basis for AOP. Indeed, AOP and context-oriented programming serve fundamentally different purposes and apply different modularization approaches.

Aspect-oriented programming focuses on the separation of crosscutting concerns from the rest of the system, while context-oriented programming is intended for the modularization of context-dependent behavior. The most important difference is that layers do not modularize a crosscutting concern. In the motivating example, there is already code for displaying objects in the base program, and different layers define different implementations for the same operations. In other words, they all deal with the same

concern, but provide different implementations for them. This is contrary to aspects, that capture one specific crosscutting concern.

Another important difference is that a layer does not use a pointcut or joinpoints to define the adaptations to the behavior of the base program. It just uses layered classes and methods, the same sort of definitions of the base-program, with the only difference of mentioning a layer in which the definitions are active.

There are however technical similarities between aspect-oriented programming and context-oriented programming. Some aspect-oriented approaches provide a `cflow` construct in their pointcut language. This allows restrict the pointcut to those joinpoints that are under the control flow of another joinpoint. This is similar to dynamic layer activation, where the definitions of the layer only become part of the running program in the control flow of the layer activation. The difference is that the `cflow` construct is used in the pointcut (making it actually a meta-level construct) while layer activation is used in the base-program (layer activation is base-level construct). This also means that the base program is not oblivious of the fact that the behavior of the program is modified at a certain point in the control flow of the program.

7.3 AOP and Context-Oriented Meta-Programming

For context-oriented meta-programming, the two approaches seem to be even more related. The reason for this is that meta-programming and aspect-oriented programming can both be used to modularize non-functional concerns. Non-functional concerns do not deal with application logic itself, but with *how* the application logic should be executed. Typical examples include logging, synchronization and persistency. Meta-programming with a metaobject protocol is a good approach for implementing these concerns. The example of synchronization is already demonstrated in Section 3.2.2. However, non-functional concerns are often crosscutting as well. This makes aspect-oriented programming also a good approach to express these concerns.

Despite this overlapping applications, the two approaches are still fundamentally different. This is visible in the example described in Section 6.2. This example is not some non-functional concern, but a different semantics for function application, something that belongs entirely at the meta-level.

7.4 Summary

In this chapter we gave a short comparison with aspect-oriented programming. We noticed that even tough there are a number of technical similarities and they have a

common range of applications, the approaches are fundamentally different. Aspect-oriented programming focuses on the separation of crosscutting concerns from the base program while both context-oriented programming an context-oriented meta-programming focus on the modularization of context-dependent behavior.

Chapter 8

Conclusions and Future Work

In this dissertation we presented how context-oriented programming at the meta-level can be used to express language semantics that depend on the context in which a program is executed and illustrated the usefulness of this approach based on our own implementation, called TinyContext.

We started out in Chapter 2 by explaining Context-Oriented Programming. We demonstrated the motivation behind context-oriented programming by working out an example where context information played a role in the desired behavior of the program. We observed and identified the problems that arise when using current mainstream programming languages for achieving a good modularization of the context-dependent behavior. We noticed that the two techniques that can be used, namely checking the context information with conditional statements and separating the context-dependent behavior into different classes, are not sufficient. We showed how context-oriented programming addresses these problems by providing a new unit of modularization that can group context-dependent behavior and activate it in a dynamic and thread-local manner.

In Chapter 3 we presented meta-programming and the notion of a metaobject protocol in particular. A metaobject protocol is a principled way of offering meta-programming facilities in a language in such a way that it allows the programmer to incrementally modify or extend the language to his own needs. We noticed that defining context-dependent meta-behavior would require the use of context-oriented programming at the meta-level because making run-time modifications to the association and composition of metaobjects would not respect the thread-locality, and because defining the context-dependent meta-behavior in the metaclasses would introduce the same problems as those encountered in Chapter 2.

In Chapter 4 we presented TinyContext, our experimental environment that allows us

to use the constructs for context-oriented programming not only for base-level objects, but for metaobjects as well. The constructs for context-oriented programming in TinyContext are based on those found in ContextL. The difference is that in TinyContext, the support is not an extension of the language, but a modification. This means that support for context-oriented programming is pushed into the language and by consequence, it can be applied on the metaobject protocol of TinyContext as well.

In Chapter 5 we presented Context-Oriented Meta-Programming. It applies context-oriented programming at the operations of the meta-level. The focus was on the operations related to slot access and generic function invocation. The semantics for slot access and generic function invocation are determined when the class or the generic function is defined but having true context adaptations requires run-time support as well. Therefore, context-oriented programming is applied in two different ways. First there is the context of definition. How a class or a generic function is defined does not only depend on the metaclass that is chosen, but also on the context in which the class or generic function is defined. This allows us to add the run-time hooks that are needed. Secondly, once available, the hooks can be used to add context-dependent meta-behavior and to activate it with layer activation.

In Chapter 6 we gave three examples of context-oriented meta-programming. They show that context-oriented meta-programming is indeed useful for defining context-dependent changes to the semantics of function application, and also to define language constructs that automatically incorporate support for context adaptations.

In Chapter 7 we gave a short comparison with aspect-oriented programming. We noticed that even tough there are a number of technical similarities and they have a common range of applications, the approaches are fundamentally different. Aspect-oriented programming focuses on the separation of crosscutting concerns from the base program while both context-oriented programming an context-oriented meta-programming focus on the modularization of context-dependent behavior.

In general, we can conclude that context-oriented meta-programming is indeed a successfull approach for expressing context adaptation at the meta-level. To support this, we created TinyContext, an experimental implementation where context-oriented prorgamming and meta-programming are combined. To show that context-oriented meta-programming is useful to define context-dependencies at the meta-level, we developed three different examples. However, this work was only a first attempt to realize explicit modularization of context-dependent meta-level behavior. Even context-oriented programming is a quite recent research topic that still needs a lot of research, for creating the technology, as well as for gaining more insight in the problem domain of context dependencies.

8.1 Future work

Based on our investigations, we identified a number of potential areas for future work.

First of all, there is still lots of room for improvements to TinyContext. In the implementation of TinyContext, little attention was paid on effeciency issues. Furthermore, in its current implementation, TinyContext does not include all the features that are available in ContextL. Layered methods and generic functions are supported, but not layered classes. Our experiments with context-oriented meta-programming do not require layered classes. It is a question for future research whether layered classes are useful as well to define context-dependent meta-level adaptations.

Another possible extension of TinyContext is the addition of support for layer specific state. The tracematches example in Chapter 6 defined a language construct that, when used, creates a layer. In the implementation of the tracematches, maintaining the internal state of the tracematch uses the lexical scoping of Scheme. It could be more appropriate to associate this state with the layer itself. In general, layer specific state could allow the different definitions in a layer to communicate with each other, even if they belong to different classes. Layer specific state might also be useful as communication channels between multiple concurrent activations of the same layers.

Another important topic for future research is to aim explore the applicability of context-oriented meta-programming in various areas that require context adaptations. In fact, this is also the case for context-oriented programming. Possible candidates include personalization and ambient intelligence.

Finally, there is a particular analogy between the motivating example of context-oriented programming and reflective architectures based on *mirrors* (Bracha and Ungar, 2004). In the motivating example, the code for displaying was separated from the objects to be displayed in order to support different views. In a similar fashion, mirrors separate the meta-level reflective capabilities from the base-level objects. This allows to have different kinds of mirrors on the same objects. However, Bracha and Ungar recognize that, depending on the problem to be solved, this separation may become problematic instead of helpful. Context-oriented programming allows the views to remain associated with the objects, but also allows to define different views in different layers and to active them depending on the context of use. This suggests that context-oriented meta-programming can be used to support different ways of reflecting on objects without separating this functionality into mirrors.

Appendix A

Lexical vs. Dynamic Scoping

In this appendix we explain and compare lexical and dynamic scoping, because layer activation in context-oriented programming uses this concept.

Costanza (2003) says the following about lexical and dynamic scoping:

> "A definition is said to be dynamically scoped if at any point in time during the execution of a program, its binding is looked up in the current call stack as opposed to the lexically apparent binding as seen in the source code of that program. The latter case is referred to as lexical scoping."

The next two sections further explore lexical and dynamic scoping and section A.3 shows how dynamic scoping can be achieved in Scheme.

A.1 Lexical Scoping

The following program demonstrates lexical scoping in Scheme (Adams et al., 1998):

```
(define x 1)

(define (f) x)

(define (g)
  (let ((x 2))
    (f)))
```

In this program, the variable `x` is bound at two places. It has a global binding where `x` has value `1` and a local binding where `x` has value `2`. There is only one reference to `x` (in `f`) and its lexically apparent binding is the global one. Because definitions in Scheme are always lexically scoped, the function `f` will always return the value of the global binding of `x`, in this case 1. The binding of `x` in `g` does not affect the behavior of `f`.

Almost all programming languages in wide use offer only lexical scoping and not dynamic scoping. For example, the following program in C is equivalent to the Scheme program above:

```
int x = 1;

int f() {
  return x;
}

int g() {
  int x = 2;
  return f();
}
```

A.2 Dynamic Scoping

The definition in section **??** states that dynamically scoped variables are looked up in the current call stack instead of their lexical environment. The decision to use lexical or dynamic scoping can be made at two places:

- in the definition of the variable itself. Every reference to that variable is looked up in the current call stack. We will refer to this case as *dynamically scoped variable lookup*.
- in the function (or a similar construct) that contains the reference to the variable. Each free variable is either looked up in the (static) environment of definition (this results in lexical scoping) or in the (dynamic) environment of execution (this results in dynamic scoping). We will refer to this case as *dynamically scoped function application*.

These two cases are separately explored in the following two sections. There is however a special case: dynamically scoped variables containing functions. This is discussed in section A.2.3.

A.2.1 Dynamically Scoped Variable Lookup

Common Lisp [1] offers dynamically scoped variables (referred to as *special* variables). Consider the following program in Common Lisp, similar to the example in Scheme, but now with a dynamically scoped variable `*x*`:

```
(defvar *x* 1)

(defun f() *x*)

(defun g()
  (let ((*x*  2))
    (f)))
```

As in the Scheme version, `*x*` has a global binding and a local binding in `g`. But even tough the lexically apparent binding for the reference to `*x*` in `f` is still the global one, `f` will use the local binding of `g` if (and only if) `f` is called from `g`. This is because in Common Lisp, a global variable introduced by `defvar` is always dynamically scoped. This means that the new binding of `*x*` in `g` affects the behavior of `f`.

To avoid confusion between lexically and dynamically scoped variables, the latter are given names that have leading and trailing asterisks. This idiom is generally accepted by Common Lisp programmers. This way, it is not possible to accidentally rebind a special variable with a local variable.

A.2.2 Dynamically Scoped Function Application

D'Hondt and De Meuter (2003) introduce a prototype-based object model, called *Pic%* [2], that unifies the environment model and the object model. Dynamically scoped function application is used to support reentrancy and late binding for methods: functions are not executed in the static environment of the callee, but in an environment specified by the caller.

Environments in Pic% are first-class values and they are used to represent objects. New objects are created by extending the current environment with new bindings and then capturing the extended environment with the primitive function `capture()`. Consider the following example, expressed in AmbientTalk (which is based on Pic%)(Dedecker et al., 2005).

[1] More information on Common Lisp can be found in (Steele, Jr., 1990).

[2] Pico is a minimal functional language and Pic% is extension to support object-oriented programming (% = o/o = object-oriented).

```
makeObject()::
  { n: 1;
    operator(a, b) :: a + b + n;
    capture() };

o: makeObject();
o.operator(3, 4);      ` => 8 `
```

In this program, `o` will contain an object with an attribute `n` initialized to 1 and one method `operator(a, b)` . Evaluating the expression `o.operator(3, 4)` is equivalent to looking up the function `operator` in the environment `o` and executing it (with parameters `3` and `4`) in the same environment `o`.

Inheritance and overriding can be achieved with mixin methods. Basically, they do the same as object creation: they extend the environment and return it as a new object.

```
f() :: {
  n: 10;
  capture() };

g() :: {
  operator(a, b) :: 2 * super().operator(a, b);
  capture() };
```

By applying these mixin methods to existing objects, they can override methods. In fact, because methods and variables are treated in the same way, variables can also be overridden. The following code fragment demonstrates the effect of applying the mixin methods on `o`. Various combinations are possible.

```
oF: o.f();
oF.operator(3, 4);     ` => 17 `
oG: o.g();
oG.operator(3, 4);     ` => 16 `
oFG: oF.g();
oFG.operator(3, 4);    ` => 34 `
```

In this program, `oF` is the result of applying the mixin method `f` on the object `o`. Invoking `operator` on `oF` will use the new binding of `n`. In the same way, `oG` will use the new binding for `operator` and `oFG` will use both new bindings.

A.2.3 Dynamically Scoped Functions

In languages like Common Lisp and Scheme, functions are first-class entities: they can be passed around as arguments or returned as the result of a computation. This can lead to confusion when talking about *dynamically scoped functions*. The distinction should be made between dynamically scoped function application and dynamically scoped variables that happen to contain a function.

The following example in Common Lisp demonstrates a dynamically scoped variable containing a function:

```
(defvar *operator*
  (let ((n 1))
    (lambda (a b) (+ a b n))))

(defun f(a b)
  (let ((n 10))
    (funcall *operator* a b)))

(defun g(a b)
  (let ((*operator* (lambda (a b) (* a b))))
    (f a b)))

(f 3 4)      ; => 8
(g 3 4)      ; => 12
```

The variable `*operator*` is a dynamically scoped *special* variable, but now it contains a function instead of a number. If `f` is called from `g`, the first binding found for `*operator*` in the current call stack is the one defined in `g`; otherwise, when `f` is directly called, it is the global one.

The value of `*operator*` will always be executed in the environment of definition, even tough `*operator*` is a dynamically scoped variable. To demonstrate this, a free variable `n` has been added to the global binding of `*operator*`. When `*operator*` is invoked in `f`, the first binding found for `n` in the current call stack is `10`. However, in the environment of definition of the global binding of `*operator*`, the binding for `n` is `1`. The latter is used, so the result of `(f 3 4)` is `8`.

In order to ease the definition of this kind of dynamically scoped function in Common Lisp, Costanza (2003) introduces a number of language constructs. The following code fragment demonstrates the definition of a dynamically scoped global function with `defdynfun`.

```
(defdynfun operator(a b) (+ a b))
```

Because the naming convention for special variables (based on asterisks) is less suited for functions, `defdynfun` will store the body in a special variable and define a function that just forwards any call to that special variable. So the previous code fragment is translated into the following definitions:

```
(defvar *operator* (lambda (a b) (+ a b)))

(defun operator (&rest args) (apply *operator* args))
```

Rebinding a global function with dynamic extent can be accomplished with `dflet`. Additionally, there is a way to refer to the previous binding of a function by way of an implicit local function `call-next-function`. This is demonstrated in the next code fragment.

```
(defun f(a b) (operator a b))
(defun g(a b)
  (dflet ((operator(a b) (* (call-next-function a b) 2)))
    (f a b)))
```

If `f` is called directly, it will use the global binding of `operator`: `a` and `b` will be added. If `f` is called from `g`, the local binding of `operator` in `g` will be used: `call-next-function` will add `a` and `b`, then the result is multiplied by `2`.

It is also possible to turn a function that was already defined with `defun` into a function that supports dynamic scoping with `redefdynfun`. So the following definitions are effectively equivalent:

- `(defdynfun operator(a b) (+ a b))`
- `(defun operator(a b) (+ a b))`
  `(redefdynfun operator)`

A.3 Dynamically Scoped Variables in Scheme

The Scheme standard does not provide any constructs for dynamic scoping. Steele, Jr. and Sussman (1976) present an implementation of dynamically scoped variables on top of lexical scoping and some Scheme dialects provide some form dynamic scoping. For example, MzScheme provides two forms: *fluid* variables and *parameter* objects (Flatt, 2005). In the following example, the variable `x` behaves as a dynamically scoped variable.

```
(define x 1)

(define (f) x)

(define (g)
  (fluid-let ((x 2))
    (f)))
```

A major drawback of `fluid-let` in MzScheme is that it is explicitly defined to assign the new values to the variables, evaluate the body and then restore the old values. This gives problems in the case of multi-threading: rebinding a variable with dynamic scope should not alter the binding for that variable in other threads.

Parameter objects in MzScheme do not have this problem. They can be given a new binding with dynamic scope in a thread-safe manner by way of the `parametrize` form. The following Scheme program is equivalent to the previous example, but now it is thread-safe:

```
(define x (make-parameter 1))

(define (f) (x))

(define (g)
  (parameterize ((x 2))
    (f)))
```

Note that in order to retrieve the value of a parameter, the parameter object has to be invoked as a function. Because of this extra indirection, the correct semantics can be implemented. In fact, MzScheme defines several standard parameter objects, for example for the current directory and the current output port.

Appendix B

TinyCLOS

In this appendix we present TinyCLOS, its metaobject protocol and its implementation in more detail. All the information for the following paragraphs was found in the source code and in an announcement [1].

TinyCLOS was developed at Xerox Parc by Kiczales (1992). It is a very simple CLOS-like language, embedded in Scheme, with a simple metaobject protocol. The primary goal was to let people play with the metaobject protocol without the relative complexity of working in Common Lisp. Because of the pedagogical purpose, the language and the Metaobject Protocol are very similar to Closette, the language created in "The Art of the Metaobject Protocol" (Kiczales et al., 1991). According to Kiczales, the metaobject protocol of TinyCLOS retains much of the power of both of those found in AMOP and even tough the implementation of TinyCLOS is not optimized, it could be done by using the techniques mentioned in AMOP. In fact, the slot access protocol used in this metaobject protocol is such that it should be possible to get better performance than is possible with the CLOS metaobject protocol.

B.1 The Base Language

The features of the default base language are:

- Classes, with instance slots, but no slot options
- Multiple inheritance
- Generic functions with multi-methods and class-specializers only

[1] From: `ftp://ftp.parc.xerox.com/pub/mops/tiny/tiny-announce.text`

- Primary methods and call-next-method (no other method combinations, like before or after methods)
- All metaobjects are first-class citizens and are addressed by using Scheme's lexical scoping

The entry points to the default base language are:

- Calling a generic function
- Defining a class by giving a list of superclasses and a list of slot names
  `(make-class list-of-superclasses list-of-slot-names)`
- Creating a new generic function; no parameters are needed
  `(make-generic)`
- Defining a method by giving a list of class specializers and a Scheme procedure (the actual method body)
  `(make-method list-of-specializers procedure)`
- Adding a method to a generic function; typically, defining some behavior involves creating a generic function, defining a number of methods and adding them all to the generic function. When invoked, the generic function selects the appropriate method by comparing the classes of the arguments and the specializers of all the methods that were added to the generic function.
  `(add-method generic method)`
- Creating an instance of a class
  `(make class . initargs)`
- Initializing an object. This generic function is provides by the language and should not be called directly. It is called by `make` when creating an instance and the argument `initargs` is passed to `initialize`. The user can add methods to this generic function in order to change how instances of his own classes are initialized.
  `(initialize instance initargs)`
- Accessing a slot of an object
  `(slot-ref object slot-name)`
- Assigning a new value to a slot of an object
  `(slot-set! object slot-name new-value)`

The following is a simple example of a TinyCLOS program.

```
(define <point>
  (make-class (list <object>)     ;; Superclass
              (list 'x 'y)))      ;; Slots

(add-method initialize
  (make-method (list <point>)
    (lambda (call-next-method point initargs)
      (call-next-method)
      (initialize-slots point initargs))))

(define p (make <point> 'x 1 'y 2))

(slot-set! p 'y 5)
```

The class `<point>` is defined as subclass of `<object>` (this class is defined by TinyCLOS). A point has two slots, an x and a y coordinate. Initializing a point should assign the values for both coordinates as given in the initialization arguments. Therefore, `initialize` is overridden for the class `<point>`. `initialize-slots` is a utility function that, when given an object and a list of the form `(slot-name value ...)`, will assign the given slots of the object with the associated value. Then a point is created with initial coordinated 1 and 2. The last line assigns the y coordinate to 5.

B.2 The Metaobject Protocol

TinyCLOS classes, generic functions and methods are objects themselves, called *meta-objects*. Their classes are respectively `<class>`, `<generic>` and `<method>`. Defining a class, generic function or method creates an instance of one of these *metaclasses*. The metaobject protocol defines several generic function that are invoked during the definition of the meta-objects. They are responsible for defining the semantics of the language.

First we start with an example of a meta-program in TinyCLOS.

Each generic function has an *apply function*. It is a procedure that is generated by the metaobject protocol and it is responsible for method dispatch. The generic function of the metaobject protocol that generates it is called `compute-apply-generic`. Each time a method is added to a generic function, the apply function of the generic function is computed again. Optimized implementations of a metaobject protocol can then perform partial evaluation by looking at the methods that are available.

The user can change the semantics of the language by defining new metaclasses and defining methods on them. Consider the following example in TinyCLOS, a new kind of generic function that logs all its invocations.

```
(define <logging-generic>
  (make <entity-class>
        'direct-supers (list <generic>)
        'direct-slots  (list 'name)))

(add-method compute-apply-generic
  (make-method (list <logging-generic>)
               (lambda (cnm generic)
                 (let ((original (cnm)))
                   (lambda args
                     (log-invocation (slot-ref generic 'name)
                                     (lambda ()
                                       (apply original args))
                                     args))))))
```

This meta-program defines a metaclass `<logging-generic>` and overrides `compute-apply-generic` for this metaclass. Every generic created with this metaclass will print messages before and after the actual computation[2]. This is demonstrated in the following transcript:

```
> (define double (make <logging-generic> 'name 'double))
> (add-method double
            (make-method (list <number>)
                         (lambda (cnm n)
                           (* n 2))))
> (double 3)
(double 3)
(double 3) => 6
6
```

The metaobject protocol of TinyCLOS is (just like the other MOPs of AMOP) divided up in an introspective and an intercessory part. The introspective part allows to look at the inside of the metaobjects (classes, generic functions and methods). Here is an overview:

[2]The function log-invocation takes three arguments: the name of the function that is being logged, a procedure that will perform the actual computation and the list of arguments of the invocation that is being logged.

- `class-direct-supers`: get the list of direct superclasses of a given class. This is the same list as the one that was given in the class definition.
- `class-direct-slots`: get the list of direct slots of a given class. This is also the same list as the one given in the definition.
- `class-cpl`: get the *class precedence list* of a given class. The class precedence list is an ordered list of all superclasses, direct or indirect. During the initialization of a class, the list is computed by linearizing the inheritance tree. In the special case of single inheritance, the inheritance tree is already linear. In the general case of multiple inheritance, the tree is linearized with a breath-first algorithm. One of the things that can be changed with the metaobject protocol is this linearization algorithm.
- `slots`: get the list of all the slots, both the direct slots and the inherited slots. This lists describes the actual structure of the instances of the given class.
- `generic-methods`: get the list of all the methods that are added to the given generic function.
- `method-specializers`: get the list of class specializers of the given method.
- `method-procedure`: get the method body of the given method.

The intercessory protocol provides a number of generic functions that are invoked during the initialization of the metaobjects.

Instance allocation protocol:

- `allocate-instance`: allocate an instance of a certain class.

Class initialization protocol:

- `compute-cpl`: compute the class precedence list
- `compute-slots`: compute the list of all the slots, direct or indirect (inherited)
- `compute-getter-and-setter`: compute a getter and a setter (two procedures) for a certain slot.

Generic invocation protocol (all these generic functions return a procedure that will be used during method dispatch):

- `compute-apply-generic`: compute the *apply function* of the generic function. This is the procedure that is invoked when the generic function is called and it is responsible for method dispatch.

- `compute-methods`: compute a procedure that, given a list of actual arguments, finds all the methods of the generic function that are applicable. Moreover, they should be sorted in such a way that the most specific method is first in the list and the least specific is at the end.
- `compute-method-more-specific?`: compute a procedure that, given two methods and a list of actual arguments, determines which method more specific.
- `compute-apply-methods`: compute a procedure that will apply the appropriate method(s) to the actual arguments. This procedure is responsible for the `call-next-method` machinery.

B.3 The Implementation

In this section we will take a look at the internal class and metaclass hierarchy of TinyCLOS. For the base-level programmer, to root class of every object is `<object>` but internally, `<object>` is a subclass of `<top>`. The following code fragment shows their definitions.

```
(define <top>           (make <class>
                              'direct-supers (list)
                              'direct-slots  (list)))

(define <object>        (make <class>
                              'direct-supers (list <top>)
                              'direct-slots  (list)))
```

Note that both classes are instances of `<class>`, even tough this class is not defined yet. This is a first example of a circularity that requires bootstrapping because, as we can see in the following definition, `<class>` is a subclass of `<object>`.

```
(define <class>
    (make <class>
          'direct-supers (list <object>)
          'direct-slots
             (list 'direct-supers
                   'direct-slots
                   'cpl
                   'slots
                   'nfields
```

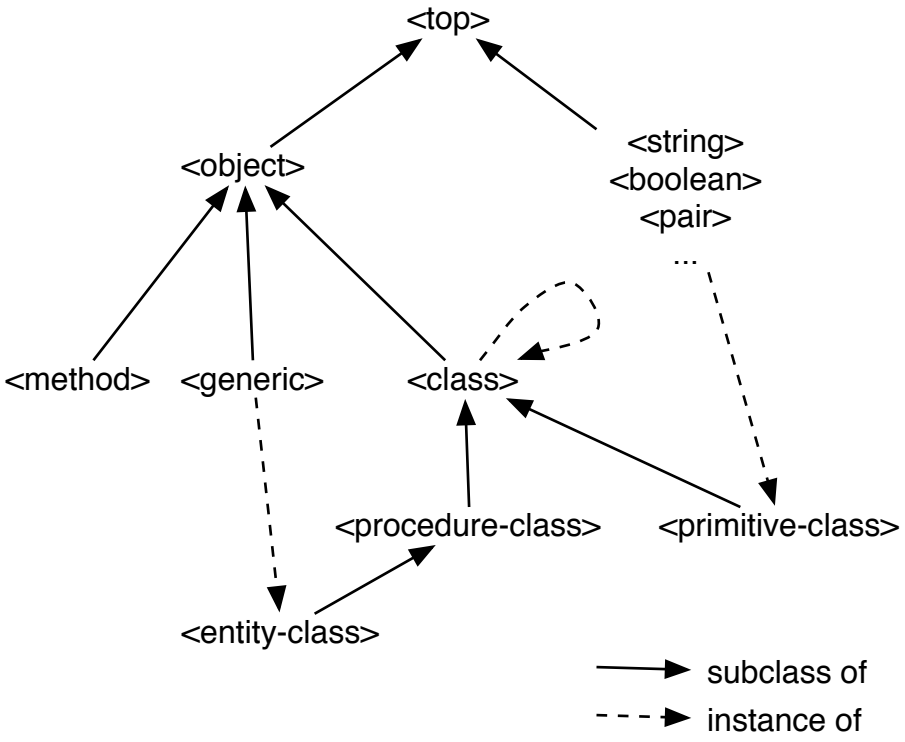

Figure B.1: Class hierarchy of the TinyCLOS implementation. If there is no "instance of" arrow, the class is an instance of <class>

```
'field-initializers
'getters-n-setters)))
```

The slot `getters-n-setters` contain a pair of procedures for each slot. The first of a pair is the setter, the second is the getter.

Here we can see another circularity: `<class>` is an instance of itself. This will cause infinite regression when accessing slots, so this will also need special attention during bootstrapping.

The rest of the meta-level classes (`<generic>`, `<method>`, `<entity-class>`, ...) are all defined in the same way. Their definitions look the same as base-level classes. We only call them "metaclasses" because of what they mean to us, but technically they are not different than ordinary classes. An overview is given in figure B.1.

Bibliography

Adams, N. I. I., Bartley, D. H., Brooks, G., Dybvig, R. K., Friedman, D. P., Halstead, R., Hanson, C., Haynes, C. T., Kohlbecker, E., Oxley, D., Pitman, K. M., Rozas, G. J., G. L. Steele, J., Sussman, G. J., Wand, M., and Abelson, H. (1998). Revised5 report on the algorithmic language scheme. *SIGPLAN Not.*, 33(9):26–76.

Allan, C., Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhotak, O., de Moor, O., Sereni, D., Sittampalam, G., and Tibble, J. (2005). Adding trace matching with free variables to aspectj. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 345–364, New York, NY, USA. ACM Press.

Bock, D. (2000). The paperboy, the wallet, and the law of demeter. http://javaguy.org/papers/demeter.pdf.

Bracha, G. and Ungar, D. (2004). Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 331–344, New York, NY, USA. ACM Press.

Chiba, S. (1995). A metaobject protocol for C++. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, SIGPLAN Notices 30(10), pages 285–299, Austin, Texas, USA.

Chiba, S. (2000). Load-time structural reflection in Java. *Lecture Notes in Computer Science*, 1850:313–??

Chiba, S. and Masuda, T. (1993). Designing an extensible distributed language with a meta-level architecture. *Lecture Notes in Computer Science*, 707:482–??

Cointe, P. (1987a). Metaclasses are first class: The objvlisp model. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 156–162, New York, NY, USA. ACM Press.

Cointe, P. (1987b). Metaclasses are first class: the ObjVLisp model. In *Proceedings of the OOPSLA '87 Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 156–162. ACM Press.

Costanza, P. (2003). Dynamically scoped functions as the essence of aop. *SIGPLAN Not.*, 38(8):29–36.

Costanza, P. and Hirschfeld, R. (2005). Language constructs for context-oriented programming - an overview of contextl. Dynamic Languages Symposium, co-located with OOPSLA'05, October 18, 2005, San Diego, California, USA.

Dedecker, J., Cutsem, T. V., Mostinckx, S., D'Hondt, T., and Meuter, W. D. (2005). Ambient-oriented programming. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 31–40, New York, NY, USA. ACM Press.

D'Hondt, T. and De Meuter, W. (2003). Of first-class methods and dynamic scope. In *Proceedings of LMO*, pages 137–149.

Flatt, M. (2005). PLT MzScheme: Language manual. Technical Report PLT-TR05-1-v300, PLT Scheme Inc. `http://www.plt-scheme.org/techreports/`.

Gybels, K. (2001). Aspect-oriented programming using a logic meta programming language to express cross-cutting through a dynamic joinpoint structure. licentiate's thesis, vrije universiteit brussel.

Kiczales, G. (1992). TinyCLOS. ftp://ftp.parc.xerox.com/pub/mops/tiny/.

Kiczales, G., Rivières, J. d., and Bobrow, D. G. (1991). *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA.

Lamping, J., Kiczales, G., Rodriguez, L., and Ruf, E. (1992). An architecture for an open compiler.

Maes, P. (1987a). Computational reflection. phd. thesis. laboratory for artificial intelligence, vrije universiteit brussel. brussels, belgium. january 1987.

Maes, P. (1987b). Concepts and experiments in computational reflection. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 147–155, New York, NY, USA. ACM Press.

Masuhara, H. and Kiczales, G. (2003). Modeling crosscutting in aspect-oriented mechanisms. In *European Conference on Object-Oriented Programming (ECOOP 2003)*, pages 2–28.

Paepcke, A. (1993). User-level language crafting: introducing the clos metaobject protocol. pages 65–99.

Rodriguez, L. H. J. (1991). Coarse-grained parallelism using metaobject protocols.

Sharp, A. (1997). *Smalltalk By Example*. McGraw-Hill.

Steele, Jr., G. L. (1990). *Common LISP: the language (2nd ed.)*. Digital Press, Newton, MA, USA.

Steele, Jr., G. L. and Sussman, G. J. (1976). Lambda: The ultimate imperative. Technical Report AI Lab Memo AIM-353, MIT AI Lab.

Tatsubori, M., Chiba, S., Itano, K., and Killijian, M.-O. (1999). OpenJava: A class-based macro system for java. In *OORaSE*, pages 117–133.