Formal Models for Aspect-Oriented Software Development

Mark Charles Skipper

Submitted in candidature of the degree of
Doctor of Philosophy
Imperial College London

2004

Abstract

ASPECT ORIENTED PROGRAMMING (AOP) is a new technology for separation of concerns. AOP techniques make it possible to modularise cross-cutting aspects of a system: those that would otherwise be scattered throughout the code because the concerns they address are not aligned with the conceptual decomposition of the system. Without AOP techniques, such aspects are hard to encapsulate. The essential components of AOP are exemplified by its flagship language ASPECTJ– an aspect oriented extension to JAVA– these are: *member introduction*: a way for one class to add members to other classes, *parent introduction*: a way for classes to influence the inheritance of other classes, *join points*: places where behaviour can be added to existing code, *advice*: a way to modify behaviour at join points; *aspects*: encapsulated units combining join point and behaviour specifications together with their associated state; and *weaving* a method of incorporating these units into programs.

This thesis is a study of the semantics of the novel features of ASPECTJ. It presents a formal model of a simple JAVA-like language with a type system and operational semantics, then goes on to extend it with the most paradigmatic features of ASPECTJ: formal definitions of: join points, advice and aspects are presented for the first time, in the context of an operational semantics. A formal definition of weaving is developed. The impact of these features on the static type system is investigated. Examples are used to demonstrate their expressiveness. The increments to the semantics of the base language are relatively small for each feature making the presentation simple to understand. The work is suitable as a base for further investigation of aspect oriented language features using formal techniques.

Contents

In loving memory of Joyce and Ernie.

Acknowledgements

Thanks to: Sophia (adviser, leader, inspiration, friend), Susan, The ASPECTJ team, Nic, Vishnu, David, Bonus, Matthew, Chris A, Kostas, Nikos, Neil, Robert, Johnny, all of SLURP, ICU Juggling, Barbara C, Chris H, Magdalena, Murias & Christy, Nathania, The Swing Bunch, Natty, Dr Mithi, Denise, Xerox PARC, Deutsche Bank, 7irene Ltd, Imperial DoC, Computer Science at Kings, Ratio Group, Steve Cook, John Daniels, Peter Messer, Ian Vlaeminke, Susan Anderson and William F Pickard.

Chapter 1

Introduction

This thesis presents a formal model for the semantics of ASPECTJ. Following the philosophy of aspect oriented development, the model is divided into parts presented as chapters of this thesis, each of which addresses a separate concern. Each part is a complete language containing the relevant features. In addition, the languages form a family; derived from a common root, they are presented here rather like classes in an object-oriented program, with the common definitions inherited to avoid unnecessary duplication. Each chapter is divided into sections that also separate concerns: each calculus is presented in terms of its syntax, functions that project values from the syntax, type system, rules for soundness and well-formedness, and operational semantics. The aim of this work is to present a model that covers ASPECTJ's main areas of innovation in a faithful, simple and precise way.

1.1 Semantics

A language's semantics is a description of how programs in that language behave. A semantics describes the execution of programs in general, and may be used to predict the behaviour of specific programs. Formal semantics can be used to provide precise machine-independent concepts for describing programs, unambiguous techniques for specifying programs and rigorous theories to support reliable reasoning. Gordon [33] describes three uses for formal semantics techniques:

- providing precise machine independent concepts
- providing unambiguous specification techniques
- providing a rigorous theory to support reliable reasoning

In defining concepts, a semantics is a model of the meaning of programs or program fragments. It is a model in the important sense that it is not the real thing. A semantics abstracts away some of the details of how programs do what they do, i.e. it avoids the need for efficient, or machine specific implementations and, usually, it models only a subset of the language, retaining the most important features.

One way to deal with programs and their behaviour is to run them and observe what they do. Sometimes, however, this is not practical because, for example:

- Different implementations behave differently and no single system has been nominated as referent.
- The language has not yet been implemented, i.e. the appropriate tools (compiler, interpreter, virtual machine, etc.) have not been written. A semantics presents information to those who must write such tools, it enables them to verify their work, and may also inform the design of the tools themselves.
- It is not certain that the tools available behave correctly. Language tools are programs and where there are programs, there are bugs as a fact of life. When the behaviour of language tools varies or is unintuitive, a semantic model can be used to verify the correct behaviour (provided, of course, the semantics is correct).

Other reasons to create a formal semantics include:

- It is not certain that the language guarantees some properties, for example, type soundness: Eiffel (as described in [53]) was found to be type unsound [18] years after its deployment.
- The design and refinement of some language features requires a formal model (as was the case with *Fickle*[25]).

Writing a semantics is generally easier than writing language tools since technical considerations are abstracted away.

Without the technical detail there is less to comprehend. A semantics is a useful way to understand a language. Programmers frequently learn a language from descriptive narrative (such as manuals or books) and example programs. If the language is similar to one the programmer knows already they will probably dispense with the narrative and just use the examples (until they get stuck); if the manuals and books are well written, they will contain lots of examples anyway. Ultimately, however, there are things that are not easily learnt from examples, perhaps because the point is hard to illustrate or simply because the example does not yet exist, or because it is not well understood: sometimes language designers have not been fully aware of the details of how their languages work. These things warrant deeper study; language semantics is the vehicle for such study. Ideally studying semantics should not require the student to first comprehend difficult concepts from mathematics. Neither should its meaning be hidden inside an opaque presentation.

There are a several ways to describe the semantics of a language; this thesis uses a formal operational semantics: a popular choice for work on object-oriented languages [2, 24].

An alternative might have been denotational semantics [66, 72] concentrates on the meaning of programming constructs, rather than on the way they are computed. Each construct is given a denotation: a mathematical entity that models its meaning. In practise, these denotations are functions, the range and domain of which are often complex structures. Denotations for constructs in very simple languages may, in fact, be functions that return the result of computation. The denotational approach can be

applied to more unusual aspects of computation, for example method lookup in OO languages [19].

In the late '80s and early '90s, several researchers tackled the development of denotational semantics for OO programming languages. A first attempt at a denotational semantics of an OO language with multiple inheritance was given in [15]. In [19], a more advanced model is given, based on an intuitive motivation of inheritance as a mechanism for deriving modified versions of recursive definitions. The domains describing classes need to represent the notion of *self* and the notion of inheritance, thus leading to rather complex domain equations. Since the mid '90s, researchers have favoured operational semantics for modelling object-oriented languages [1, 28].

Nevertheless Wand has used denotational semantics to describe the semantics of dynamic join points as defined by ASPECTJ [81].

The operational semantics presented here describes the steps involved in executing programs on abstract machine and run-time system. The execution of an expression is presented as inference rules that establish the truth of rewrite judgements of the form

$$P \vdash e, h, s \rightsquigarrow v, h'$$

Where:

P	is a program
e	is an expression
h	is the heap in which objects reside
s	is the a stack frame that binds local variables
v	is the resultant value
h'	is the modified heap

The presentation for each language comprises: a definition of the syntax of preterms, validity rules for terms, a set of basic functions that project values from terms (used in the definition of more interesting functions to insulate them re-usable on different syntaxes). Types and type inference rules for expressions are also presented.

1.2 ASPECTJ

ASPECTJ is a an extension of JAVA intended to better support separation of concerns. It was developed at Xerox PARC between the mid 1990s and December 2002 when, at version 1.1, it was transferred to an openly developed project at `eclipse.org`. It allows modular separation of concerns that defy modularisation in other languages, because they cut across the natural decomposition into functional modules. Examples of these, so called *cross cutting*, concerns are: error checking and handling, synchronisation, context sensitive behaviour, performance optimisation, monitoring and logging, debugging support, and multi-object protocols.

ASPECTJ supports all JAVA's object-oriented features and adds aspect-oriented features:

aspects are special classes that serve as modules to encapsulate concerns in source code. Aspect classes can contain everything that normal classes can but cal also contain ASPECTJ's additional declarations.

introduction allows one class to declare members that belong to another class. (in this thesis, introduction is called *member introduction* to distinguish it from parent introduction).

declare parents allows an aspect class to change another class's superclass and implemented interfaces. (in this thesis, this feature is known as *parent introduction*).

join points are points in the execution of a program where separated concerns should affect one another.

pointcuts are the language mechanism for specifying sets of join points. Join points can be specified in a number of ways including using notation that refers to the sate of the method call stack at run-time.

advice is ASPECTJ's way of affecting the behaviour at join points. Advice definitions comprise a block of code, a pointcut and a specification of whether the block should run before, after or in place of occurrences of the join points in the pointcut when they occur.

The example below introduces the ASPECTJ language, explains these concepts in more detail together with the associated jargon words, and illustrates the benefits of aspect-oriented programming as as exemplified by ASPECTJ.

Subject observer example

The subject-observer pattern, described in [31], is a response to the problem of encoding a dependency between objects that is required only in the context of a particular application, that is to say, it is not an intrinsic property of any of the objects involved. The pattern uses the power of abstraction available in object-oriented languages to separate the definition of the objects from the definition of the protocol that they implement. An aspect-oriented solution in ASPECTJ shows improved modularity [35].

The classes below illustrate the traditional approach to the subject-observer protocol. First, in figure 1.1, are the generic protocol classes (those that encode the protocol concern).

Next, in figure 1.2 are the application-specific classes that encode the definition of application objects that, additionally, ought to support the subject-observer protocol.

In the example code above,

- `Subject` (figure 1.1) defines a list of registered observers, a method `add` to register new observers, and method `changed` which tells all registered observers that a change has occurred in the subject to which they are registered.
- `Observer` (figure 1.1) defines method `update` by which it is informed of changes in the subject to which it is registered

```
class Subject{
   private List observers = ...;
   public void add(Observer o){
     observers.add(o);
   }
   private void changed(){
     ... for each o in observers...
     ... o.update()...
   }
}

interface Observer{
   public void setSubject(Subject s);
   public void update();
}
```

Figure 1.1: Object-oriented Subject and Observer

```
class MySubject extends subject{
   Model data = ...;
   public void setData(Model newData){
      data = newData;
      this.changed();
   }
   public Model getData(){
     return data;
   }
}

class MyObserver implements Observer {
   MySubject mySubject;
   setSubject(Subject s){
     mySubject = (MySubjct)s;
   }
   void update(){
     ... this.render(mySubject.getData())...
   }
   void render (Model data){...}
}
```

Figure 1.2: Object-oriented application classes

– `MySubject` (figure 1.2) extends `Subject` and so inherits its definitions, it adds the definition of model-specific data, the `set...` method for the model specific data includes a call to the inherited `changed` method to indicate that a change to the subject should be reflected by a change in the observer.
– `MyObserver` (figure 1.2) implements `update` in some way to change its own state in response to changes in the subject.

This well-known pattern is a good example of the use of object-oriented abstraction to modularise complex software. The benefits of this solution are:-

- The definitions of the mechanics of the protocol are separated into the artifacts `Subject` and `Observer`.
- `Subject` and `Observer` contain no application-specific code.
- `MuSubject` contains no reference to `MyObserver` and vice versa: the two are decoupled.
- The protocol classes (`Subject` and `Observer`) are linked with the application classes (`MySubject` and `MyObserver`) by inheritance: the programmer does not need to specify their dependencies except insofar as they extend and implement the protocol classes.

These are laudable advantages, but the object-oriented solution only goes so far toward separation of concerns.

- `MyObserver` explicitly implements `Observer` and contains the implementations of its methods. It is not totally separated from the protocol, therefore; the parameter of the formal argument to `setSubject` has type `Subject`.
- `MySubject` explicitly extends `Subject`.
- `MySubject` contains a reference to `changed` defined in `Subject`. This does not *feel* wrong to an object-oriented programmer since it is normal for classes to refer to methods defined in their superclasses. But it does signify a failure of the object-oriented approach to completely separate the application concern from that of the subject-observer protocol protocol.

There follows a program that illustrates an aspect-oriented encoding of the subject-observer protocol. First in figure 1.3, are the modules that encode the generic protocol.

Next, in Figure 1.4, are the application-specific classes. Finally, in figure 1.5, is an extension of the protocol module that effectively instantiates the subject-observer protocol for the specific application classes.

In this aspect-oriented version of the protocol:

- `Subject` and `Observer` (figure 1.3) are interfaces. They define the operations that objects must support in order to take part in the protocol, but they do not define the interactions that involve them.
- Objects implementing `Subject` must:
  - Allow observers to be registered with `addObserver`.
  - return its internal data with `update`.
- Objects implementing `Observer` must:
  - Accept a subject with `setSubject`.
  - provide an `update` operation with which to respond to changes in the subject.
- `Protocol` is a special kind of class, called an *aspect*, that is provided by ASPECTJ as the module that encapsulates occurrences of ASPECTJ's new features. Aspect `Protocol` defines the following features:
  - `abstract pointcut stateChanges(...)` is an instance of a new feature called a *pointcut*. A pointcut is a declaration that, in the context of a specific program, defines a set of points in the execution of that program. The purpose of a pointcut is to specify points where the execution of code that relates to one concern is affected by code relating to another concern.

```
interface Subject{
  public void addObserver(Observer obs);
  public object getData();
}

interface Observer{
  public void setSubject(Subject s);
  public void update();
}

aspect Protocol {
  abstract pointcut stateChanges(Subject s);
  private List Subject.observers = ...;
  after(Subject s): stateChanges(s) {
     ... for each o in s.observers...
     ... o.update()...
  }
  private Subject Observer.subject = null;
  public void    Observer.setSubject(Subject s){
    subject = s;
  }
  public Subject Observer.getSubject(){
    return subject;
  }
}
```

Figure 1.3: Aspect-oriented Subject, Observer and Protocol

```
class MySubject {
   Model data = ...;
   public void setData(Model newData){
      data = newData;
   }
}

class MyObserver {
  void render(Model data){...}
}
```

Figure 1.4: Aspect-oriented application classes

Pointcuts are used to define the targets for instances of another new language feature called *advice*. An advice declaration can cause a block of code to be executed after[1] any point in the execution of the program that can be expressed as a pointcut.

[1] or before or in place of

```
aspect Impl extends Protocol {
  pointcut stateChanges(Subject s):
    target(s) && set(Model MySubject.data);

  declare parents: MySubject implements Subject;

  Object MySubject.getData(){ return data; }

  declare parents: MyObserver implements Observer;

  public void MyObserver.update(){
    render((Model)s.getData());
  }
}
```

Figure 1.5: Instantiating the protocol for the aspect-oriented solution

This is an *abstract pointcut* which means that it declares a name (in this case `stateChanges`) for a pointcut that is defined elsewhere. This name is used in the definition of an advice in this aspect.

- A list of registered observers is defined for all objects that implement `Subject`: `private List Subject.observers = ...;` This is an example of a new feature called a *member introduction*. Inclusion of "`Subject.`" in the declaration means the feature is defined to be part of `Subject` not part of `Protocol` even though `aspect Protocol` is its containing module.
- `after(...): stateChanges(s)...` is an `advice` declaration. Advice is another new feature of ASPECTJ, that allows for a block of code to be executed when a specific point (known as a join point) in the program is reached. In this case the advice block is to be executed after occurrences of join points defined by the `stateChanges` pointcut.
  The body of the advice, simplified for this example, iterates through the list of observers, and executes each one's `update` method.
- `private Subject Observer.subject ...` is another member introduction. This time the field `subject` is introduced into all classes that implement interface `Observer`.
- `public void Observer.setSubject ...` is a member introduction that introduces method `setSubject` into all classes that implement `Observer`.
- `public Subject Observer.getSubject()...` is a member introduction that introduces method `getSubject` into all classes that implement `Observer`.

- `MySubject` is the class of application-specific objects that will play the roll of subject in the subject-observer protocol. It defines the model data and a method to update it.
- `MyObserver` is the class application-specific objects that will play the roll of observer. It defines the method for updating its state based on the change in its subject's data.

– `Impl` is another aspect class. It extends the `Protocol` aspect and concretely defines the abstract pointcut `stateChanges` declared in its super-aspect.
  – the pointcut `stateChanges` has a formal parameter `(Subject s)` which is used in two ways: it allows advice that is defined on this join point to bind to the `Subject` object involved in the event, and also used within the definition of the pointcut itself to specify the type of the *target* object.

    The definition of the pointcut comprises the conjunction of two expressions: `target(s)` (which means that the object involved is one whose class implements `Subject` – is the type of `s`) and `set(Model MySubject.data)` (which means that the event is an assignment to field `data` – with type `Model` – in an object of class `MySubject`. Altogether this means that all assignments to `data` fields are included in the pointcut.
  – `declare parents: MySubject implements Subject` is another new feature of ASPECTJ, called *parent introduction*, which allows a class to change the parents (superclasses and implemented interfaces) of another class in the program. In this case `Protocol` is adding `Subject` as an implemented interface of `MySubject`. The effect of this is as if `MySubject` had been declared as: `class MySubject implements Subject`.
  – `Object MySubject.getData() ...` is a member introduction that adds method `getData` to class `MySubject` which it must have in order to implement interface `Subject`.
  – `declare parents: MyObserver implements Observer` is a parent introduction that adds `Observer` as an implemented interface of `MyObserver`.
  – `public void MyObserver.update()...` introduces method `update` into `MyObserver`, which it needs in order to implement interface `Observer`.

  So the effect of `Protocol` is to instantiate the pointcut `stateChanges` to refer to changes in state in `MySubject`, and to make `MySubject` effectively implement `Subject`, and `MyObserver` effectively implement `Observer`.

This aspect-oriented encoding of the protocol is somewhat more verbose than the object-oriented one presenter earlier but most of the extra weight arises from syntactic packaging. It does, however, completely separate the application concern from the protocol concern:

– The details of the protocol are encapsulated within the `Protocol` aspect class. `Subject` and `Observer` are required merely to give types to the roles in the protocol. Some presentations of this example in ASPECTJ may make them inner interfaces of `Protocol`, so that everything to do with the protocol is lexically encapsulated by that aspect.
– `MySubject` and `MyObserver` are devoid of any reference to the protocol. Even the join point that occurs when the state of `MySubject` changes is not explicitly marked in that class, but defined in pointcut in `Impl`.
– The protocol can be instantiated many times over with different classes playing the roles of subject and observer, or with the same classes but different definition of `stateChanges`, etc.

⌟

Development of a formal semantics for core features of AOP is a significant step in the development of the language. This thesis contributes a starting point for the study of ASPECTJ by means of its operational semantics. A subset of the features of ASPECTJ

are modelled: introduction of members and parents, before and after advice, join-points comprising method call, field access and field update. These, as explained later in this section, give good coverage of the innovative features of ASPECTJ.

1.3 This thesis

This thesis presents a formal semantics of the language features *introduction* and *advice* in the context of a simple, JAVA-like language. The aim is to model the semantics of ASPECTJ and thereby to provide a useful source of insight into its features, and to provide a base for future development of models of ASPECTJ and related languages. As described above, formal semantics techniques may be used to provide precise, machine-independent concepts, that specify the language unambiguously and provide the basis of a rigorous theory for reliable reasoning. In order that this work be useful in this respect, some care has been taken with the presentation:

- The features are presented as additions to a simple JAVA-like language, which is presented in full, without any extensions, in chapter 4.
- Each feature is presented with examples that relate to realistic programming applications.
- The features are presented one-at-a-time, in the context of a language with the relevant supporting features.
- The features are presented as increments to the base language; the presentation is organised so that the minimum of definitions from the base language need be repeated in each case, thus the presentation of each language focuses the appropriate new feature.

The remainder of this thesis is organised as follows. Chapter 2 gives background to aspect oriented software development and advanced separation-of-concerns mechanisms. Chapter 3 presents the formal model, its scope and coverage (in terms of features from JAVA and ASPECTJ) and the technical paraphernalia used in its presentation. Chapter 4 introduces the j calculus: a simple object oriented language based on a subset of the features of JAVA. This is the 'parent' language from which my aspect-oriented variants are derived. Chapter 5 presents ij, an extension of j to support member introduction wherein one class may define features for another class. Chapter 6 presents pj, which extends ij to support parent introduction wherein one class may define superclasses and super-interface for another. Chapter 7 presents aj, an extension of j that supports the definition of advice: blocks of code whose execution is triggered by the occurrence of predefined events during the execution of the program. Chapter 8 presents observations and conclusions from the work presented herein.

Chapter 2

Background

Development of complex software is facilitated by modular decomposition [62]. With effective modular decomposition, complex systems can be more easily understood, modules can be developed separately and changes to one module should be possible without necessitating changes to the others. Modularisation involves decomposing the system into modules and, later, recombining the modules to create the whole. The formalism in which the system description is encoded must support such decomposition and there must exist tools that can perform the required recombination. Effective modularisation requires the system to be divided according to particular criteria. Each module should encapsulate the design decision relating to a particular concern and separate the details of that decision from the rest of the system. This is known as separation of concerns.

As observed in [16], program modularisation arose from the necessity of splitting large programs into fragments in order to compile them in systems with limited resources. The source fragments could be compiled separately and the resulting binary fragments combined automatically with a linker. Consequently, early forms of modularity were designed for the convenience of automatic linking; compilation dependencies were minimised by the introduction of interfaces for library modules. Interfaces help control dependencies because a module's interface lists the only things that it can depended upon to provide: its internal details are encapsulated by its interface (This is the root of the principle of design by contract [54]). A module's interface is its outside edge. Linked modules in a system, like bricks in a building, can only touch one another at the edges, their insides are hidden and untouchable.

Later it was realised that the benefits of modularity are not limited to solving technical problems of compilation. Modularity offers additional benefits if system's module boundaries are aligned with its conceptual boundaries [62]. As languages have developed the power to encode new concepts, their module systems have developed too; continuing to support the alignment of modules with those concepts. It is not always easy to balance the requirements of separate compilation with those of matching modules to concepts. Modules in SML, for example, correspond to the concept of functors, but do not allow separate compilation [56]. MODULA 2 modules correspond to collections of types, functions and variables, but there are limitations regarding what kinds

of types may appear in an interface [83]; this is to simplify the task of the linker. Modern languages support abstract structures that allow programs to model very closely the concepts in their problem domains. This is most famously true of object-oriented languages where the unit of modularity is the class.

Object-oriented decomposition

In object-oriented software development the structure of a system is derived from the structure that its designer perceives in the concepts of the problem domain [52]. The software comprises objects: encapsulated units of state and function; that are described by a set of classes[1] The objects model the important phenomena in the problem domain. The classes encode an abstraction of these phenomena into concepts according to a particular classification scheme [52] . Classes are also the principle unit of modularity in object-oriented development. Thus the modularisation of an object-oriented system depends on the classification scheme employed by its designer.

Think of the problem space as a plane and classification schemes as orthogonal axes in which it may be divided into strips. Having divided into strips in one axis we may divide the strips themselves in the other, but due to the linear nature of programs, one must come first, this is known as the dominant decomposition [75] of the formalism. In object-oriented languages, for example as illustrated in figure 2.1, classes are the dominant decomposition mechanism, and methods offer a secondary decomposition. Contrast with, say, SML, where functions are the only decomposition mechanism.

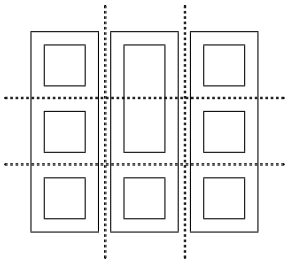


Figure 2.1: Concern space divided (vertically) into classes and (horizontally) into methods.

Many advantages are claimed for the object-oriented approach, such as seamless development [80] or closing the semantic gap between analysis and design [51] and, most famously, software reuse [20]. Much of the benefit of using object-oriented software comes from the flexibility of its decomposition mechanisms which allows code concepts to correspond closely to those in the conceptual problem domain.

The limits of object-oriented modularity

Despite the many advantages afforded by object-orientation, object-oriented languages are limited in their ability to support separation of concerns.

[1] Some object-oriented formalisms allow objects to be described directly without classes. Even in those (e.g. SELF) constructs for classification of objects emerge as an organising principle (e.g. traits [21])

Object-oriented class based decomposition does not give the best structure to encapsulate some kinds of design decisions [36]. Some design decisions, particularly those to do with technical aspects of the system, do not align well with object-oriented structuring mechanisms [44]. Such decisions are not well localised by any classification of problem domain phenomena, nor can they easily be dealt with by an object-oriented classification of the relevant technical domain. For example decisions about synchronisation of accesses to shared resources might require particular actions to be taken before and after every procedure or method invocation. The concepts of method send and method activation can be represented in a reflective object-oriented system but for this to be the basis of a an effective modular decomposition, the target system must support some kind of meta-object protocol [42] that lets the program intercept the real method invocation events. This problem is the motivation behind Aspect-Oriented Programming (AOP) [43, 44] which aims to support modular development based on separation of concerns even when those concerns cut across one another in ways that would normally lead to interleaving and tangling of code.

Different classification schemes might be more appropriate for different aspects of a system. In a non-trivial system, the phenomena can be classified in different ways depending on the perspective or motivation of the designer. Different perspectives might be appropriate to localise the decisions relating to different concerns within the same system. Decisions that are well localised by one classification scheme might be scattered throughout the system by another. Thus it might be desirable to decompose the system into classes according to multiple classification schemes. Most existing object-oriented development environments, however, do not allow multiple classifications of the problem world to coexist [37]. This problem is the motivation for Subject-Oriented Programming (SOP) [37, 60] and its generalisation HYPERSPACES [61] which aim to support decentralised software development based on decomposition into modules each with its own subjective view of the system being developed.

Ungar and Smith [71] observed that as language modularity has improved, the number of dimensions involved in the selection of code to run at a function call, has increased: In imperative languages with procedures and functions, the function name or address uniquely identifies the code to run (one dimension). In object-oriented programming procedure calls become method calls with dynamic binding: the target object's class must be taken into account as well as the method name (two dimensions). In SOP the perspective of the sender must also be considered. This increasing number of dimensions reflects an increasing number of available dimensions of classification. The aim of the *hyperspaces* project and its flagship language HYPERJ [74] is to support modularisation in any number of dimensions.

This ideal situation is illustrated (as well as possible in the two dimensional medium of paper) in figure 2.2: the physical decomposition of the system corresponds exactly to the conceptual one. Unfortunately this ideal is hard to achieve as problem domain concerns rarely align with physical module systems. In fact, having chosen one classification scheme that neatly separates one set of concerns, there is often a set that cut across them. This is the motivation for aspect-oriented programming.

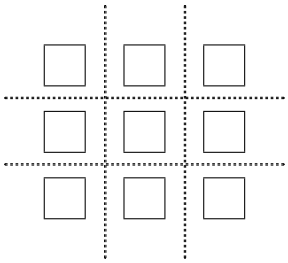


Figure 2.2: Concern space decomposed in multiple dimensions (in this case only two, for purposes of illustration)

2.1 Aspect oriented programming

AOP addresses the problem of achieving good separation of concerns using the mechanisms for modularity in existing languages. AOP recognises the strengths of existing module schemes for capturing most kinds of concerns in a system. But it also recognises their weaknesses. Some concerns do not align well with existing module boundaries, these are called *cross-cutting* concerns. Sometimes what would otherwise be considered a good design – one in which each concern is encapsulated in a module – will have to be radically changed in order to separate a cross-cutting concern. Some times it cannot be done at all: existing module mechanisms simply cannot capture some concerns.

AOP is most often discussed in the context of object-orientation, though it applies to other kinds of languages too. Cross-cutting concerns can be seen in a system when code that addresses a single concern is scattered across the methods of many classes. Code addressing one concern cuts across (horizontally) the traditional class based (vertical) decomposition (see figure 2.3). Often the code of the cross-cutting concern is duplicated many times in different contexts. This phenomenon is referred to as *code tangling*. The disadvantages of this are obvious, they are the disadvantages of poor modularity. The aspect-oriented solution is to supplement the class system with an additional kind of module – the aspect – which can separate out cross-cutting concerns. Aspects can contain features that affect the structure and behaviour of a program in places determined declaratively (rather than lexically as is the case with most forms of modularity). Member and parent introduction declare changes to the members and parents of other classes, and advice change the behaviour of expressions at locations determined by *pointcuts*: declarations that identify certain points in the execution of the program. Thus the extent of these aspect-oriented features can be encapsulated within aspects, and but their influence can cut across the program: corresponding to cross-cutting concerns (See figure 2.4).

Aspects are combined with the classes at compilation by a special kind of linking phase known as *weaving*. Weaving resolves the dependencies between aspect code and the object-oriented code that it augments.

Support for AOP is provided for JAVA by the ASPECTJ language from Xerox [7]. ASPECTJ augments JAVA with two new kinds of class member for defining cross-cutting code: *member introduction* allows one class to define fields and methods that belong (as

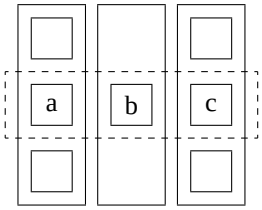


Figure 2.3: Code belonging to a single cross-cutting concern (represented by the dashed box) is scattered across many classes.

members) to another class. *advice* allows specification of code blocks to be executed when specified methods are invoked on instances of specified classes. Cross-cutting relationships between classes in ASPECTJ are resolved at compilation by a composition process known as *weaving*. AOP supports separation of concerns because a number of related members (including the two kinds of advice) can be encapsulated inside a class. Such a class is called an *aspect*.

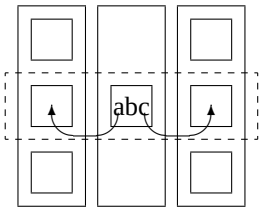


Figure 2.4: Aspects, though physically aligned with classes (vertically) have an effect on a program, the scope of which is controlled by pointcut declarations that can cut (horizontally) across the conventional modularity. The dashed box represents a conceptual cross-cutting module corresponding to the aspect in the centre.

2.2 Other approaches

2.2.1 Subject oriented programming

SOP supports improved decomposition by providing a new dimensions in which to divide software artifacts. Traditional object-oriented development involves decomposing the domain vertically into classes. SOP adds to this the ability to decompose horizontally into modules called *subjects*. A subject comprises a number of, typically smaller, related classes each defined where the subject intersects with a class in the traditional vertical decomposition. A class may be encapsulated in a single subject or divided over several with some of its members being relevant in each subject's perspective. A member may appear in more than one subject if it describes a conceptual property that

is relevant in more than one subject's perspective. The separately developed code that implements such duplicated members must be combined when the subjects are composed. Subject composition is controlled by rules known as composition expressions that say which classes and class members correspond and how they are to be combined. The composition expressions along with the input subjects are processed by a linking tool called a subject compositor which generates the new composed subject. To simplify the task of writing composition expressions common patterns of composition can be identified to form a library of reusable compositions. Two such patterns have been identified: *merge*, where corresponding definitions from different subjects are combined, and *override* where definitions from one subject override those of from another.

Harrison et. al. set some ambitions goals for tools to support SOP [37]. Subjects should be capable of separate compilation and distribution so subject composition must be possible on compiled binary subjects. To manage inter-subject dependencies some kind of interface is must be distributed with a binary subject. The proposed solution, known as subject labels, detail all composable elements – classes, methods and attributes – within a subject, giving their names and signatures [59]. These names are used to specify correspondence among the composable classes, methods and attributes. A subject label has an observable structure that reveals which methods and attributes are contained by which classes in the subject. Members of a class in one subject can be combined with those from another subject. Methods, for example, can be composed in such a way that when either is called both are executed. Thus subjects reveal the insides of classes, and subject-composition allows software structured with classes to be composed in ways that cut across their structure.

Support for SOP in C++ has been implemented in an enhanced version of IBM's visual age C++ compiler [40, 38]. In this system, subjects correspond to namespaces[2] so that classes with the same name (intended to model the same concept) can be defined in different subjects. Composition rules are written in a simple rule language which supports specification of corresponding classes and members, and includes support for the merge and override rules mentioned above.

Subject-oriented subject-observer example

A subject-oriented implementation of the subject-observer protocol would comprise classes described below. The example is presented following, roughly, the notation of HYPERJ [74]. the protocol is encoded simply in figure 2.5, classes `Subject` and `Observer` are as they would be in the object-oriented solution. The application classes `MySubject` and `MyObserver` are shown in figure 2.6, they are free from references to the protocol. Finally, the instructions for the HYPERJ program to combine these classes is given in a *hypermodule* file like the one in figure 2.7. The hypermodule file specifies the details of the composition of multi-dimensional modules from the source artifacts. The idea is that the source files represent single-dimensional decompositions of modules whose complexity requires that they can only be described in a multi-dimensional space. The details of how this works are beyond the scope of this discussion but, in essence, the excerpt in figure 2.7 says that features (such as

[2] Though not exclusively, another way to specify a subject is to list the classes and functions that belong to it.

classes and methods) should be merged if they have the same name, that the `setData` method and the `changed` methods should be merged even though they have different names, and that when a call to `setData` causes `changed` to run as well, the body of `setData` should run before that of `update`. In addition, `Subject` and `MySubject`, and `Observer` and `MyObserver` are explicitly stated to correspond, which means they will be merged by the compiler.

```
class Subject {
  public void ddObserver(Observer o){
    observers.add(o);
  }
  private void changed(){
    ... for each o in observers...
    ... o.update()...
  }
}

interface Observer {
  public void update();
}
```

Figure 2.5: Protocol classes for SOP subject-observer

```
class MySubject {
  Model data = ...;
  public void setData(Model newData){
    data = newData;
  }
}

interface MyObserver {
  void render(Model data){...}
}
```

Figure 2.6: Application classes for SOP subject-observer

⌟

2.2.2 Self and US

Ungar and Smith created an experimental implementation of SOP on top of their SELF language. The result, known as US [71] is an object-oriented language in which method dispatch is decided based on both the class of the message's receiver and the perspective of its sender. One of the main motivations for the design of US is the principle of perspective-receiver symmetry. Loosely speaking, this states that the method lookup mechanism should depend on the sender's perspective in much the same way that it

```
...
mergeByName;

order operation Feature.protocol.changed
          after Feature.data.setData;

equate class Feature.data.Subjct,
             Feature.protocol.MySubject;
equate class Feature.data.Observer,
             Feature.protocol.MyObserver;

...
```

Figure 2.7: A HYPERJ Hypermodule specification

depends on the class of the receiver. For this to work, perspectives are reified into objects and the syntax of message sending is extended to include a perspective object expression. Object definitions[3] are constructed in layers that correspond to different perspectives that client objects may have at run time. The perspective object is used to decide dynamically which perspective the sender has and, therefore, what method code should be executed in response.

US offers an interesting model of structures to support a dynamic SOP at run-time. Various policies are discussed for the default handling of perspectives so as to avoid the clutter and conceptual overload of having to specify contexts explicitly with every call. The perspective-receiver symmetry principle has led to a design in which: object definitions are split into layers, the layers are arranged into an hierarchy where attributes and methods defined on the outer (lower) layers mask those defined on the inner (higher) ones. This layer hierarchy deliberately resembles the object hierarchy where an object's method and attribute definitions can override those of that object's prototype. In most class-based languages there is a mechanism (such as `super` in JAVA) that gives the overriding method access to the superclass context so that the overridden method can be called. In US this is achieved by arranging (dynamically) for the overriding method to send the same message that was used to invoke itself but specifying the appropriate perspective object to select the code from the overridden layer.

2.2.3 Role modelling

Reenskaug's role modelling[4] [65] develops the same themes as AOP and SOP for the task of system design. The idea is to construct the design of a complex system in The form of a number of abstractions called role models. Role models capture the essential collaborative behaviour of some aspect of the system. Participating objects are represented by roles, and interactions between the roles are described in detail. A role can be played by one or many objects and serves as a partial description of those

[3] SELF, and hence also US are classless object-based languages

[4] also known as OORAM [65] and OORASS [64]

objects that may play it. The description of a role (called a *role class*) is similar to the partial description of classes that occur in subjects. It is suggested [45] that, to implement a role model, classes be constructed for the objects that will play the roles. These classes are called *intrinsic classes* and comprise the combined behaviour of all the role classes that the their instances must play. The process or combining role classes to form intrinsic classes is called synthesis [63].

The similarity of role modelling to AOP and SOP is well known; to quote [5] : 'A role is considered an aspect or view of an object in a particular context'. It allows the designer to capture the essential interactions between objects that are present in the system without requiring him to make unnecessary decisions about which objects will be involved in those interactions. Since the focus is on the interactions and not on the objects themselves, the main unit of modularity is the role model. Since interactions take place between objects all interesting role models include more than one role. Thus the modularity cuts across class based decomposition. Roles describe the interaction behaviour of objects and not their identity so many roles in a role model might serve to specify complex interactions between one object and itself.

Synthesis is a general term use to describe a number of ways to derive new models from old. It includes projections for abstraction and refinement by hiding or introducing detail. It also includes the notion of composing two or more models in which some roles correspond and therefore should be combined. The details of this composition process are formalised somewhat in [4] where interfaces are modelled as sets of entities so that combination of roles can be modelled by forming the union of interfaces.

Roles are characterised in terms of paths to other roles, and the services that they will request along those paths. This comprises what is called the role's output interface. Role models can also contain classes which, as in conventional object-oriented design, specify the kinds of objects that can be instantiated in a system in terms of a set of public methods and attributes that represent services that can be requested of it by message sends. This is known as an input interface.

Since all roles will eventually be 'played' by objects in the system that are described by intrinsic classes, it is important to be sure that those classes define at least the required services and support the necessary paths. Classes have both input and output interfaces. If a class is synthesised from a number of roles its output interface will be the union of its component roles and its input interface will be the union of the input interfaces on paths leading to it from other roles. Thus synthesis may be used to create classes whose interfaces match the requirements of them in a system.

Role modelling concentrates on the things that happen in the space between objects rather than on the internal details of the objects themselves. Classes that are synthesised from roles resemble abstract classes or interfaces in Java in that they specify a contract of abstract methods representing the services that the class defines. These methods must be implemented before a system of objects can be instantiated. Role modelling offers insights into the usefulness of a composition-oriented approach. It does not, however, address the problems of separation of concerns that arise when dealing with method the statements that implement methods.

Various attempts have been made to add some sort of role mechanism to conventional object-oriented systems. Many of these focus on how to implement varying behaviour

at run-time in ways akin to the state or strategy patterns of [31]. [34] is a little more interesting since it uses roles as a way of managing the evolution of schema in object-oriented databases. Their implementation, in SMALLTALK, creates classes for every combination of roles that might be extant at run-time and moves objects between these classes using SMALLTALK's `become` method. Roles in this system also act like views [67] so that it is not possible to access properties of a one role when the object is viewed from the perspective of another.

More recently, the problem of changing an object's class at run-time (which would be needed to support direct native implementation of role models where roles change at run-time) has been given a formal treatment using similar techniques as this thesis [25].

2.2.4 Inheritance

Inheritance in object-oriented languages provides a powerful form of linking between modular units (classes), which is different from simply joining modules at their interfaces. A subclass gets to be joined to the *inside* of its superclass: it can access its private[5] members and change the behaviour of its methods. A subclass definition is combined with that of its superclass by means of a kind of synthesis where the new artifact has some characteristics of each of its components. Clients of the superclass – classes that uses the services available through its interface – are combined simply by having their message sends bound to appropriate method invocations. The difference between these two kinds of linking is captured by Meyer's Open-Closed principle[55] which states that modules should be capable of being both *open* and *closed*, and explains these terms thus:

> - A module is said to be open if it is still available for extension. For example, it should be possible to expand its set of operations or add fields to its data structures.
> - A module is said to be closed if it is available for use by other modules. This assumes that the module has been given a well-defined, stable description (its interface in the sense of information hiding). At the implementation level, closure for a module also implies that you may compile it, perhaps store it in a library, and make it available for others (its clients) to use. [...]
>
> The need for modules to be closed, and the need for them to remain open, arise for different reasons. Openness is a natural concern for software developers, as they know that it is almost impossible to foresee all the elements – data, operations – that a module will need in its lifetime; [...]

The technicalities of inheritance as a way of combining class definitions were explored in work by Cook [17, 19], which revealed flaws in Meyer's design for EIFFEL [18].

[5] *Private* in the sense that they are not accessible to clients via the published interface. In some languages, such as C++ and JAVA, subclasses cannot access members that are marked *private* but they can access those that are marked *protected*, which are the ones I am talking about in this case.

Despite the flaws in its type system EIFFEL remains of interest: it has one of the best developed mechanisms for dealing with multiple inheritance[54]. It has a sophisticated sub-language for prescribing how inheritance should work much as PASCAL includes a sub-language for describing data types[82]. If the inheritance language were extracted from the syntax of the target language the result would resemble subjects and composition expressions as proposed for SOP [60]. One can get a taste of this by creating empty subclasses that serve only to combine multiple superclasses in a sensible way.

2.2.5 Mixin inheritance

Mixins are subclasses that are independent of their superclasses. That is to say they are class extensions that can be used to extend more than one superclass [13].

Since mixins are classes that are generic with respect to their supertypes, they can be implemented in object-oriented languages with powerful genericity mechanisms that allow definition of a class that is parameterised over its superclass. This can be done in C++ [70]. Language extensions to support mixin based programming have been proposed for languages such as JAVA [3, 29]. These studies, together with other formal treatments of mixins [30], have helped to formalise the typing requirements for mixins and in particular how types can capture the constraints that a mixin class places on its superclass parameter.

Since any mixin class is a kind of subclass to the classes it extends, a mixin may have dependencies on its superclass. A system of mixins be well-behaved if the mixin classes are only applied to superclasses that can fulfil the mixin's expectations. This can be established with a type check. Since a mixin is a class that is generic with respect to its superclass, the superclass can be specified with a generic parameter (for example `class C[G] extends G {...}`). The generic parameter can be given a type constraint (for example `class C[G->T] extends G {...}` where class `C` extends some class `G` where `G` has a type that is a subtype of `T`[6]) In MIXEDJAVA this effect is achieved using interfaces to define the constraining types and a special syntax for mixin classes. For example, in MIXEDJAVA, the simple example above would look something like this:

```
interface T {...}
mixin C extends T implements ... { ... }
```

The term mixin is sometimes used in relation to a practise sometimes employed in languages with multiple inheritance: constructing subclasses by mixing a number of base classes that have been designed to maximise their composability.

Many multiple-inheritance mechanisms based on mixins use linearisation of superclasses to resolve conflicts among inherited features. In [12] mixin based inheritance is extended to include the possibility of multiple superclasses. A layer of renaming is

[6]This pseudo-code for constrained generic types looks a bit like that of EIFFEL. Sadly EIFFEL does not allow superclasses to be specified as generic parameters.

necessary to allow a mixin to refer separately to different features that were inherited with the same name. Flexible operations on mixins offer the programmer more control over renaming, sharing, joining, etc. of inherited features [78]. Taken to its extreme this might result in the creation of an elaborate inheritance sub-language like that of EIFFEL.

A mixin approach to modularity has been applied to functional languages, specifically SML [27] where composable units are proposed to support recursive definitions that cross module boundaries. These mixins are combined by a standard composition operator and then closed to create conventional SML modules. Since there are no classes in SML, and therefore no subclassing mechanism with which to extend existing structures, Duggan and Sourelis devise an extension mechanism that allows new cases to be added to data type and function declarations. This is achieved by the use of a construct called `inner` which, when used in the body of a type of function definition in a module denotes the cases defined in other mixins that will be composed with the current when a module is formed. This facility deliberately mimics the `inner` call system for combining method bodies in BETA.

Mixin inheritance is more flexible than most fixed forms of class inheritance and it has been proposed as a basis for implementations of SOP [50]. Smaragdakis proposed a technique for implementing layered object-oriented designs that uses mixins as the basic composition mechanism [69]. A layer is similar to an AOP aspect or an SOP subject in that it encapsulates a concern that can cut across many objects. The technology proposed entails:

- mixins (subclasses parameterised over their superclasses)
- nesting of mixins
- mixin layers (a mixin with encapsulated inner mixins whose parent parameter encapsulates the parent types of all its inner mixins)

The main point claimed for this work is the importance of nested mixins. The outer mixin plays a role similar to the aspect in ASPECTJ: encapsulating a collection of related code artifacts that, together, implement a cross-cutting concern. The inner mixins play a role similar to advice, allowing methods of the superclasses to be augmented using the host language's inheritance mechanism for composition; methods are overridden but the overriding method body may call the overridden one using `super` or an equivalent mechanism. By deciding where to put the call to `super` a programmer can encode idioms similar to before, after and around method composition in ASPECTJ.

FLAVORS

FLAVORS is an object-oriented programming system built on top of LISP [39, 57]. It was a predecessor to, and influence upon, the design of CLOS (discussed below). Classes in FLAVORS are called flavors. Dynamic binding of function calls to the executed code is achieved by means of *generic functions*: function definitions that have many implementations (know as *methods* in FLAVORS and CLOS but referred to as method implementations in this thesis to avoid confusion with the conventional object-oriented meaning of "method"). FLAVORS supports multiple inheritance under the

moniker ”mixing flavors”: a flavor obtains the union of its own and its inherited generic function and instance variables (when more than one instance variable have the same name, they are merged into one). The method implementations are combined by writing LISP code to decide which blocks of code should run, in what order and how their results should be combined. A standard approach is to nominate some method implementations as *before* or *after daemons*, these are marked with a symbol in their declarations, and are run either before or after the primary method implementation (one that is not a daemon) that has been selected to run. The style of programming with FLAVORS: using multiple parent flavors – each encapsulating a separate concern or aspect of the program – to mix up the desired class, is an early forerunner of aspect oriented programming. It has influenced, via CLOS (see below), the evolution of the design of ASPECTJ.

2.2.6 CLOS: Method combination

CLOS [11] is the Common Lisp Object System, part of the COMMON LISP programming language that supports object-oriented programming. CLOS implementations are metacircular interpreters written in CLOS: the definition of the language is encoded by means of an object model of the language itself written in CLOS. CLOS programmers have access to this meta-object model by means of its meta-object protocol [41].

CLOS's features include:

- Defining classes (class-like data types) with possibly multiple superclasses, *slots* (attributes) that have default initial values and accessor functions.
- Multiple inheritance is supported in CLOS; rules are used to determine a total order among the superclasses of each class, this is used to resolve the way in which inherited methods relate to one another. Rather than overriding inherited methods, they are combined according to rules described below.
- Instantiation of new objects with initial values for slots that might override the defaults defined in the classes.
- *Generic functions* for dynamic binding of methods. In CLOS a generic function is just the specification of a function, this declaration might have many implementations, each known (in CLOS) as a *method*[7], that are applicable under different circumstances. Method implementation are defined like COMMON LISP functions but with the addition of optional type annotations on the formal parameters. To be precise, these annotations are really CLOS class names. When a generic function is called all its method implementations are collected and those whose typed formal parameter lists match the types of the actual parameters are executed.
- Method combination is the mechanism for specifying the way in which methods should be combined when more than one of them are applicable for a particular generic function invocation.

Method implementations may be given “roles” that determine how they fit together with other method implementations. These roles include `:before`, `:after` and `:around`.

[7] These will be referred to as method implementations in this thesis to avoid confusion with the normal object-oriented notion of method.

A method implementation without such an annotation is a *primary* method. Methods marked `:around` are executed first. The most specific (according to the rules for prioritising superclasses) is called first. The method body may invoke `call-next-method` to call the next most specific `:around` method. When the least specific `:around` method calls `call-next-method` the `:before` methods run (in priority order) followed by the primary methods followed by the `:after` methods. The primary methods are chained together using `call-next-method` like the `:around` methods.

CLOS's mechanism for combining method implementations has many similarities to the way in which ASPECTJ's advice are combined. Another similarity between these two languages is in the matching mechanism used to select which method implementation should run. In ASPECTJ, advice bodies are selected based on a similar match between the types used in patterns in pointcut expressions (think of these as formal parameters) and the types of values associated with the actual join point when it occurs (think of these as actual parameters).

These similarities are not surprising since the principal designer of ASPECTJ, Gregor Kiczales, was also the principal architect of the CLOS meta-object protocol [43]. Indeed, in a talk at the Aspect Oriented Programming Workshop at ECOOP'99 he explained that one of his motivating principles in the design of ASPECTJ was to provide easy-to-use programmer-access to the power of meta-programming by making ASPECTJ package up some of the frequently-used idioms of meta-programming. The relationship is elegantly summed up in [73]:

> "Thus it is through aspect-oriented languages that we tame the complexity and power of metaobject protocols. An aspect language is the interface to the functionality of a metaobject protocol".

2.2.7 Composition Filters

The composition filters object model [8, 9] was developed in the early nineteen nineties, by a group at the university of Twente in the Netherlands, as a general-purpose computation model offering high-level abstraction techniques that support language features for various application domains such as concurrency and synchronisation. The model is based on object-oriented programming, and comprises objects, described by classes, with fields (instance variables) and methods, some of which (those with no parameters that return a Boolean value and do not side-effect the concrete state of the object) are designated *conditionals* and are used to indicate that the object satisfies a predicate that means it is in a named state. Objects communicate by sending one another messages. What happens when an object sends or receives a message is determined by the additional features of the model: an object is "wrapped" in an *interface part* that encapsulates composition-filters' novel features, namely message filters. For both the incoming and outgoing messages, an ordered set of filter declarations is first matched against the message (patterns match message selectors and target objects) and, if they match, their effects might be to modify the message (by substitution) or to redirect it, or reject it, etc. The interface part also contains its own variables and references other external objects.

Inheritance is not provided as a basic language feature but it is encoded using filters: the final filter in an input filter list dispatches the message to the encapsulated object causing the corresponding method to be invoked. This filter is designed so that there is a list of candidate objects to receive the message, if no corresponding method is found in the first then the second is checked, and so on. The subsequent potential message receivers effectively provide parent part-objects that are delegated to. The fact that there may be many of these means that multiple inheritance is supported.

Since a filter matches message by pattern, and since an intercepted message can perform such actions as invoking other messages, it is clear that there is a similarity between composition-filters technology and aspect-oriented programming technology. The objectives are similar too, as discussed in [10], where the model is extended slightly with *superimposition* for composing filter interface parts. This latter extension is consistent with the composition-filters model, in fact it is more of a language technology feature than an extension to the model. It enables the powerful matching and manipulation of messages by filters to be used in a way that crosses module (i.e. class) boundaries and, thus, shows it to be a form of aspect-oriented programming.

Chapter 3

A formal model of ASPECTJ

The subject of this thesis is a model of selected features of ASPECTJ in the form of a family of related models, each one of which presents one interesting aspect of ASPECTJ in the form of a calculus that models the important language features and a definition of its operational semantics. This chapter describes the history of the development of ASPECTJ and explains why a formal model of its features is useful. It also describes the technicalities of the model including the selection of features, the structure of the model and the notational conventions and mathematical techniques used.

3.1 A Model of ASPECTJ

This section describes the motivation for the formal model and the selection of ASPECTJ's features for inclusion.

3.1.1 Motivation

The move of the ASPECTJ project to `eclipse.org` reflects the view of the ASPECTJ team, that the initial research and prototype development of the language is complete. ASPECTJ was developed in an iterative, interactive way, with many prototype versions released to the user community. The ASPECTJ development team were constantly responding to the feedback of those users. The direction of the language's development was always steered by the ASPECTJ team though it was influenced by the user community. There were several significant changes to the language and its features during its development. This made it something of a moving target for the development of semantic models. As of December 2002, ASPECTJ has reached a significant level of maturity, with a growing number of commercial users[1].

[1] As of November 2003 the language is at version 1.1.1, released in September 2003 and commercial acceptance continues to increase.

Before the stability of version 1.1 of ASPECTJ there were many changes due to the iterative approach to language design. The most significant of these are described below:

Two argument calls

In versions of the language before version 0.7beta, the `calls` joinpoint designator took two arguments like this:

`calls (`C, M`)`

Where C is the name of a class type and M is a method declaration. The meaning of this is to designate calls to any method matching M on an object of type C as belonging to the current join point. Due to the structure of method declarations, M has the format:

M ::= T_r [T.]m (T_x)

Where T_r specifies the return type, T is the (optional) type in which the method was defined, m is the method name and T_x is the type of the formal parameter[2]. So the total information that has to be supplied to specify a `calls` joinpoint is: C, T_r, T, m and T_x. Note that C is matched against the dynamic type of the receiver, T_r and T_x are part of the signature of the method and therefore matched statically, whereas T, if present, indicates the class in which the method was defined (so as to distinguish it from other, semantically distinct, methods with the same name and signature) and therefore affects the static selection of the method. Of course C and T could be the same, since in a strongly typed language the object on which the method is invoked must belong to a class that has the method defined upon it, though T might be an ancestor of C.

The subtle distinctions between the ways in which these types are interpreted are potentially confusing to the language's users. A formal model of the semantics of pointcut designators removes such confusion.

In subsequent versions of the language `calls` was simplified to take only one argument[3]:

`calls (`M`)`

Now only static information about the method is supplied. Selecting a pointcut based on the dynamic type of the receiver is possible using a new designator (introduced at the same time that `calls` became a one-argument designator) called `target`; this ASPECTJ pointcut:

```
call(void Subject.setData(*)) && target(MySubject)
```

specifies occurrences of calls of method `setData` defined in class `Subject` when the object receiving the call is an instance of `MySubject` (Where `MySubject` is a subclass of `Subject` and inherits method `setData`.

[2] Assuming only one formal parameter is present. In ASPECTJ this was a sequence of types but one is used here for consistency with the model later in this thesis

[3] it also changed name from `calls` to `call` in version 1.0alpha1

Binding join points to variables

In ASPECTJ the specification of variables that will bind to values associated with join points is mixed with the specification of join point patterns themselves. In ASPECTJ before version 1.0alpha1 the notation looked like this:

```
before(Tr r, C d, Tx x): calls(r d.foo(x)){
   // ... r, d and x in scope ...
}
```

Formal parameters are declared in the arguments of the advice and the parameter names replace their types in the pointcut designator. This mixing of function – the slots in the pointcut designator serve to hold either variables or type patterns – was potentially confusing to language users. Especially so when the type pattern specifies a pattern (using wildcards) and not just a type. This is, to a certain extent, unavoidable without making the language verbose and redundant: the alternative would be to include separate patterns for specifying the type patterns and the variables.

Later versions of ASPECTJ adopt an ingenious compromise, illustrated here:

```
after(C d, Tx x) returning(Tr r):
 calls(* d.foo(x)) && target(d) && args(x){
   // ... r, d and x in scope ...
}
```

The optional `target` and `args` designators are matched dynamically against the types they specify. It is only these, not the slots in method declaration patterns, that can be used to bind variables, Returned values are handled slightly differently: the `returning(...)` notation is an optional form of the `after` advice declaration syntax; it specifies that a returned value of the specified type is expected.

In aj the two are kept separate. One of the aims of the j family is simplicity; some features are omitted from the model for the sake of this aim. A simplified but different mechanism is used for binding variables to joinpoint values. The binding is implicit, not explicit: certain variables are always in scope inside an advice body, whether or not the programmer needs to use them. The programmer has no control over the names of these. This leaves the definition of pointcuts uncluttered by variable definition. Any mechanism for binding variables to these values would require matching of the values to the names in some way. The complexity of such a mechanism outweighs the advantage of including variable naming in the model.

3.1.2 Included features

This thesis focuses on a subset of the features of ASPECTJ, chosen to be representative of the major innovations in the language.

Introduction and parent introduction

ASPECTJ allows members (fields and methods) to be declared in classes other than those they belong to. An aspect class can contain a member declaration for a member of any class. An aspect class can also declare changes to the superclasses and interfaces from which another class inherits. These features are modelled in ij and pj respectively. Declaring members for other classes is historically known as *member introduction*; in this thesis, declaring changes to the inheritance hierarchy is referred to as *parent introduction*. To support encapsulation of cross-cutting features, in ASPECTJ introduction must be declared inside aspects. The languages in this thesis do not attempt to model the encapsulation of aspect-oriented features, but to describe their semantics. There is no semantic requirement for aspects to exist to support either kind of introduction so ij and pj allow introduction to exist in normal classes. ASPECTJ allows a member introductions to be annotated as `private`. This means that the introduced member can be accessed only from within the aspect that introduced it. This thesis does not deal with the issue of member visibility so this feature is not addressed.

Aspects and aspect instantiation

ASPECTJ introduces a new kind of module called an aspect. An aspect is like a class: both a type and a template for the creation of objects. Aspects are also the modular units of aspect-oriented programming. Classes in ASPECTJ are similar to normal JAVA classes: they can contain field and method definitions. In addition they can contain the definitions of the ASPECTJ's aspect-oriented features: introduction, advice and pointcuts.

Unlike normal JAVA classes, aspects cannot be instantiated by `new` expressions in the code. Instead aspect instances are created automatically according to rules. Aspect instances may be bound to the `this` parameters of executing advice bodies. By default an aspect class is a singleton: an instance is created as the aspect's class file is loaded. Alternatively in ASPECTJ, aspects can be annotated to indicate that instances should be created, for example, one per object that matches a criteria defined in a pointcut, or one each time the flow of control passes a specified point. In aj, for the sake of keeping the model simple, Only the default case is modelled here: singleton aspects. Aspect instance creation is modelled by preparing the heap before execution of a program with instances of all the aspect classes.

Inheritance among aspects is allowed in ASPECTJ, though aspects and classes must belong to separate hierarchies. Aspects may inherit only from abstract aspects. Abstract aspects may contain abstract pointcut definitions: named pointcuts that do not have an associated pointcut expression to define them. Concrete sub-aspects must provide implementations for abstract pointcuts. Both aspects and classes can implement interfaces. Named pointcuts are not modelled in aj, so there is no reason to model aspect inheritance. In aj, therefore, all aspects are assumed to inherit directly from `Object`, the top of the class hierarchy, and both classes and aspects can implement interfaces.

ASPECTJ allows aspects to be declared as `privileged`, which means they can access private fields of other classes; it also allows an aspects to be declared as dominating

others, this affects the order in which their respective advice will run if they are defined on the same join point. aj does not deal with the issue of member visibility, so aspect privilege is not relevant and therefore excluded, nor is aspect domination included: advice order is modelled, but using a simpler mechanism since aspect domination is used rather infrequently, in practise, to control execution order when some there is some dependency between aspects.

Point-cuts

ASPECTJ supports the definition of named pointcuts, including abstract pointcuts. Named pointcuts are used by reference in the definition of advice. Named pointcuts are primarily a convenience to avoid duplication of pointcut expressions. In order to keep aj simple, named pointcuts are not included. The result is that sometimes it might be necessary to duplicate pointcut expressions in aj when expressing advice which can be more succinctly expressed using named pointcuts in ASPECTJ.

ASPECTJ has a rich pointcut sub-language with a number of primitives and combinators for describing sets of join points. The primitives are listed below, the combinators are: `!` (compliment), `&&` (intersection) and `||` (union).

call – calls to methods and constructors

execution – execution of methods and constructors

initialization – execution of a new objects initialiser code

staticinitialization – execution of a class's static initialiser code

get – field access

set – field update

handler – execution of an exception handler

within – all join points defined within a specified class

withincode – all join points defined inside a method or constructor with the specified signature item[cflow] – all join points in the control flow of a given join point (inclusive of the defining join point itself)

cflowbelow – same as `cflow` except exclusive of the defining join point

if – all join points at which the specified arbitrary boolean expression holds

this – all join points where the executing object is an instance of a specified type

target – all join points where the target object is an instance of a specified type

args – all join points where the target object is an instance of a specified type

The set of potential join points are defined by the first seven of these; the latter eight are restrictions on the set of potential join points. In the interests of simplicity, aj includes a pointcut sub-language which is a subset of ASPECTJ's. The j family of languages does not include constructors or initialisation blocks, nor does it include exception handlers. Method call and execution are sufficiently similar as not to warrant separate study. The potential join points in aj are, therefore: field access, field update and method call, so the primitives pointcut expressions are `get`, `set` and `call`. No pointcut combinators are included.

Advice

ASPECTJ allows the definition, in aspects, of advice that specifies expressions that should be run at join points specified by pointcuts. Advice body expressions can be specified to run before, after or in place of (via the modifier `around`) the join point expression. Around advice may contain a special kind of call called *proceed* that causes the replaced join point expression to be executed. In this way it is possible for advice to modify the incoming values (e.g. parameters to a method call and returned value of a join point). ASPECTJ allows after advice to access (via the keyword `returning`) to the value returned by the join point expression. It also allows variables to be declared for advice that bind to the target object, parameter values and currently executing object (as specified by the `target`, `args` and `this` primitive pointcuts). Simple before and after advice is supported in aj. Though around allows greater control over the join point, an around advice with a call to `proceed` can be seen as before and after advice. Around advice could be used to model before and after advice. I chose before and after advice for aj, since they differ from one another only by the `before` or `after` modifier and thus give a more insightful explanation of the process of choosing advice to run at a join point. Since access to the returned result requires a special construct, that is also excluded. As aj does not include primitives `target`, `args` and `this`, some other mechanism is necessary to expose values from the join point within the advice body. The chosen mechanism is very simple and consistent with the choice for j to allow only one method argument that is always called `x`: certain variables, with well known names, are made available in advice bodies to expose these values.

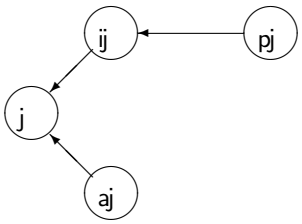
3.2 The j calculi

The body of the thesis comprises a family of calculi that model simple object-oriented programming languages with aspect-oriented extensions. Each calculus presents some important and cohesive concept in the model. The calculi are related by being derived from one another and together they form a family.

3.2.1 The j family

The calculi in the j family are as follows: j is a model of a simple language based on JAVA; the syntax of j resembles that of JAVA to emphasise the representation. ij, pj, and

aj add selected features of ASPECTJ. These calculi are related to one another according to the following diagram.



The arrows indicate derivation:

- j models key object-oriented JAVA features: objects, classes with non-static methods and fields, inheritance, interfaces and simple expressions: method call, field access and update and object creation.
- ij is derived from j, and adds *member introduction* wherein one class may contain the definition of members (fields and methods) for any classes in the same program.
- pj is derived from ij and adds *parent introduction* wherein a class may affect the superclass and super-interface relationships of other classes in the same program.
- aj is derived from j and adds the *advice* construct which allows for the definition of blocks of code that are to be executed at specific points (called *join points*) in the execution of the program.

3.2.2 Parameterised definitions

Many of the definitions – of terms, functions and rules – that these calculi comprise are similar or the same from one calculus to another. In the presentation of these calculi, The aim of this presentation of these calculi is to emphasise what is interestingly different between them. Using the idea of object-oriented style derivation of calculi, definitions of the terms, functions and rules that make up a calculus are *inherited* by those which are derived from it. In this vein, some definitions are parameterised over the name of the calculus to which they belong in order that they may refer to the appropriate calculus-specific versions of other redefined terms (or functions or rules). The calculus name parameter is ℓ.

⌜ For example function GetMethods : $Prog \times ClassName \longrightarrow Declaration^*$ which gives the sequence of method declarations for a named class in a given program (figure 4.12), is defined in each language as:

$$\mathsf{GetMethods}(P, C) = \mathsf{Methods}'(\mathsf{GetDecs}(P, C))$$

Where DeclarationsX gives all member declarations in the class and Methods′ filters out all except method declarations. ⌟

But some languages (e.g. ij, see figure 5.7) have different versions of function DeclarationsX and, consequently, their GetMethods functions are also different, though

their definitions are textually identical. To handle such redundancy, I will present definitions only once and use superscripts[4] to distinguish the functions that belong to different calculi. Reusable definitions are presented in a boxed figure throughout which ℓ is used as a superscript to stand for the language. For example the definition of GetMethods is presented as in figure 3.1, using superscript ℓ. This definition is textually the same in all languages but varies in terms of which version of DeclarationsX is referred to. This is handled by the parameterised presentation. The text of the chapter on j refers to the figure and say that it defines $\mathsf{GetMethods}^{\mathsf{j}}$; substituting j for ℓ. The text of the chapter on ij, refers to the same figure, but says that it defines $\mathsf{GetMethods}^{\mathsf{ij}}$. So a reference to $\mathsf{GetMethods}^{\mathsf{j}}(P, C)$ refers to the function defined as $\mathsf{Methods}'\Big(\mathsf{GetDecs}^{\mathsf{j}}(P, C)\Big)$ and $\mathsf{GetMethods}^{\mathsf{ij}}(P, C)$ refers to the function defined as $\mathsf{Methods}'\Big(\mathsf{GetDecs}^{\mathsf{ij}}(P, C)\Big)$

$$\mathsf{GetMethods}^{\ell}(P, C) = \mathsf{Methods}'\Big(\mathsf{GetDecs}^{\ell}(P, C)\Big)$$

Figure 3.1: Function to project methods from a class in language $[\ell]$

3.2.3 Notational conventions

Syntax

Syntax is defined using a Bacus Naur style notation in which sets and variables that range over their members are introduced together. Syntactic definitions of the form:

v,w $\in S$ ::= *prod*

Define S as the set of all elements that can be constructed from the production *prod*, and that v and w are the base names of variables that range over members of that set. Such base names may be used in the text adorned with primes and/or numeric subscripts.

Sequences

Sequences are most often constructed *and* concatenated by juxtaposition:

$$seq =< head >< tail >$$
$$seq = S1\ S2$$

though, in some circumstances an explicit cons operator is used, for clarity:

$$seq =< head >::< tail >$$
$$seq = S1 +\!\!+ S2$$

[4] and sometimes subscripts

An empty sequence is denoted by ϵ. The length of a sequence S is given by $\#S$. Where possible, names of functions returning sequences of values start with an upper case letter, those returning single values start with lower case. Sequences are written as lists of items without enclosing brackets. Thus, X may stand for a class and also for a sequence of classes with length one. When using variables to bind explicitly to sequences, where the meaning is unambiguous, their names will be pluralised.

So, for example, $X :: Xs$ stands for a list with head X and tail Xs. ⌟

Placing a sequence inside curly set braces, for example: $\{A_1..A_n\}$, defines a set by extension. The same technique is used to eliminate duplicates from the collection returned by some functions that return sequences, for example: $\{\mathsf{Hs}(P, C, m)\}$ refers to the set containing the unique values in the set returned by $\mathsf{Hs}(P, C, m)$.

Ellipses

Two dots ".." is used to stand for the elided middle of a sequence. A sequence may be bound to a sequence of subscripted variables using the following notation:

$$seq = X_1..X_n$$

Three dots "..." is used to stand for the elided end of a syntactic construct such as a method or a class. So a class with name name C may be shows as

$$\texttt{class}\ C\ \ldots$$

In this way the different kinds of class construct defined in this thesis (classes, interfaces, etc.) can be distinguished by matching on this form:

$$\texttt{class}\ C \ldots$$
$$\texttt{interface}\ C' \ldots$$
$$\texttt{aspect}\ C'' \ldots$$

For example in rule [*CLASST*] (from figure 4.14 and duplicated here):

$$\frac{\vdash_\ell\ P\ \overset{\triangle}{uc} \qquad P = X_1..X_k,\ \texttt{class}\ C \ldots,\ X_{k+2}..X_n}{P \vdash_\ell\ C\ \overset{\triangle}{ct}}\ [classT]$$
$$P \vdash_\ell\ C\ \overset{\triangle}{ot}$$

which says that a class name C is judged to be a class type $P \vdash_\ell\ C\ \overset{\triangle}{ct}$ and also an object type $P \vdash_\ell\ C\ \overset{\triangle}{ot}$ for the given program P, if P satisfies rule $\vdash_\ell\ P\ \overset{\triangle}{uc}$ for unique class names, and P is comprised of a sequence of class declarations (each matching a subscripted variable X_n) containing at position n a class (but not an interface or any other kind of class-like construct) with the name C. ⌟

Ellipses bind more tightly than sequence constructors and domain restriction. Thus: $F :: D_1..D_n$ (from figure 4.10) is a patten comprising a sequence of $n+1$ declarations beginning with a field F. And $D_1..D_n/_k$ (from figure 4.11) denotes a restriction of the whole sequence $D_1..D_n$.

Universal quantification uses the following notation:

$$\forall\, var[\in set] \bullet proposition$$

For example, the rule [*UC*] in figure 4.13 contains the universal quantification of two variables, i and j, over values from 1 to n:

$$\forall\, i, j \in 1..n \bullet \mathsf{classId}^{\ell}(X_i) = \mathsf{classId}^{\ell}(X_j) \Longrightarrow i = j$$

This means the same as

$$\forall\, i \in 1..n \bullet \Big(\forall\, j \in 1..n \bullet \mathsf{classId}^{\ell}(X_i) = \mathsf{classId}^{\ell}(X_j) \Longrightarrow i = j\Big)$$

3.2.4 Operational semantics

The operational semantics of the j calculi are presented as follows. I define a rewrite judgement of the form:

$$P \vdash e, h, s \rightsquigarrow v, h'$$

On the left of the turnstile ($\vdash$) P is a program that provides the context for execution, and is used to for method lookup and, in the aspect-oriented calculus, for deciding what advice to run; On the left of the arrow ($\rightsquigarrow$) are the 'input' valuables for rewrite: e is the expression being reduced; h models the heap containing the dynamically allocated objects in use by the program and in which side-effects of execution are made; s models the stack frame containing bindings for variables such as `this` (the address of the currently executing object) and `x` (a formal parameter), etc.; on the right of the arrow are the 'output' variables for rewrite: v is the value to which e reduced; h' is a new heap which may be the same as h or different if there are side effects in the reduction of e.

For the cases where reduction of e does not result in a value, variable dv is used in place of v to stand for one of a set of predefined deviations: npe for errors resulting from dereferencing a null pointer and stk signifying that execution has got 'stuck' due to, for example, a type error such as an attempt to access a non-existent field in an object. In an ideal system, such error cases would only occur in programs that are deemed to by the program validation rules, to be not well-formed.

The circumstances under which a rewrite judgement, of the form described above, is deemed to hold true are defined by a set of inference rules. An inference rule has the following form:

$$[name]\ \frac{requirement}{conclusion}$$

The name is a short name to identify the rule, the requirement is a set of judgements and equations and the conclusion is a set of judgements. Variables in a inference rules are implicitly existentially qualified, and their scope is the whole rule. All of the conclusion judgements may be deemed to hold true if all the requirements can be shown to be true, which is to say, if it is possible to find a set of values for the variables such that the requirements hold. Since expressions may take many forms, there are rules with separate conclusion judgements for each syntactic form of expression. Variables are bound to subterms within expressions, for example the rule for rewriting a sequence of expressions has a conclusion of the form:

$$P \vdash \quad e;e' \quad , h, s \rightsquigarrow v, h'$$

where $e;e'$ represents execution of e followed by execution of e'. Application of a rule to a judgement with some variables bound effectively binds those variables in the rule, thus for a specific program P_1, expression e_1, heap h_1 and stack frame s_1, the rewrite judgement: $P_1 \vdash e_1, h_1, s_1 \rightsquigarrow v, h'$ means that e_1 rewrites to some value v in the context of program P_1, with heap h_1 and stack frame s_1 giving some new heap h'. This will hold if a rule matching the form of e_1 can be found and, when substituting the values P_1, e_1, h_1 and s_1 into the rule, values can be found for v and $h1$ (and any other variables appearing in the requirement of that rule) such that the requirements hold true.

For example, the rule for executing a sequence of expressions (defined in figure 4.35) is as follows:

$$[RWSEQ]\ \frac{P \vdash e, h, s \rightsquigarrow v', h'' \qquad P \vdash e', h'', s \rightsquigarrow v, h'}{P \vdash e;e', h, s \rightsquigarrow v, h'}$$

The sequential execution of e and e' is indicated by the way their side-effects on the heap are chained together: the requirement equations are not ordered: it is required merely that they all be true. However they are linked by virtue of the fact that they both refer to variable h'': the output variable from reduction of e and the input to reduction of e'. In order for both requirement judgements to be true, a value must be found for h'' that makes both these judgements true.

Chapter 4

A simple model OO language

4.1 The language j

j is a simple class-based imperative object-oriented language with methods and fields. Its expressions include method calls, field access and update. These features capture the essence of OOP and form a basis for models of AOP.

To keep j simple only one basic type (`bool`) is modelled and all methods are required to have exactly one formal parameter which is always called `x`. The all methods are functions and `void` type is included so methods that do not return anything of interest are declared in examples to have return type `Object`. Constructors and static members are not included in the model[1]. Method-local variables can be simulated simply using member fields as in figure 4.1, although this method is not reentrant and is, therefore, unsuitable for recursive methods. Re-entrant local variables, as well as additional formal parameters, can be simulated as illustrated in figure 4.2 by using, as the actual parameter, an object of a bespoke class with appropriate fields.

Static members are not vital for OOP [2]

⌜ The program in figure 4.1 includes three j classes that implement a very simple list comprising a head with data-bearing nodes.

The program in figure 4.2 illustrates use of an object of a bespoke class to provide local variables and additional formal parameters to a method call. It also shows use of an `init` method in place of a constructor to initialise a new instance. ⌟

$P_{list} =$

```
class ListRef extends Object {
  Node next

  Object setNext(Node x){
    this.next := x
  }
}
class List extends ListRef {
  Node localTempNode
  Object add(Object x){
    this.localTempNode = new Node
    this.localTempNode.setData(x)
    this.localTempNode.setNext(this.next)
    this.setNext(this.localTempNode)
  }

  Object drop(Object x){
    this.setNext(this.next.next)
  }

  bool isEmpty(Object x){
    this.next == null
  }
}
class Node extends ListRef {
  Object data
  Object setData(Object x){
    this.data := x
  }
}
```

Figure 4.1: List and list node in j

```
class Formal ext Object {
   Object localD
   Object formalX
}
class C ext Object{
   Object foo(Formal x){
      x.localD = new D
      x.localD.init(x.formalX)
   }
}
```

Figure 4.2: Use of a bespoke class to model additional locals and formals

$W \in Inter$	::=	`interface` C `ext` C^* `{` H^* `}`
$H \in Header$	::=	$T\ m$ `(` T' `)`
$D \in Declaration$	::=	$F \mid M$
$F \in Field$	::=	$T\ f$
$M \in Method$	::=	$T\ m$ `(` T `x` `)`{ e }
$e \in Exp$	::=	e `==` $e \mid e.f \mid e.f$ `:=` $e \mid e.m$`(`e`)` $\mid$ e `;` $e \mid$ `new` $C \mid$ `if` `(` e `)`{ e }`else`{ e } $\mid$ $sv \mid var$
$sv \in Special$	::=	`true` | `false` | `null`
$id, m, f \in Ident$		
$C \in ClassName$		
$T \in Type$	::=	$C \mid$ `bool`

Figure 4.3: Common syntax declarations for the j family of languages

$P \in Prog^{\mathsf{j}}$	::=	X^*
$X \in Class^{\mathsf{j}} \cup Inter$		
$Y \in Class^{\mathsf{j}}$	::=	`class` C `ext` C `[` `imp` C^* `]` `{` D^* `}`
$var \in Var^{\mathsf{j}}$	::=	`this` | `x`

Figure 4.4: The syntax of j

4.2 Syntax

Programs, as described in figures 4.3 and 4.4 comprise a sequence of classes and interfaces; a class has a name, a superclass, a list of implemented interfaces, a body comprising fields and method declarations. An interface has a name, a sequence of super-interfaces and a body comprising method headers (i.e. method signatures without body expressions). Fields comprise a type and a name. Methods comprise a return type, a name, a single typed argument and a body expression. For simplicity, all methods are functions returning the value of their body expression (rather than requiring an explicit return statement). One built-in value type `bool` is provided with values `true` and `false`. Other types are class types whose values are references to objects or the special value `null`. Expressions are special values (`true`, `false` and `null`), object creation via `new`, conditional expressions `if` ..., method calls and sequential composition.

Sets $ClassName$ and $Ident$ are distinct sets of identifiers representing, for example those beginning with upper case or lower case letters. `Object` is a distinguished member of the set $ClassName$.

[1]The precise interaction of advice with constructors in the presence of constructor chaining rules such as those of JAVA would be an interesting subject of future study.

[2]Static members are often used to encode singleton classes [31]; class variables (as found in SMALLTALK [32]) and once-functions (as found in EIFFEL [54]) are alternative ways of achieving similar ends. j itself lacks any such mechanisms but later extensions will include support for singleton aspects.

4.2.1 Functions on the syntax

Functions for projecting values from terms in the syntax are defined in figures 4.5 and 4.6.

For a class : $Y = \texttt{class}\ C\ \texttt{ext}\ C'\ \texttt{imp}\ C''_1..C''_k \{D_1..D_n\}$:

$$\begin{aligned}
\mathsf{classId}^{\mathsf{j}}(Y) &= C \\
\mathsf{superClass}^{\mathsf{j}}(Y) &= C' \\
\mathsf{DeclarationsX}^{\mathsf{j}}(Y) &= D_1..D_n \\
\mathsf{Interfaces}^{\mathsf{j}}(Y) &= C''_1..C''_k \\
\mathsf{Headers}^{\mathsf{j}}(Y) &= \epsilon
\end{aligned}$$

Figure 4.5: Functions that project components of class terms in language j

And for an interface: $W = \texttt{interface}\ C\ \texttt{ext}\ C'_1..C'_n\ \{H_1..H_k\}$:

$$\begin{aligned}
\mathsf{classId}^{\mathsf{j}}(W) &= C \\
\mathsf{superClass}^{\mathsf{j}}(W) &= Udf \\
\mathsf{DeclarationsX}^{\mathsf{j}}(W) &= \epsilon \\
\mathsf{Interfaces}^{\mathsf{j}}(W) &= C'_1..C'_n \\
\mathsf{Headers}^{\mathsf{j}}(W) &= H_1..H_n
\end{aligned}$$

Figure 4.6: Functions that project components of interface terms in language j

For a method: $M = T\ m(T'\ \mathsf{x})\{e\}$:

$$\begin{aligned}
\mathsf{body}(M) &= e \\
\mathsf{methodId}(M) &= m \\
\mathsf{return}(M) &= T \\
\mathsf{arg}(M) &= T'
\end{aligned}$$

Figure 4.7: Generic functions on the syntax of methods

For a function: $F = T \quad f$:

$$\begin{aligned}
\mathsf{fieldType}(F) &= T \\
\mathsf{fieldId}(F) &= f
\end{aligned}$$

Figure 4.8: Generic functions on the syntax of functions

For example, for the list program in figure 4.1, if:

$\mathsf{id}(X) = \mathsf{classId}^{\ell}(X)$
$\mathsf{id}(f) = \mathsf{fieldId}(f)$
$\mathsf{id}(m) = \mathsf{methodId}(m)$

Figure 4.9: Function id overloaded.

$X_{Node} =$ `class Node ext ListRef {` $F\ M$ `}`

where

$M =$ `Object setData(Object x){ this.data := x }`
$F =$ `Object data`

then:

$\mathsf{classId}^{\mathsf{j}}(X_{Node}) =$ `Node`
$\mathsf{superClass}^{\mathsf{j}}(X_{Node}) =$ `ListRef`
$\mathsf{DeclarationsX}^{\mathsf{j}}(X_{Node}) = F\ M$
$\mathsf{body}(M) =$ `this.data:=x`
$\mathsf{methodId}^{\mathsf{j}}(M) =$ `setData`
$\mathsf{arg}(M) =$ `Object`

⌟

Selecting from a sequence

Functions Fields' and $\mathsf{Methods}'$ separate functions and methods respectively from a sequence of declarations:

$\mathsf{Fields}' : Declaration^* \longrightarrow Field^*$
$\mathsf{Fields}'(\epsilon) = \epsilon$
$\mathsf{Fields}'(F :: D_1..D_n) = F :: \mathsf{Fields}'(D_1..D_n)$
$\mathsf{Fields}'(M :: D_1..D_n) = \mathsf{Fields}'(D_1..D_n)$

$\mathsf{Methods}' : Declaration^* \longrightarrow Method^*$
$\mathsf{Methods}'(\epsilon) = \epsilon$
$\mathsf{Methods}'(M :: D_1..D_n) = M :: \mathsf{Methods}'(D_1..D_n)$
$\mathsf{Methods}'(F :: D_1..D_n) = \mathsf{Methods}'(D_1..D_n)$

Figure 4.10: Generic functions to get features from a sequence

Domain restriction

Domain restriction is defined in figure 4.11 for sequences of declarations, method headers and classes (all of which have function defined that returns that returns an identifier representing the name of the declaration).

Let k be a set of identifiers, then

$$\epsilon/_k = \epsilon$$

$$D_1..D_n/_k = \begin{cases} D_1 :: (D_2..D_n/_k) & \text{if } id \in k \text{ and } id \text{ is the id of } D_1 \\ D_2..D_n/_k & \text{otherwise} \end{cases}$$

Figure 4.11: Domain restriction on sequences of sub-terms for which an id function is defined.

Declarations, fields and methods

In figure 4.12 GetDecs is defined, together with GetFields and GetMethods , to take a program P and a class name C, and give a sequence containing the declarations, fields or methods (respectively) belonging to class C in P; an empty sequence is given if there are none or if P contains no class named C.

$$\mathsf{GetDecs}^{\ell} : Prog^{\ell} \times ClassName \longrightarrow Declaration^*$$

$$\mathsf{GetDecs}^{\ell}(P, C) = \begin{cases} \mathsf{DeclarationsX}^{\ell}(X) & \text{if } P/_{\{C\}} = X \\ \epsilon & \text{otherwise} \end{cases}$$

$$\mathsf{GetFields}^{\ell} : Prog^{\ell} \times ClassName \longrightarrow Field^*$$

$$\mathsf{GetFields}^{\ell}(P, C) = \mathsf{Fields}'\Big(\mathsf{GetDecs}^{\ell}(P, C)\Big)$$

$$\mathsf{GetMethods}^{\ell} : Prog^{\ell} \times ClassName \longrightarrow Method^*$$

$$\mathsf{GetMethods}^{\ell}(P, C) = \mathsf{Methods}'\Big(\mathsf{GetDecs}^{\ell}(P, C)\Big)$$

Figure 4.12: Declarations, fields and methods parameterised over language $[\ell]$

Unique names and class lookup

Judgement $\vdash_{\mathsf{J}} P \overset{\triangle}{_{uc}}$ (defined in figure 4.13) states that the classes and interfaces in program P have unique names. If a program P has unique class names, then domain

$$[UC]\ \frac{\forall i, j \in 1..n \bullet \mathsf{classId}^{\ell}(X_i) = \mathsf{classId}^{\ell}(X_j) \Longrightarrow i = j}{\vdash_{\ell} X_1..X_n \overset{\triangle}{uc}}$$

Figure 4.13: Unique class names for language $[\ell]$

restriction $P/_{\{C\}}$ can be used to select classes, from within P, by name.

Our example program P_{list} has unique class names `ListRef`, `List` and `Node`, thus we have: $\vdash_{\mathsf{J}} P_{list} \overset{\triangle}{uc}$. and can look up classes by name, e.g. $P_{list}/_{\{\texttt{Node}\}} = X_{Node}$ and so on.

4.2.2 Types

Variables may be annotated with types that are the names of classes or interfaces, or the special type `bool`. The following judgements about types in j are defined in figure 4.14.

- $P \vdash_{\mathsf{J}} C \overset{\triangle}{ct}$ says that C is the name of a class in P.
- $P \vdash_{\mathsf{J}} C \overset{\triangle}{it}$ says that C is the name of an interface in P.
- $P \vdash_{\mathsf{J}} C \overset{\triangle}{ot}$ says that C is an object type, that is to say it is the name of either an interface or class in P.
- $P \vdash_{\mathsf{J}} T \overset{\triangle}{vt}$ says that T is suitable as the type of a variable: either an object type or the built-in type `bool`.

$$[classT]\ \frac{\vdash_{\ell} P \overset{\triangle}{uc} \quad P = X_1..X_k\ \texttt{class}\ C \ldots X_{k+2}..X_n}{P \vdash_{\ell} C \overset{\triangle}{ct}}$$
$$P \vdash_{\ell} C \overset{\triangle}{ot}$$

$$[intT]\ \frac{\vdash_{\ell} P \overset{\triangle}{uc} \quad P = X_1..X_k\ \texttt{interface}\ C \ldots\ X_{k+2}..X_n}{P \vdash_{\ell} C \overset{\triangle}{it}}$$
$$P \vdash_{\ell} C \overset{\triangle}{ot}$$

$$[objectT]\ \frac{}{P \vdash_{\ell} \texttt{Object} \overset{\triangle}{ot}} \qquad [varT]\ \frac{P \vdash_{\ell} C \overset{\triangle}{ot}}{P \vdash_{\ell} C \overset{\triangle}{vt}} \qquad [boolT]\ \frac{}{P \vdash_{\ell} \texttt{bool} \overset{\triangle}{vt}}$$

Figure 4.14: Class and interface judgements parameterised over language $[\ell]$

4.2.3 Class and type hierarchy

Classes specify their *superclass* after the `ext` keyword and their implemented interfaces after `imp`. Interfaces specify their super-interface after the `ext` keyword. These declarations describe a hierarchical relation among classes and interfaces. In order to check the well-formedness of a program's hierarchy first the hierarchy – a relation on class names, denoted Γ_P – is abstracted from the text of the program P, then validated with the rules in figures 4.17 and 4.18.

Subtype hierarchies

Judgement $P \vdash_{\mathsf{j}} C \leq C'$ (defined in figure 4.15) is the subtype relation among the classes of a given program. Reflexivity is defined for all types derived from classes or interfaces in P (rule [*reflexive*]) and for the special type `Object` (rule [*object*]). C is a subtype of C' if C' is the named superclass or super-interface of C (rule [*superclass*]), or if C' is a named super-interface of C (rule [*interface*]). The bottom type is introduced (rule [*bottom*]); this will be used as the type of `null`. Rule [*transitive*] gives the transitive closure of the class and interface relations defined in the program.

$$[\textit{object}]\ \frac{}{P \vdash_\ell \texttt{Object} \leq \texttt{Object}} \qquad [\textit{reflexive}]\ \frac{\vdash_\ell\ P \overset{\triangle}{uc}}{P \vdash_\ell\ C \overset{\triangle}{ot}}$$

$$P \vdash_\ell\ \bot \leq \bot \qquad \frac{}{P \vdash_\ell\ C \leq C}$$

$$[\textit{superclass}]\ \frac{\vdash_\ell\ P \overset{\triangle}{uc} \quad \mathsf{superClass}^\ell\big(P/_{\{C\}}\big) = C'}{P \vdash_\ell\ C \leq C'} \qquad [\textit{transitive}]\ \frac{P \vdash_\ell\ C'' \leq C' \quad P \vdash_\ell\ C \leq C''}{P \vdash_\ell\ C \leq C'}$$

$$[\textit{interface}]\ \frac{\vdash_\ell\ P \overset{\triangle}{uc} \quad \mathsf{Interfaces}^\ell\big(P/_{\{C\}}\big) = ...C'...}{P \vdash_\ell\ C \leq C'} \qquad [\textit{bottom}]\ \frac{P \vdash_\ell\ C \leq C}{P \vdash_\ell\ \bot \leq C}$$

Figure 4.15: The subtype relation

The subtype hierarchy Γ is a relation on class names: $\Gamma \subseteq (ClassName \times ClassName)$, it is an abstraction of the subtype hierarchy derived from a program. For a j program P with unique class names ($\vdash_{\mathsf{j}}\ P \overset{\triangle}{uc}$) the hierarchy Γ^{j}_P is defined in figure 4.16. For convenience, I will write $C \sqsubseteq C'$ in place of $(C, C') \in \Gamma$ when Γ is clear from the context.

Well formed hierarchies

A valid type hierarchy for a program P (with $\vdash_{\mathsf{j}}\ P \overset{\triangle}{uc}$) must have the following properties, which are formalised by judgements $\vdash\ \Gamma \overset{\triangle}{po}$ (figure 4.17) and $P \vdash_\ell\ \Gamma \diamond$ (figure 4.18):

$$\Gamma^{\ell}_{P} = \{(C, C') | P \vdash C \leq C'\}$$

Figure 4.16: Type hierarchy for a language $[\ell]$

- Γ must be a partial order. Reflexivity is ensured by the [*reflexive*] rule in figure 4.15. Antisymmetry the first requirement of the partial order rule in figure 4.17, and transitivity is the second requirement of that same rule.
- It is also required that `Object` is the top element, this is the third requirement of the rule in figure 4.17.
- Each class name in Γ is the name either of a class or an interface in P, this is the first requirement of the rule in figure 4.18.
- Interfaces have only interfaces as supertypes, this is the second requirement of the rule in figure 4.18.
- Every class has a unique immediate superclass, this is the third requirement of the rule in figure 4.18. Note that only class types are required to have unique superclasses and that, according to the rules in figure 4.14, `Object` is not a class type so this requirement does not break at the top of the hierarchy.

A hierarchy which satisfies both the requirements above is well formed for a j program: classes inherit singly from other classes and may implement many interfaces. Interfaces extend only other interfaces and `Object` is the top of the inheritance tree.

$$\frac{\begin{array}{l}\forall C, C' \bullet \begin{pmatrix} C \sqsubseteq C' \\ C' \sqsubseteq C \end{pmatrix} \Longrightarrow C = C' \\ \forall C, C', C'' \bullet \begin{pmatrix} C \sqsubseteq C'' \\ C'' \sqsubseteq C' \end{pmatrix} \Longrightarrow C \sqsubseteq C' \\ C \sqsubseteq \texttt{Object}\end{array}}{\vdash \Gamma \overset{\triangle}{_{po}}}$$

Figure 4.17: Partial order on type hierarchies

According to the rules for well formed hierarchies, the valid class types in the example program P_{list} defined in figure 4.1 are `Object`, `ListRef`, `List` and `Node`.

The hierarchy for our example program of figure 4.1 is $\Gamma^{\mathsf{j}}_{P_{list}}$ where: `Node` $\sqsubseteq$ `Node`, `Node` $\sqsubseteq$ `ListRef`, `Node` $\sqsubseteq$ `Object`, `List` $\sqsubseteq$ `List`, `List` $\sqsubseteq$ `ListRef`, `List` $\sqsubseteq$ `Object`, `ListRef` $\sqsubseteq$ `ListRef`, `ListRef` $\sqsubseteq$ `Object`, `Object` $\sqsubseteq$ `Object`, $\bot \sqsubseteq \bot$, $\bot \sqsubseteq$ `Node`, $\bot \sqsubseteq$ `List`, $\bot \sqsubseteq$ `ListRef` and $\bot \sqsubseteq$ `Object` as illustrated below.

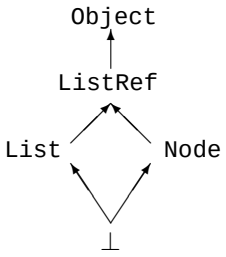
$$\forall C, C' \bullet C \sqsubseteq C' \implies \begin{pmatrix} P \vdash_\ell C \overset{\triangle}{_{ot}} \\ P \vdash_\ell C' \overset{\triangle}{_{ot}} \end{pmatrix}$$

$$\forall C, C' \bullet \begin{pmatrix} P \vdash_\ell C \overset{\triangle}{_{it}} \\ C \sqsubseteq C' \end{pmatrix} \implies P \vdash C' \overset{\triangle}{_{it}}$$

$$\forall C \bullet \left(P \vdash_\ell C \overset{\triangle}{_{ct}} \right) \implies \exists_1 C' \bullet \begin{pmatrix} C \neq C' \\ C \sqsubseteq C' \\ P \vdash_\ell C' \overset{\triangle}{_{ct}} \\ \forall C'' \bullet \begin{pmatrix} C \neq C'' \\ C \sqsubseteq C'' \\ P \vdash_\ell C'' \overset{\triangle}{_{ct}} \end{pmatrix} \implies C' \sqsubseteq C'' \end{pmatrix}$$

$$\overline{P \vdash_\ell \Gamma \diamond}$$

Figure 4.18: Well formed type hierarchy in language $[\ell]$



By inspection: this hierarchy is reflexive, transitive and antisymmetric. there are no interfaces; every class has a unique superclass; the top element is `Object`; therefore we have $\vdash \Gamma^{\mathsf{j}}_{P_{list}} \overset{\triangle}{_{po}}$ and $P \vdash_\urcorner \Gamma^{\mathsf{j}}_{P_{list}} \diamond$ ⌟

Unique immediate superclasses are required because of the way member lookup works: if a member is not found in a given class, its superclass is searched, and so on until either the member is found or `Object` is reached. If a class had more than one superclass it would be necessary to handle the case where a member with the required name is found in more than one immediate superclass. Such multiple inheritance is not permitted in JAVA and is not modelled in j either; it does not appear to contribute to our understanding of aspect oriented software.

Super

Function $\mathsf{super}^{\mathsf{j}}$, defined in figure 4.19, gives the immediate superclass of a named class C in a given program P if P has well a formed hierarchy (i.e. $\vdash \Gamma^{\mathsf{j}}_P \overset{\triangle}{_{po}}$ and $P \vdash_\urcorner \Gamma^{\mathsf{j}}_P \diamond$). Note that $\mathsf{super}^{\mathsf{j}}(P, C)$ is Udf if C is the name of an interface. Obtaining the superclass would normally involve examining the structure of the class definition and extracting the superclass. However, in anticipation of chapter 6 where parent introduction is introduced, the immediate superclass of C is defined to be the least element

$$\mathsf{super}^{\ell} : Prog^{\ell} \times ClassName \rightharpoonup ClassName$$

$$\mathsf{super}^{\ell}(P, C) = \begin{cases} Udf & \text{if } C = \texttt{Object} \\ Udf & \text{if } P \vdash_{\ell} C \overset{\triangle}{it} \\ C' & \text{if } \begin{pmatrix} C \neq C' \\ C \sqsubseteq C' \\ \forall C'' \bullet \begin{pmatrix} C \neq C'' \\ C \sqsubseteq C'' \\ P \vdash C'' \overset{\triangle}{ct} \end{pmatrix} \Longrightarrow C' \sqsubseteq C'' \end{pmatrix} \\ \texttt{Object} & \text{otherwise} \end{cases}$$

Figure 4.19: Superclass lookup for language $[\ell]$

of the superclasses of C which may be `Object`. If $C =$ `Object` or if C is an interface, then the immediate superclass is undefined. The default case is `Object` to allow for the addition, in chapter 7, of aspect classes whose superclass is implicitly `Object`.

Thus, for P_{list} defined in figure 4.1, we may write $\mathsf{super}^{\mathsf{j}}(P_{list}, \texttt{Node})$ which has the value `ListRef`. ⌟

All fields

In a program P with a well formed type hierarchy ($\vdash \Gamma_P \overset{\triangle}{po}$ and $P \vdash_{\mathsf{j}} \Gamma_P \Diamond$) function AllFields, defined in figure 4.20, gives a sequence of all the fields that should be present in an instance of a named class. This includes those defined in the class itself and also those that it inherits from its superclasses.

$$\mathsf{AllFields}^{\ell} : Prog^{\ell} \times ClassName \rightharpoonup Field^{*}$$

$$\mathsf{AllFields}^{\ell}(P, C) = \begin{cases} \epsilon & \text{if } C = \texttt{Object} \\ \mathsf{GetFields}^{\ell}(P, C) \mathbin{+\!\!+} \mathsf{AllFields}^{\ell}\big(P, \mathsf{super}^{\ell}(P, C)\big) & \text{if } \begin{pmatrix} P \vdash_{\mathsf{j}} C \overset{\triangle}{ct} \\ C \neq \texttt{Object} \end{pmatrix} \\ Udf & \text{otherwise} \end{cases}$$

Figure 4.20: All fields for objects of a given class in language $[\ell]$

In our example of figure 4.1 we have:

$$\mathsf{AllFields}^{\mathsf{j}}(P_{list}, \texttt{Node}) = \texttt{Object data Node next}$$

⌟

All interfaces

All interfaces is defined for the names of interfaces in a program. It satisfies the recursive decision below provided the program has a well formed hierarchy.

$$\mathsf{allInterfaces}^{\ell}(P, C) = C \cup \{\mathsf{allInterfaces}^{\ell}(P, C') | C' \in \mathsf{Interfaces}\big(P/_{\{C\}}\big)\}$$

Figure 4.21: All interfaces and super-interfaces of a named interface

4.2.4 Soundness

A program is structurally sound (judgement $\vdash_{\mathsf{j}} P \overset{\triangle}{SS}$) if its type hierarchy is well formed and its classes and interfaces have unique member names, which is to say no two members are allowed to have the same name, so a class may not posses a method and a field with the same name. This deviation from the behaviour of JAVA simplifies checking for sound programs.

The structural soundness judgement for j is defined by rule [*SS*] in figure 4.22).

$$\frac{\begin{array}{l}\vdash_{\ell} P \overset{\triangle}{uc} \\ P \vdash_{\ell} \Gamma^{\ell}_{P} \diamond \\ \vdash \Gamma^{\ell}_{P} \overset{\triangle}{po} \\ \forall C, id \bullet P/_{\{C\}} \neq \epsilon \Longrightarrow \begin{pmatrix} \#\left(\mathsf{GetDecs}^{\ell}(P, C)/_{\{id\}}\right) \leq 1 \\ \#\left(\mathsf{Headers}^{\ell}\big(P/_{\{C\}}\big)/_{\{id\}}\right) \leq 1 \end{pmatrix}\end{array}}{\vdash_{\ell} P \overset{\triangle}{SS}} \; [SS]$$

Figure 4.22: Structural soundness and unique method and field names, parameterised for language $[\ell]$

The three classes in P_{list} have unique names. The hierarchy $\Gamma^{\mathsf{j}}_{P_{list}}$ None of the three classes in P_{list} have more than one declarations with the same name. Since, as described earlier: $\vdash \Gamma^{\mathsf{j}}_{P_{list}} \overset{\triangle}{po}$ and $P \vdash_{\ell} \Gamma^{\mathsf{j}}_{P_{list}} \diamond$, we have $\vdash P_{list} \overset{\triangle}{SS}$.

Structurally sound programs admit member lookup (by class and member name) including finding the right declaration when members have been overridden.

4.2.5 Method and field lookup

The antisymmetric type hierarchy means that these lookup functions may the use class hierarchy to locate a class's inherited methods and fields as well as those it defines directly.

Field select $\mathsf{Fs}^{\mathsf{j}}(P, C, f)$ gives the field definition called f in class C of program P, or an empty sequence (ϵ) if no such field exists. Method select $\mathsf{Ms}^{\mathsf{j}}(P, C, m)$ gives the method definition called m in class C, header select $\mathsf{Hs}^{\mathsf{j}}(P, C, m)$ gives the method header declaration, called m, in interface C. These functions are defined in figure 4.23.

$$\mathsf{Fs}^{\ell}(P, C, f) = \mathsf{GetFields}^{\ell}(P, C)/_{\{f\}}$$
$$\mathsf{Ms}^{\ell}(P, C, m) = \mathsf{GetMethods}^{\ell}(P, C)/_{\{m\}}$$
$$\mathsf{Hs}^{\ell}(P, C, m) = \mathsf{Headers}^{\ell}\big(P/_{\{C\}}\big)/_{\{m\}}$$

Figure 4.23: Member selection for language $[\ell]$

Field lookup $\mathsf{F}^{\mathsf{j}}(P, C, f)$ gives the definition of field f either defined in or inherited by class C in program P, or Udf if C has no such field. Method lookup $\mathsf{M}^{\mathsf{j}}(P, C, m)$ gives the method declaration for method m defined in or inherited by class C of program P, or Udf if C has no such method. These functions are defined in figure 4.24 together with $\mathsf{H}^{\mathsf{j}}(P, C, m)$ which gives all headers called m defined in class or interface C, or one of its super-interfaces.

$$\mathsf{F}^{\ell}(P, C, f) = \begin{cases} Udf & \text{if } C = \texttt{Object} \\ F & \text{if } \mathsf{Fs}^{\ell}(P, C, f) = F \\ Udf & \text{if } \#\mathsf{Fs}^{\ell}(P, C, f) > 1 \\ \mathsf{F}^{\ell}\big(P, \mathsf{super}^{\ell}(P, C), f\big) & \text{if } \mathsf{Fs}^{\ell}(P, C, f) = \epsilon \end{cases}$$

$$\mathsf{M}^{\ell}(P, C, m) = \begin{cases} Udf & \text{if } C = \texttt{Object} \\ M & \text{if } \mathsf{Ms}^{\ell}(P, C, m) = M \\ Udf & \text{if } \#\mathsf{Ms}^{\ell}(P, C, m) > 1 \\ \mathsf{M}^{\ell}\big(P, \mathsf{super}^{\ell}(P, C), m\big) & \text{if } \mathsf{Ms}^{\ell}(P, C, m) = \epsilon \end{cases}$$

$$\mathsf{H}^{\ell}(P, C, m) = \begin{cases} Udf & \text{if } S = \emptyset \\ Udf & \text{if } \#S > 1 \\ H & \text{if } \mathsf{S}= \{\ H\ \} \end{cases}$$

where

$$S = \bigcup_{C' \in\ \mathsf{allInterfaces}^{\ell}(P, C)} \{\mathsf{H}^{\ell}(P, C', m)\}$$

Figure 4.24: Member lookup for language $[\ell]$

⌜ For example:

$$\mathsf{F}^{\ell}(P, \texttt{List}, \texttt{next}) = \texttt{Node next}$$

⌟

4.3 Well formedness in j

4.3.1 Type checking

Environment

Type checking is performed in the context of an environment γ that maps selected identifiers to their types. In j, identifiers are $Var^{\mathsf{j}} = \{\texttt{this}, \texttt{x}\}$:

$$\gamma : (\{\texttt{this}\} \longrightarrow ClassName) \cup (\{x\} \longrightarrow Type)$$

$$\gamma^{\ell}(var) = \begin{cases} T & \text{if } (var \mapsto T) \in \gamma^{\ell} \\ Udf & \text{otherwise} \end{cases}$$

$$\gamma^{\ell}[var' \mapsto T](var) = \begin{cases} T & \text{if } var = var' \\ \gamma^{\ell}(var) & \text{otherwise} \end{cases}$$

Figure 4.25: Type checking context for language $[\ell]$.

Such an environment is well formed $P \vdash \gamma \diamond$ if it maps the identifiers it contains to appropriate types. In j, `this` must always have a class type and x may have any variable type as defined in figure 4.26.

$$[\text{WFE}]\ \frac{P \vdash_{\ell} \gamma(x) \overset{\triangle}{vt} \quad P \vdash_{\ell} \gamma(\texttt{this}) \overset{\triangle}{ct}}{P \vdash_{\ell} \gamma \diamond}$$

Figure 4.26: Well formed typing environment for j

Type rules

The rules in figure 4.27 define type inference for expressions. The special values `true` and `false` have type `bool`. The types of identifiers `this` and `x` are looked up in the environment. Object creation `new` C returns the address of a new object of class C and therefore has type C. Identifier `null` has the bottom type which is a subtype of all class types. Field access expressions have the type declared for the field being accessed. Sequential composition of expressions returns the result of the second

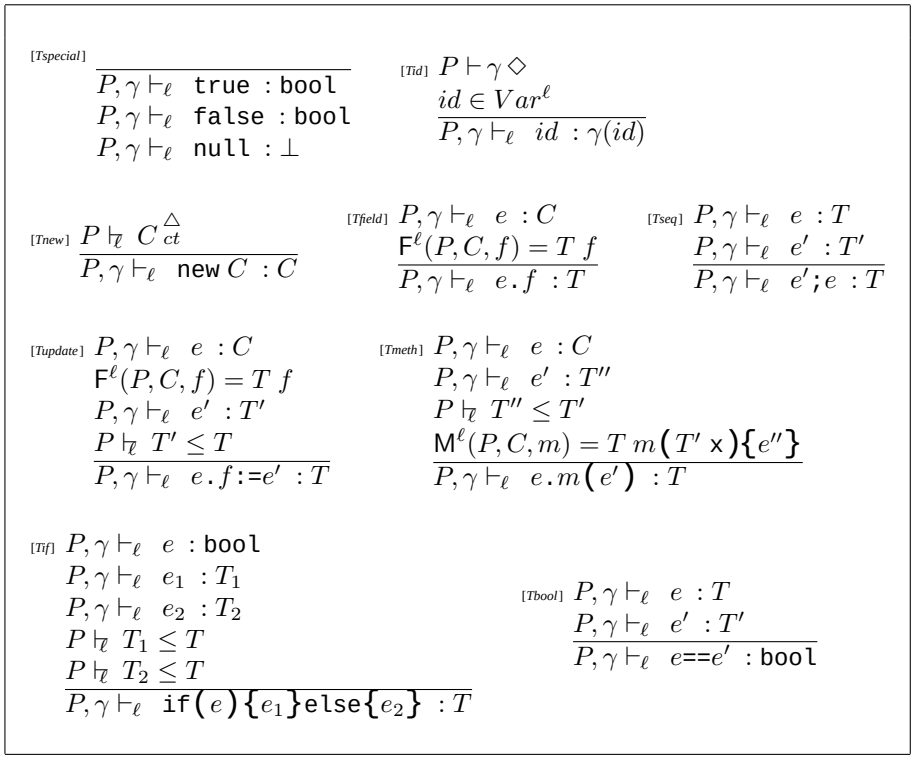


Figure 4.27: Type inference for expressions parameterised by language $[\ell]$

expression. Both sub-expressions must typecheck and the result of the second sub-expression is the result of the composite. Type checking for field update requires both sub-expressions to be well typed and for the RHS to have a type which is compatible with (i.e. a subtype of) the declared type of the field on the left. The type of the whole is the declared type of the field being updated. Method call expressions must have well typed sub-expressions for receiver and argument and the type of the argument sub-expression must be compatible with (i.e. a subtype of) the declared formal parameter type. The type of a method call is the declared return type of the method being invoked. Conditional statements must have conditions with type `bool` and there must exist a type that is a supertype of both the `if` and the `else` parts; this is the overall type of the expression.

4.3.2 Well-formed Programs

A structurally sound j program (P) is well formed (judgement $\vdash_{\mathsf{j}} P \diamond$) if all declarations contained in its classes and interfaces are well formed (rule [*WFP*] in figure 4.28). A method declaration (M) is well formed (judgement $P, C \vdash M \diamond$) if its argument and return types are valid variable types, its body expression typechecks in an appropriate environment with a type compatible with the method's declared return type; when a method overrides another, the signatures of both methods must be the same. (rule [*WFM*] in figure 4.28) A field declaration (F) is well formed (judgement

$P, C \vdash F \diamond$) if it has a valid type and if it does not hide an inherited field (rule [*WFF*] in figure 4.28). A method header is well formed (judgement $P, C \vdash H \diamond$) if its return type and argument type are valid variable types; when a header matches an inherited one, their signatures must be the identical (rule $\vdash P \diamond_h$ in figure 4.28). A class is well formed (judgement $P \vdash_{\mathsf{J}} C \diamond$) if it extends either another class in the same program (or `Object`), and implements only interfaces; all its declarations must be well formed and it must provide implementations for all method headers declared in all its super-interface, these must have matching signatures (rule [*WFC*] in figure 4.28). An interface is well formed (judgement $P \vdash_{\mathsf{J}} C \diamond$) if it extends either an interface or `Object` and all its headers are well formed (rule [*FWI*] in figure 4.28). A program is well formed (judgement $\vdash_{\mathsf{J}} P \diamond$) if it is structurally sound and each of its classes and interfaces are well formed (rule [*WFP*] in figure 4.28).

The example program P_{list} is a well-formed program: it is structurally sound and all of the declarations in each of its classes are well- formed in the context of their defining class. For example

$$P_{list}/_{\{\texttt{Node}\}} = X_{Node}$$
$$\mathsf{DeclarationsX}(X_{Node}) = D_{next}\ D_{setNext}$$
where
$$D_{next} = \texttt{Object next}$$
$$D_{setNext} = \texttt{Object setNext(Object x)\{this.next:=x\}}$$

For D_{next} we use rule [*WFF*]. We require: $P_{list} \vdash_{\mathsf{J}} \texttt{Object} \overset{\triangle}{vt}$, which we can obtain from the rule in figure 4.14, and $\mathsf{F^j}(P_{list}, \mathsf{super}(P_{list}, \texttt{Node}), \texttt{next}) = Udf$ which we obtain by first by evaluating the super on the left hand side to give $\mathsf{F^j}(P_{list}, \texttt{ListRef}, \texttt{next})$ which is Udf by definition of $\mathsf{F^j}$.

For example, for method $D_{setNext}$ we use rule [*WFM*]. We require:

$\vdash_{\mathsf{J}} P_{list} \overset{\triangle}{SS}$ which we have

$P_{list} \vdash_{\mathsf{J}} \texttt{Object} \overset{\triangle}{vt}$ (twice) which we get by type rules [*objectT*] and [*varT*] (figure 4.14)

$P \vdash_{\mathsf{J}} T'' \leq \texttt{Object}$

$P_{list}, \{\texttt{this} \mapsto \texttt{Node}, \texttt{x} \mapsto \texttt{Node}\} \vdash_{\mathsf{j}} \texttt{this.next:=x} : T''$

$$\begin{pmatrix} \mathsf{M^j}(P_{list}, \mathsf{super^j}(P_{list}, \texttt{Node}), \texttt{setNext}) = Udf \\ \text{or} \\ \mathsf{M^j}(P_{list}, \mathsf{super^j}(P_{list}, \texttt{Node}), \texttt{setNext}) = \texttt{Object setNext(Node x)\{}e'\texttt{\}} \end{pmatrix}$$

$$[\mathit{WFM}]\ \frac{\begin{array}{l}\vdash_\ell\ P\ \overset{\triangle}{SS}\\ P\vdash_\ell\ T\ \overset{\triangle}{vt}\\ P\vdash_\ell\ T'\ \overset{\triangle}{vt}\\ P\vdash_\ell\ T''\leq T\\ P,\{\mathtt{this}\mapsto C,\mathsf{x}\mapsto T'\}\vdash_{\mathsf{j}}\ e\ :T''\\ \mathsf{M}^\ell\big(P,\mathsf{super}^\ell(P,C),m\big)\neq \mathit{Udf}\\ \quad\Longrightarrow \mathsf{M}^\ell\big(P,\mathsf{super}^\ell(P,C),m\big)=T\ m\big(T'\ \mathsf{x}\big)\{e'\}\end{array}}{P,C\vdash_\ell\ T\ m\big(T'\ \mathsf{x}\big)\{e\}\diamond}$$

$$[\mathit{WFF}]\ \frac{P\vdash_\ell\ T\ \overset{\triangle}{vt}\qquad \mathsf{F}^\ell\big(P,\mathsf{super}^\ell(P,C),f\big)=\mathit{Udf}}{P,C\vdash_\ell\ T\ f\diamond}$$

$$[\mathit{WFH}]\ \frac{\vdash_\ell\ P\ \overset{\triangle}{SS}\qquad P\vdash_\ell\ T\ \overset{\triangle}{vt}\qquad P\vdash_\ell\ T'\ \overset{\triangle}{vt}}{P,C\vdash_\ell\ T\ m\big(T'\big)\diamond}$$

$$[\mathit{WFC}]\ \frac{\begin{array}{l}\vdash_\ell\ P\ \overset{\triangle}{uc}\\ P\vdash_\ell\ C\ \overset{\triangle}{ct}\\ P\vdash_\ell\ \mathsf{superClass}(C)\ \overset{\triangle}{ct}\\ \forall\, D\in\mathsf{GetDecs}^\ell(P,C)\bullet P,C\vdash_\ell\ D\diamond\\ \forall\, C'\in\mathsf{Interfaces}^\ell\big(P/_{\{C\}}\big)\bullet P\vdash_\ell\ C'\ \overset{\triangle}{it}\\ \forall\, m\bullet \mathsf{H}^\ell(P,C,m)=T\ m\big(T'\big)\Longrightarrow \mathsf{M}^\ell(P,C,m)=T\ m\big(T'\ \mathsf{x}\big)\{e\}\end{array}}{P\vdash_\ell\ C\diamond}$$

$$[\mathit{WFI}]\ \frac{\begin{array}{l}\vdash_\ell\ P\ \overset{\triangle}{uc}\\ P\vdash_\ell\ C\ \overset{\triangle}{it}\\ \forall\, m\bullet \mathsf{H}^\ell(P,C,m)=T\ m\big(T'\big)\Longrightarrow P,C\vdash_\ell\ T\ m\big(T'\big)\diamond\\ \forall\, C'\in\mathsf{Interfaces}^\ell\big(P/_{\{C\}}\big)\bullet P\vdash_\ell\ C'\ \overset{\triangle}{it}\end{array}}{P\vdash_\ell\ C\diamond}$$

$$[\mathit{WFP}]\ \frac{\vdash_\ell\ P\ \overset{\triangle}{SS}\qquad \forall\, m,C\bullet \#\{\mathsf{H}^\ell(P,C,m)\}\leq 1\qquad \forall\, C\bullet P\vdash_\ell\ C\ \overset{\triangle}{ot}\Longrightarrow P\vdash_\ell\ C\diamond}{\vdash_\ell\ P\diamond}$$

$$[\mathit{WFO}]\ \frac{}{P\vdash_\ell\ \mathtt{Object}\diamond}$$

Figure 4.28: Rules for well-formed programs in $[\ell]$

which we have, because

$\mathsf{super}^{\mathsf{j}}(P_{list}, \texttt{Node}) = \texttt{ListRef}$

$\mathsf{M}^{\mathsf{j}}(P, \texttt{ListRef}, \texttt{setNext}) = \texttt{Object setNext(Node x)\{}e'\texttt{\}}$

where

$e' = \texttt{this.next:=x}$

$P_{list}, \gamma \vdash_{\mathsf{j}} \texttt{this.next:=x} : T'$

where

$\gamma = \{\texttt{this} \mapsto \texttt{Node}, \texttt{x} \mapsto \texttt{Object}\}$

where $T' = \texttt{Node}$

since:

$\mathsf{F}^{\mathsf{j}}(P_{list}, \texttt{Node}, \texttt{next}) = \texttt{Node next}$

$P_{list}, \gamma \vdash_{\mathsf{j}} \texttt{this} : \texttt{Node}$

$P_{list}, \gamma \vdash_{\mathsf{j}} \texttt{x} : \texttt{Node}$

$P_{list} \vdash_{\mathsf{j}} \texttt{Node} \leq \texttt{Node}$

⌟

4.4 Operational semantics

4.4.1 Runtime values

The syntax for runtime values, addresses and objects, and deviations is defined in figure 4.29.

$\iota \in Addr$		
$v \in Val$	::=	$sv \mid \iota$
$dv \in Deviation$	::=	$npe \mid stk$
$o \in Object$	::=	$[\![\ (\ f{:}v \ )^* \ ]\!]^C$

Figure 4.29: Values and objects in j

ι ranges over addresses for objects, sv is defined in figure 4.3 to range over the special values `true`, `false` and `null`. dv ranges over the deviation result values: npe indicating an error arising from dereferencing a null pointer, stk indicating that the program execution has become stuck. Objects are a pair comprising a sequence of field-value pairs and the name of the object's dynamic type.

Value equality

Permitted values are addresses (ι) or special values (sv). Equality on values is defined to be the least relation that satisfies the following rule:

[VALEQ] $\dfrac{}{\iota == \iota}$

`true==true`

`false==false`

`null==null`

Figure 4.30: Equality rules for runtime values.

4.4.2 Objects

Objects (instances of j classes) are modelled by a pair comprising a list of field bindings and a class name that identifies the objects dynamic type. The syntax for objects is given in figure 4.29. Field selection and update operations are defined in figure 4.31. Update $o[f \mapsto v]$ replaces the current value of field f in object o with v. The result is undefined if o does not have a field called f; otherwise it is the value of f.

For an object $o = [\![f_1{:}v_1..f_n{:}v_n]\!]^C$ with unique field names we define:

$$o[f \mapsto v] = \begin{cases} [\![f_1{:}v_1..f_i{:}v,..f_n{:}v_n,]\!]^C & \text{if } f = f_i \text{ where } i \in 1..n \\ Udf & \text{otherwise} \end{cases}$$

$$o(f) = \begin{cases} v_i & \text{if } f = f_i \text{ where } i \in 1..n \\ Udf & \text{otherwise} \end{cases}$$

Figure 4.31: Object update and field selection

Object update changes field values but never changes the class to which an object belongs.

Conjecture 1 (Update Preserves Class). *If* $o = [\![\ldots]\!]^C$ *and* $o[f \mapsto v] \neq Udf$ *then* $o[f \mapsto v] = [\![\ldots]\!]^{C'}$ *and* $C = C'$

New objects are created from a class. The fields of the object are the defined and inherited fields of the class (given by function $\mathsf{AllFields}^{\mathsf{j}}$); they are initialised with default values for their types: `false` for `bool` and `null` for class types. Function new constructs an appropriately initialised object for a given class C in program P. Function new is defined for well formed programs ($\vdash_{\mathsf{J}} P \diamond$).

4.4.3 The store: heaps and stack frames

The store, which forms the context in which an expression executes and wherein side effects are recorded is defined in figure 4.33. The store comprises a heap and a stack

$$\mathsf{new}^{\ell}(P,C) = \begin{cases} [\![f_1{:}v_1 .. f_n{:}v_n]\!]^{\mathsf{C}} \\ \text{where} \\ \forall\, i \in 1..n \bullet \\ \quad \left(\begin{array}{l} T_i = \texttt{bool} \\ \quad \Longrightarrow v_i = \texttt{false} \\ P \vdash T_i \overset{\triangle}{_{ct}} \\ \quad \Longrightarrow v_i = \texttt{null} \end{array} \right) & \text{if } \left(\begin{array}{l} P/_{\{C\}} \neq Udf \\ \mathsf{AllFields}^{\ell}(P,C) \\ \quad = T_1\ f_1 .. T_n\ f_n \end{array} \right) \\ Udf & \text{otherwise} \end{cases}$$

Figure 4.32: Creation and initialisation of new objects for language $[\ell]$

frame. I use variable h to range over values in the set $Heap$ of heaps, and s to range over values in the set $Stack$ of stack frames:

$$h \in Heap = Addr \rightharpoonup Object$$
$$s \in Stack = \{\texttt{this}, \texttt{x}\} \rightharpoonup Val$$

The heap is a partial mapping from addresses to objects. The stack frame provides bindings for variables in the execution of the block and so comprises a partial map from identifiers to values (simple values or object ids). Stack frames can be extended and/or overridden with additional variable bindings using the right-preferential composition $\oplus$ operator: $s \oplus \{id \mapsto v\}$. Objects in the heap can be replaced with heap update $h[\iota \mapsto o]$.

$$\begin{array}{lcl} h(\iota) & = & \begin{cases} o & \text{if } (\iota \mapsto o) \in h \\ Udf & \text{otherwise} \end{cases} \\ h[\iota \mapsto o](\iota') & = & \begin{cases} o & \text{if } \iota = \iota' \\ h(\iota') & \text{otherwise} \end{cases} \\ \\ s(id) & = & \begin{cases} v & \text{if } (id \mapsto v) \in s \\ Udf & \text{otherwise} \end{cases} \\ (s \oplus s')(y) & = & \begin{cases} s'(y) & \text{if } s(y) \neq Udf \\ s(y) & \text{otherwise} \end{cases} \end{array}$$

Figure 4.33: Heaps and stacks

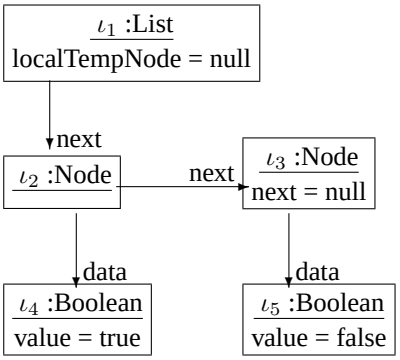
I write stack frames as mappings from variables to values. for example:

$$s = \{\texttt{this} \mapsto \iota, \texttt{x} \mapsto \texttt{true}\}$$

As an example of values and objects in the heap, we extend the example program P_{list} to include a class `Boolean` which wraps boolean values:

```
class Boolean extends Object {
  bool value
}
```

We can now construct the following configuration of objects representing a list of two Boolean values:



This configuration might be modelled in j with the following heap:

$$
\begin{aligned}
h = \{ \quad & \iota_1 \mapsto [\![\texttt{localTempNode:null}, \texttt{next:}\iota_2]\!]^{\texttt{List}}, \\
& \iota_2 \mapsto [\![\texttt{data:}\iota_4, \texttt{next:}\iota_3]\!]^{\texttt{Node}} \\
& \iota_3 \mapsto [\![\texttt{data:}\iota_5, \texttt{next:null}]\!]^{\texttt{Node}} \\
& \iota_4 \mapsto [\![\texttt{value:true}]\!]^{\texttt{Boolean}} \\
& \iota_5 \mapsto [\![\texttt{value:false}]\!]^{\texttt{Boolean}} \}
\end{aligned}
$$

4.4.4 Runtime types

Function type (defined in figure 4.34) gives the type of values in the context of a heap h. Special values `true` and `false` always have type `bool`, the address ι of an object in the heap has the class type which is that object's superscript. The null pointer has the special bottom type $\bot$. Addresses for which there is no object in the store have undefined type.

$$\mathsf{type}(h,v) = \begin{cases} \texttt{bool} & \text{if } v \in \{\texttt{true},\texttt{false}\} \\ C & \text{if } \begin{pmatrix} v = \iota \\ h(\iota) = [\![\ldots]\!]^{\mathsf{C}} \end{pmatrix} \\ \bot & \text{if } v = \texttt{null} \\ Udf & \text{otherwise} \end{cases}$$

Figure 4.34: type gives the type of a value

4.4.5 Reduction rules

Reduction rules for normal execution

The reduction rules rewrite a configuration comprising an expression e, a heap h and a stack frame s in the context of a program P. Such a configuration rewrites ($\leadsto$) to a value – the result of evaluating the expression – and a new heap which contains the results of any side-effects in the evaluation of the expression.

$$\leadsto : Prog^{\mathsf{j}} \longrightarrow Exp \times Heap \times Stack \longrightarrow ((Val \cup Deviation) \times Heap)$$

The stack frame is modelled separately from the model of the heap as opposed to modelling the two together in a single map as in [24]. Side effects to the heap are modelled by allowing a new heap along with the result of reduction, on the right hand side of the rewrite judgement. There is, however, no new stack frame which means that side effects to the stack frame are not possible. Using a single variable to model the stack and heap and propagating changes to the heap implicitly propagates changes to the stack frame too. This keeps the rewrite frame simple (with a single variable) but requires additional work to re-bind local variables after calls to methods (or advice bodies) which might change the value of values on the stack. For example, the rule for method reduction might look like this:

$$\text{[RWMETH]}\ \frac{\begin{array}{l} e_0, \sigma \underset{P}{\leadsto} \iota, \sigma_1 \\ e_1, \sigma_1 \underset{P}{\leadsto} v, \sigma_2 \\ \mathsf{type}(\sigma_2, \iota) = C_r \\ \mathsf{M}(P, C_r, m) = T\ m\big(\,T'\ \mathsf{x}\,\big)\{e\} \\ e, \sigma[\texttt{this} \mapsto \iota][\mathsf{x} \mapsto v'] \underset{P}{\leadsto} v, h' \end{array}}{e_0.m\big(\,e_1\,\big), \sigma \underset{P}{\leadsto} v, \sigma'[\texttt{this} \mapsto \sigma(\texttt{this})][\mathsf{x} \mapsto \sigma(\mathsf{x})]}$$

Rather than as in figure 4.35. Since stack frame changes are not propagated in j, it is not possible to model changes to the formal parameter x, recall, however, that formal parameters and local variables are more realistically modelled in j using fields of an actual parameter object created for this purpose. Changes to these would, of course, be modelled by changes to the heap. Separating the heap and stack frame in this way lead to significant simplifications in the rules relating to execution of advice bodies (which

have a number of read-only special variables defined in their stack frame in addition to `this` and `x`).

The rules are given in figure 4.35. Rule [*RWVAL*] defines an identity rewrite for atomic values. Rule [*RWGET*] defines how field accesses are reduced: the target object expression e is reduced to give an address ι to which the heap is applied ($h'(\iota)$, see figure 4.33) to give the owning object which in turn is applied to the field name ($h'(\iota)(f)$, see figure 4.31) to give the field value. Rule [*RWSET*] defines field update: the target object expression e is reduced to address ι; the new value expression e' is reduced to value v and the heap is updated with an updated version of the target object: $h'(\iota)$ gives the target object, $h'(\iota)[f \mapsto v]$ gives the updated target object and $h'[\iota \mapsto h'(\iota)[f \mapsto v]]$ updates the heap with the updated object. Rule [*RWNEW*] defines object creation: a new object (the result of function $\mathsf{new}^{\ell}(P, C)$) is included in the heap at new address ι. Rule [*RWID*] defines how variable identifiers (id) are looked up in the stack frame ($s(id)$). Rule [*RWSEQ*] defines sequential execution of expressions: the first expression e is reduced to value v' (which is discarded) and intermediate heap h'', then the second expression e' is reduced in the context of the intermediate heap, to give the result v. Rules [*RWEQ*] and [*RWNEQ*] define reduction of equality tests for equal values. The terms to be compared for equality are reduced and then the results compared using the rules for value equalty defined in figure 4.30. If the values are euqla, then the judgement in rule [*RWEQ*] holds, otherwise that in [*RWNEQ*] holds. Rule [*RWIFT*] and [*RWIFF*] defined reduction of `if... else` expressions: the condition expression e is reduced to give a value Boolean value v and either e_1 or e_2 is reduced depending on whether or not $v = \texttt{true}$. Rule [*RWMETH*] defines reduction of method invocation: the target object expression e is reduced to give address ι of the target object; actual parameter expression e' is reduced to value v', The method body e'' is looked up in the program text. This expression is reduced in the context of a new stack frame that binds the target object address to `this` and actual parameter value v' to formal parameter `x`.

For example, the expression `new List.isEmpty(null)` can be reduced in the context of P_{list} and an empty heap and stack as follows. We require:

$$P_{list} \vdash_{\mathsf{J}} \texttt{new List.isEmpty(null)}, h_0, s_0 \rightsquigarrow v, h \tag{4.1}$$

where $h_0 = \emptyset$ and $s_0 = \emptyset$.

By rule [*RWMETH*] with $e = \texttt{new List}$, $m = \texttt{isEmpty}$, $e_0 = \texttt{null}$, $h = \emptyset$ and $s = \emptyset$, we require:

$$P_{list} \vdash_{\mathsf{J}} \texttt{new List}, h_0, s_0 \rightsquigarrow \iota, h_1 \tag{4.2}$$

$$P_{list} \vdash_{\mathsf{J}} \texttt{null}, h_1, s_0 \rightsquigarrow v_1, h_2 \tag{4.3}$$

$$\mathsf{M}^{\mathsf{j}}(P_{list}, \mathsf{type}(h_2, \iota), \texttt{isEmpty}) = T\ m\texttt{(}T'\ \texttt{x}\texttt{)\{}e_1\texttt{\}} \tag{4.4}$$

$$P_{list} \vdash_{\mathsf{J}} e_1, h_2, s_1 \rightsquigarrow v, h' \tag{4.5}$$

where: $s_1 = \{\texttt{this} \mapsto \iota, \texttt{x} \mapsto v_1\}$.

For 4.2, using rule [*RWNEW*], we require: ι fresh in h_0 (trivially true since $h_0 = \emptyset$) and $\mathsf{new}(P_{list}, \texttt{List}) \neq Udf$.

By definition of new (figure 4.32): $\mathsf{new}(P_{list}, \texttt{List}) = o$

[RWVAL] $$\frac{}{P \vdash_\ell \ sv, h, s \ \rightsquigarrow \ sv, h}$$

[RWGET] $$\frac{P \vdash_\ell \ e, h, s \ \rightsquigarrow \ \iota, h' \qquad h'(\iota)(f) = v}{P \vdash_\ell \ e\,.\,f, h, s \ \rightsquigarrow \ v, h'}$$

[RWSET] $$\frac{P \vdash_\ell \ e, h, s \ \rightsquigarrow \ \iota, h'' \qquad P \vdash_\ell \ e', h'', s \ \rightsquigarrow \ v, h' \qquad h'(\iota)[f \mapsto v] \neq Udf}{P \vdash_\ell \ e\,.\,f := e', h, s \ \rightsquigarrow \ v, h'[\iota \mapsto h'(\iota)[f \mapsto v]]}$$

[RWNEW] ι is fresh in s
$$\frac{\mathsf{new}^\ell(P, C) \neq Udf}{P \vdash_\ell \ \mathsf{new}\ C, h, s \ \rightsquigarrow \ \iota, h[\iota \mapsto \mathsf{new}^\ell(P, C)]}$$

[RWID] $$\frac{s(id) \neq Udf}{P \vdash_\ell \ id, h, s \ \rightsquigarrow \ s(id), h}$$

[RWSEQ] $$\frac{P \vdash_\ell \ e, h, s \ \rightsquigarrow \ v', h'' \qquad P \vdash_\ell \ e', h'', s \ \rightsquigarrow \ v, h'}{P \vdash_\ell \ e; e', h, s \ \rightsquigarrow \ v, h'}$$

[RWEQ] $$\frac{P \vdash_\ell \ e, h, s \ \rightsquigarrow \ v', h'' \qquad P \vdash_\ell \ e', h'', s \ \rightsquigarrow \ v, h' \qquad v = v'}{P \vdash_\ell \ e == e', h, s \ \rightsquigarrow \ \mathtt{true}, h'}$$

[RWNEQ] $$\frac{P \vdash_\ell \ e, h, s \ \rightsquigarrow \ v', h'' \qquad P \vdash_\ell \ e', h'', s \ \rightsquigarrow \ v, h' \qquad v \neq v'}{P \vdash_\ell \ e == e', h, s \ \rightsquigarrow \ \mathtt{false}, h'}$$

[RWIFT] $$\frac{P \vdash_\ell \ e, h, s \ \rightsquigarrow \ \mathtt{true}, h'' \qquad P \vdash_\ell \ e_1, h'', s \ \rightsquigarrow \ v, h'}{P \vdash_\ell \ \mathtt{if}(e)\{e_1\}\mathtt{else}\{e_2\}, h, s \ \rightsquigarrow \ v, h'}$$

[RWIFF] $$\frac{P \vdash_\ell \ e, h, s \ \rightsquigarrow \ \mathtt{false}, h'' \qquad P \vdash_\ell \ e_2, h'', s \ \rightsquigarrow \ v, h'}{P \vdash_\ell \ \mathtt{if}(e)\{e_1\}\mathtt{else}\{e_2\}, h, s \ \rightsquigarrow \ v, h'}$$

[RWMETH] $$\frac{P \vdash_\ell \ e, h, s \ \rightsquigarrow \ \iota, h'' \qquad P \vdash_\ell \ e', h'', s \ \rightsquigarrow \ v', h''' \qquad \mathsf{M}^\ell(P, \mathsf{type}(h''', \iota), m) = T\ m(T'\ \mathtt{x})\{e''\} \qquad P \vdash_\ell \ e'', h''', \{\mathtt{this} \mapsto \iota, \mathtt{x} \mapsto v'\} \ \rightsquigarrow \ v, h'}{P \vdash_\ell \ e\,.\,m(e'), h, s \ \rightsquigarrow \ v, h'}$$

Figure 4.35: Reduction rules for normal execution in $[\ell]$

where $o = [\![\texttt{localTempNode:null}, \texttt{next:null}]\!]^{\texttt{List}}$. Which gives:

$$\begin{aligned} h_1 &= h_0[\iota \mapsto o] \\ &= \{\iota \mapsto o\} \end{aligned}$$

Substituting into 4.3, using rule [*RWVAL*], we have: $v_1 = \texttt{null}$ and $h_2 = h_1$.

Substituting into 4.4 we require: $\mathsf{M}^{\mathsf{j}}(P_{list}, \mathsf{type}(h_1, \iota), \texttt{isEmpty}) = T\, m(T'\ \texttt{x})\{e_1\}$. $\mathsf{type}(h_1, \iota) = \texttt{List}$. By definition of M (figure 4.24) we have: $T = \texttt{bool}$, $m = \texttt{isEmpty}$, $T' = \texttt{Object}$ and $e_1 = \texttt{this.next == null}$.

Substituting into 4.5, we require:

$$P_{list} \vdash_{\mathsf{j}} \texttt{this.next == null}, h_1, s_1 \rightsquigarrow v, h'$$

where: $s_1 = \{\texttt{this} \mapsto \iota, \texttt{x} \mapsto \texttt{null}\}$.

By rule [*RWEQ*] we require:

$$P_{list} \vdash_{\mathsf{j}} \texttt{this.next}, h_1, s_1 \rightsquigarrow v_2, h_3 \tag{4.6}$$

$$P_{list} \vdash_{\mathsf{j}} \texttt{null}, h_3, s_1 \rightsquigarrow v_3, h_4 \tag{4.7}$$

$$v_2 = v_3 \tag{4.8}$$

For 4.6, by rule [*RWGET*], we require:

$$P_{list} \vdash_{\mathsf{j}} \texttt{this}, h_1, s_1 \rightsquigarrow \iota', h_3 \tag{4.9}$$

$$h_3(\iota')(\texttt{next}) = v_2 \tag{4.10}$$

For 4.9, by rule [*RWID*] we have that: $\iota' = \iota$, since $s_1(\texttt{this}) = \iota$, and $h_3 = h_1$.

Substituting into 4.10, we require: $h_1(\iota)(\texttt{next}) = v_2$. $h_1(\iota) = o$, and $o(\texttt{next}) = \texttt{null}$ so $v_2 = \texttt{null}$ and $h_3 = h_1$.

Substituting into 4.7, by rule [*RWVAL*], we have: $h_4 = h_1$ and $v_3 = \texttt{null}$.

For 4.8 we have $\texttt{null} = \texttt{null}$ by value equality (figure 4.30), and therefore $v = \texttt{true}$ and $h' = h_1$. ⌟

Reduction rules for getting stuck

Execution will get stuck if the conditions for reduction cannot be met. These rules formalise the conditions under which execution gets stuck, using the special value stk to represent stuckness. Rules describing when execution gets stuck are defined in figure 4.36. stk is treated as a deviation in the rules for propagation of deviations in figure 4.38.

Creation of a new object of class C [*SKNEW*] gets stuck if the program does not contain a class definition for C. Access to a variable [*SKID*] will get stuck if the current stack frame does not contain a binding for that variable. Access or update to a field f [*SKGET*] will get stuck if the receiver object does not contain a field called f. Method call to method m [*SKMETH*] will get stuck if the receiver object's class does not define a method called m.

⌜ For example, the expression `new Widget` reduces in P_{list} to stk, preserving the heap, since there is no class `Widget` in P_{list} and so $\mathsf{new}^{\mathsf{j}}(P_{list}, \texttt{Widget}) = Udf$ ⌟

$$[\textit{SKNEW}]\ \frac{\mathsf{new}^{\ell}(P, C) = \mathit{Udf}}{P \vdash_{\ell}\ \mathsf{new}\ C, h, s \rightsquigarrow \mathit{stk}, h} \qquad [\textit{SKID}]\ \frac{s(id) = \mathit{Udf}}{P \vdash_{\ell}\ id, h, s \rightsquigarrow \mathit{stk}, h}$$

$$[\textit{SKGET}]\ \frac{P \vdash_{\ell}\ e, h, s \rightsquigarrow \iota, h' \quad h'(\iota)(f) = \mathit{Udf}}{P \vdash_{\ell}\ e\,.\,f, h, s \rightsquigarrow \mathit{stk}, h'} \qquad P \vdash_{\ell}\ e\,.\,f\,\texttt{:=}e', h, s \rightsquigarrow \mathit{stk}, h'$$

$$[\textit{SKMETH}]\ \frac{P \vdash_{\ell}\ e_0, h, s \rightsquigarrow \iota, h' \quad \mathsf{M}^{\ell}(P, \mathsf{type}(h_1, \iota), m) = \mathit{Udf}}{P \vdash_{\ell}\ e_0\,.\,m\texttt{(}\,e_1\,\texttt{)}, h, s \rightsquigarrow \mathit{stk}, h'}$$

Figure 4.36: Reduction rules for getting stuck

Reduction rules for errors arising from dereferencing null pointers

Execution cannot proceed if an attempt is made to dereference a null pointer, that is to say, to access a field or method of an object via a reference where the reference has the value `null`. This condition is formalised in the rule below; the special value npe is used to represent null pointer errors. Rules for dereferencing null pointers are given in figure 4.37. npe is treated as a deviation in the rules for propagating deviations in figure 4.38.

$$[\textit{NP}]\ \frac{P \vdash_{\ell}\ e, h, s \rightsquigarrow \texttt{null}, h'}{P \vdash_{\ell}\ e\,.\,f, h, s \rightsquigarrow \mathit{npe}, h'} \qquad P \vdash_{\ell}\ e\,.\,f\,\texttt{:=}e', h, s \rightsquigarrow \mathit{npe}, h' \qquad P \vdash_{\ell}\ e\,.\,m\texttt{(}\,e_1\,\texttt{)}, h, s \rightsquigarrow \mathit{npe}, h'$$

Figure 4.37: Reduction rules for errors arising from dereferencing null pointers

For example, attempting to access the `next` field on the null address (rather than the address of an object) rewrites to the null pointer error value.

$$P \vdash_{\mathsf{J}} \texttt{null.next}, h, s \rightsquigarrow \mathit{npe}, h$$

Reduction rules for propagating deviations

When a subexpression of a composite expression either gets stuck, or results in a null pointer error, the composite expression will also be stuck or in error by strictness. The rules in figure 4.38 formalise this property.

$$\text{[DVXASS]}\ \frac{P \vdash_\ell\ e, h, s \rightsquigarrow dv, h'}{\begin{array}{l} P \vdash_\ell\ x\texttt{:=}e, h, s \rightsquigarrow dv, h' \\ P \vdash_\ell\ e\texttt{;}e', h, s \rightsquigarrow dv, h' \\ P \vdash_\ell\ \texttt{if(}e\texttt{)\{}e_1\texttt{\}else\{}e_2\texttt{\}}, h, s \rightsquigarrow dv, h' \\ P \vdash_\ell\ e\texttt{.}f, h, s \rightsquigarrow dv, h' \\ P \vdash_\ell\ e\texttt{.}f\texttt{:=}e', h, s \rightsquigarrow dv, h' \\ P \vdash_\ell\ e\texttt{.}m\texttt{(}e_1\texttt{)}, h, s \rightsquigarrow dv, h' \\ P \vdash_\ell\ e\texttt{==}e', h, s \rightsquigarrow dv, h' \end{array}}$$

$$\text{[DVSEQ]}\ \frac{\begin{array}{l} P \vdash_\ell\ e, h, s \rightsquigarrow v', h'' \\ P \vdash_\mathrm{j}\ e', h'', s \rightsquigarrow dv, h' \end{array}}{\begin{array}{l} P \vdash_\ell\ e\texttt{;}e', h, s \rightsquigarrow dv, h' \\ P \vdash_\ell\ e\texttt{==}e', h, s \rightsquigarrow dv, h' \end{array}}$$

$$\text{[DVIFT]}\ \frac{\begin{array}{l} P \vdash_\ell\ e, h, s \rightsquigarrow \texttt{true}, h_1 \\ P \vdash_\ell\ e_1, h_1, s \rightsquigarrow dv, h' \end{array}}{P \vdash_\ell\ \texttt{if(}e\texttt{)\{}e_1\texttt{\}else\{}e_2\texttt{\}}, h, s \rightsquigarrow dv, h'}$$

$$\text{[DVIFF]}\ \frac{\begin{array}{l} P \vdash_\ell\ e, h, s \rightsquigarrow \texttt{false}, h_1 \\ P \vdash_\ell\ e_2, h_1, s \rightsquigarrow dv, h' \end{array}}{P \vdash_\ell\ \texttt{if(}e\texttt{)\{}e_1\texttt{\}else\{}e_2\texttt{\}}, h, s \rightsquigarrow dv, h'}$$

$$\text{[DVSET]}\ \frac{\begin{array}{l} P \vdash_\ell\ e, h, s \rightsquigarrow \iota, h_0 \\ P \vdash_\ell\ e', h_0, s \rightsquigarrow dv, h' \end{array}}{P \vdash_\ell\ e\texttt{.}f\texttt{:=}e', h, s \rightsquigarrow dv, h'} \qquad \text{[DVMETH2]}\ \frac{\begin{array}{l} P \vdash_\ell\ e_0, h, s \rightsquigarrow \iota, h_1 \\ P \vdash_\ell\ e_1, h_1, s \rightsquigarrow dv, h' \end{array}}{P \vdash_\ell\ e_0\texttt{.}m\texttt{(}e_1\texttt{)}, h, s \rightsquigarrow dv, h'}$$

$$\text{[DVMETH3]}\ \frac{\begin{array}{l} P \vdash_\ell\ e_0, h, s \rightsquigarrow \iota, h_1 \\ P \vdash e_1, h_1, s \rightsquigarrow v, h_2 \\ \mathsf{M}^\ell(P, \mathsf{type}(s_2, \iota), m) = T\ m\texttt{(}T'\ \mathsf{x}\texttt{)\{}e\texttt{\}} \\ P \vdash_\ell\ e, h_2, \{\texttt{this} \mapsto \iota, x \mapsto v'\} \rightsquigarrow dv, h' \end{array}}{P \vdash_\ell\ e_0\texttt{.}m\texttt{(}e_1\texttt{)}, h, s \rightsquigarrow dv, h'}$$

Figure 4.38: Rules for propagation of deviations in $[\ell]$

4.5 Observations

4.5.1 The type hierarchy

The requirements on well-formed hierarchy include the single inheritance condition (all classes have a unique immediate superclass). This is guaranteed in well formed j programs by the following facts:

- the hierarchy is derived from the header of the class and interface declarations.
- well formed programs are allowed to list only other classes as their superclasses

In pj, presented in chapter 6, the hierarchy is inferred from a set of parent introductions as well as from the header of the classes and interfaces. This means the subtype relations are less structurally constrained and hence the requirement that every class has a unique superclass.

4.6 Concept List

Term	Meaning	Location
$\mathsf{GetDecs}^{\mathsf{j}}(P, C)$	All member declarations of class C in program P	figure 4.12
$\mathsf{GetFields}^{\mathsf{j}}(P, C)$	All fields defined in class C in program P	figure 4.12
$\mathsf{GetMethods}^{\mathsf{j}}(P, C)$	All methods defined in class C in program P	figure 4.12
$\vdash_{\mathsf{J}} P \overset{\triangle}{uc}$	Unique class and interface names	figure 4.13
$P \vdash_{\mathsf{J}} C \overset{\triangle}{ct}$	Class type	figure 4.14
$P \vdash_{\mathsf{J}} C \overset{\triangle}{it}$	Interface type	figure 4.14
$P \vdash_{\mathsf{J}} C \overset{\triangle}{ot}$	Valid object type (class or interface name)	figure 4.14
$P \vdash_{\mathsf{J}} T \overset{\triangle}{vt}$	Type suitable for variable declaration	figure 4.14
$P \vdash_{\mathsf{J}} C \leq C'$	Subtype (by declaration)	figure 4.15
Γ^{j}_P	The type hierarchy for program P	figure 4.16
$\vdash \Gamma \overset{\triangle}{po}$	Well formed type hierarchy (top, reflexive, antisymmetric)	figure 4.17
$P \vdash_{\mathsf{J}} \Gamma \diamond$	Valid hierarchy for program (covers all types, reflexive single class inheritance and separate interface hierarchy)	figure 4.18
$\mathsf{super}^{\mathsf{j}}(P, C)$	Find unique superclass of C in P	figure 4.19
$\mathsf{AllFields}^{\mathsf{j}}(P, C)$	Gives sequence of all fields supported by objects of class C	figure 4.20
$\vdash_{\mathsf{J}} P \overset{\triangle}{SS}$	Structurally sound: unique member names and well formed hierarchy.	figure 4.22
$\mathsf{Fs}^{\mathsf{j}}(P, C, f)$	Select field f of class C in P	figure 4.23
$\mathsf{Ms}^{\mathsf{j}}(P, C, m)$	Select field f of class C in P	figure 4.23
$\mathsf{F}^{\mathsf{j}}(P, C, f)$	Defined or inherited field f in C	figure 4.24
$\mathsf{M}^{\mathsf{j}}(P, C, m)$	Defined or inherited method m in C	figure 4.24
$\mathsf{H}^{\mathsf{j}}(P, C, m)$	Defined or inherited header m in C	figure 4.24
γ	Typing environments	figure 4.25
$P \vdash \gamma \diamond$	Well formed type contexts	figure 4.26
$P, \gamma \vdash \ e\ : T$	Expression e has type T	figure 4.27
$P, C \vdash_{\mathsf{J}} D \diamond$	Well formed member declarations (including overloading/shadowing)	figure 4.28
$P, C \vdash T\ m(\ T'\ ) \diamond$	Well formed method headers	figure 4.28
$P \vdash_{\mathsf{J}} X \diamond$	Well formed classes or interfaces	figure 4.28
$\vdash_{\mathsf{J}} P \diamond$	well formed programs	figure 4.28
h	Heaps	figure 4.33
s	Stack frames	figure 4.33
$[\![f{:}v \ldots]\!]^C$	Objects	figure 4.31
$\mathsf{new}^{\mathsf{j}}(P, C)$	New object of class C	figure 4.32
$\mathsf{type}(h, v)$	Dynamic type value v	figure 4.34
$P \vdash_{\mathsf{J}} e, h, s \rightsquigarrow v, h'$	Expression e rewrites in heap h and stack s to value v and modified heap h'	figure 4.35

Chapter 5

A language with member introduction

5.1 The language ij

ij is an extension of j that supports member *introduction*, which is to say, it allows fields and methods to be defined in classes other than those to which they belong. In ASPECTJ, introduction is only allowed in aspects[1]. ij does not include aspects and its extension to include introductions does not warrant the addition of aspects; thus, in ij, any class is allowed to contain introductions.

ij programs can be mapped onto j programs by a process, known as *weaving*, which resolves introductions.

5.2 Syntax

The syntax of ij is a small adaptation of the syntax of j (given in figure 4.3); the differences are shown in figures 5.1. Classes are extended to include introductions of the form $C \leftarrow D$ wherein a declaration D is introduced into a class called C.

According to this syntax we may write, the following example.

Examples of ij syntax

The following two classes:

[1] Special classes that contain aspect-oriented extensions. Aspect classes are introduced in chapter 7

$$
\begin{array}{rcl}
P \in Prog^{\text{ij}} & ::= & X^* \\
Y \in Class^{\text{ij}} & ::= & \texttt{class}\ C\ \texttt{ext}\ C\ [\ \texttt{imp}\ C^*\ ]\ \{\ D^*\ I^*\ \} \\
X \in Class^{\text{ij}} \cup Inter & & \\
I \in Introduction^{\text{ij}} & ::= & C \leftarrow D
\end{array}
$$

Figure 5.1: Syntax of ij; $Class^{\text{j}}$ is redefined, $Introduction$ is added; all other productions remain unchanged.

```
class A ext Object {}
class B ext Object { A <-- bool flag }
```

would be woven into:

```
class A ext Object { bool flag }
class B ext Object { }
```

Since there are no restrictions on which classes may introduce into which other classes a class may introduce members into itself:

```
class A ext Object { A <-- bool flag }
```

This would weave into:

```
class A ext Object { bool flag }
```

Or, for example, a class may introduce members into its own parent:

```
class A ext Object {}
class B ext A { A <-- bool flag }
```

This would be woven into:

```
class A ext Object { bool flag }
class B ext A { }
```

This suggests that programming with introductions is a more general form of programming and includes normal object-oriented programming as a special case where classes are only allowed to introduce declarations into themselves. ⌟

Below is another example, based on the subject-observer pattern.

⌜ **The subject-observer example**

Figure 5.2 shows an encoding of the subject-observer pattern ([31]) using this syntax. In this example the subject class `Subject` represents whatever model the programmer wishes to have observed by the observer class (`Observer`), it is not intended or required to be used as a superclass, in order to 'mix-in' the required subject behaviour.

To this end `Subject` does not contain code for the subject's behaviour, it contains a single member: the field `flag`, which represents the mutable state of the model. `Observer` contains the code for subject behaviour: introduced field `obs` and introduced methods `changed` and `setObserver`. `Observer` also contains the code for observer behaviour: method `update`.

```
class Subject ext Object {
  bool flag
}

class Observer ext Object {

  void update(Subject x){
    ... x.flag ...
  }

  Subject <-- Observer obs

  Subject <-- void changed(Object x){
                obs.update(this)
              }

  Subject <-- void setObserver(Observer x){
                this.obs = x;
              }

}
```

Figure 5.2: The subject-observer protocol built with introduction in ij

5.2.1 Functions on the syntax

Figure 5.3 shows the definition of the following functions on the syntax of ij: $\mathsf{classId}^{\mathsf{ij}}(X)$, $\mathsf{superClass}^{\mathsf{ij}}(X)$, $\mathsf{DeclarationsX}^{\mathsf{ij}}(X)$, $\mathsf{Interfaces}^{\mathsf{ij}}(X)$, $\mathsf{Headers}^{\mathsf{ij}}(X)$. In addition $\mathsf{Introductions}(X)$ is defined, which selects the introductions defined in a class or interface X.

Figure 5.5 defines $\mathsf{Intros}(P)$, which gives a sequence containing all introductions from all classes in program P.

Domain restriction is defined on introductions (in figure 5.6) using the receiving class as the identifying value.

Function $\mathsf{DeclarationsI}$ is defined in figure 5.7 for sequences of member introductions to give a sequence containing just the member declarations, ignoring the target classes. A sequence of introductions $(I_1..I_n)$ can be restricted, using domain restriction $I_1..I_n/_{\{C\}}$ (defined in figure 5.6), to just those that introduce members into a named

For a class : $Y = \texttt{class}\ C\ \texttt{ext}\ C'\ \texttt{imp}\ C''_1..C''_k\{D_1..D_n\ I_1..I_p\}$:

$$\begin{aligned}
\mathsf{classId}^{\ddot{\mathsf{ij}}}(Y) &= C \\
\mathsf{superClass}^{\ddot{\mathsf{ij}}}(Y) &= C' \\
\mathsf{DeclarationsX}^{\ddot{\mathsf{ij}}}(Y) &= D_1..D_n \\
\mathsf{Interfaces}^{\ddot{\mathsf{ij}}}(Y) &= C''_1..C''_k \\
\mathsf{Headers}^{\ddot{\mathsf{ij}}}(Y) &= \epsilon \\
\mathsf{Introductions}^{\ddot{\mathsf{ij}}}(Y) &= I_1..I_p
\end{aligned}$$

Figure 5.3: Functions that project components of class terms in language ij

For an interface:

$$\mathsf{Introductions}^{\ell}(W) = \epsilon$$

Figure 5.4: Generic function to extract the empty sequence of introductions from an interface in a language $[\ell]$.

$$\begin{aligned}
&\mathsf{Intros} : Prog^{\ell} \longrightarrow Introduction^{*} \\
&\mathsf{Intros}^{\ell}(\epsilon) = \epsilon \\
&\mathsf{Intros}^{\ell}(X :: P) = \mathsf{Introductions}^{\ell}(X) +\!\!+ \mathsf{Intros}^{\ell}(P)
\end{aligned}$$

Figure 5.5: Functions specific to languages $[\ell]$ with introduction

Let K be a set of class names ($K \in \wp(ClassName)$):

$$\begin{aligned}
(\epsilon)\,/_K &= \epsilon \\
(I :: I_1..I_n)\,/_K &= \begin{cases} I :: (I_1..I_n)\,/_K & \text{if } \begin{pmatrix} I = C \leftarrow D \\ C \in K \end{pmatrix} \\ (I_1..I_n)\,/_K & \text{otherwise} \end{cases}
\end{aligned}$$

Figure 5.6: Domain restriction for sequences of introductions

class (C), $\mathsf{GetDecs}^{\ddot{\mathsf{ij}}}(P, C)$ is defined, in figure 5.7; it gives all declarations for a named class: those defined within the class and those introduced to it from any other class in the program.

Functions $\mathsf{GetFields}^{\ddot{\mathsf{ij}}}(P, C)$ and $\mathsf{GetMethods}^{\ddot{\mathsf{ij}}}(P, C)$ (defined in figure 4.12), use $\mathsf{GetDecs}^{\ddot{\mathsf{ij}}}(P, C)$ to give the fields/methods belonging to the class named C in program P (whether defined within that class or introduced into it).

$$\mathsf{DeclarationsI} : Introduction^* \longrightarrow Declaration^*$$
$$\mathsf{DeclarationsI}(\epsilon) = \epsilon$$
$$\mathsf{DeclarationsI}(C \leftarrow D :: I_2..I_n) = D :: \mathsf{DeclarationsI}(I_2..I_n)$$

$$\mathsf{GetDecs}^{\ell} : Prog^{\mathsf{ij}} \times ClassName \longrightarrow Declaration^*$$
$$\mathsf{GetDecs}^{\ell}(P, C) =$$
$$\mathsf{DeclarationsX}\big(P/_{\{C\}}\big) +\!\!+ \mathsf{DeclarationsI}\big(\mathsf{Intros}(P)/_{\{C\}}\big)$$

Figure 5.7: DeclarationsI and GetDecs redefined for ij.

Unique names and class lookup

Judgment $\vdash_{\mathsf{ij}} P \overset{\triangle}{uc}$ (defined in figure 4.13) states that the classes and interfaces in program P have unique names. For a program with unique class names, restriction $P/_{\{C\}}$ gives the class named C in P (or ϵ if there is no such class).

5.2.2 Types

Types in ij work just the same way as in j. Judgments $P \vdash_{\mathsf{ij}} C \overset{\triangle}{ct}$, $P \vdash_{\mathsf{ij}} C \overset{\triangle}{it}$, $P \vdash_{\mathsf{ij}} C \overset{\triangle}{ot}$ and $P \vdash_{\mathsf{ij}} T \overset{\triangle}{vt}$ say that the given identifier is a class, interface, object or variable type, respectively. These Judgments are defined in figure 4.14.

5.2.3 Type hierarchy

The type hierarchy in ij is the same as in j: types are either class names or the special type `bool`. The declared subtype relation $P \vdash_{\mathsf{ij}} C \leq C'$ is defined in figure 4.15.

The subtype hierarchy is a relation Γ on class names: $\Gamma \subseteq (ClassName \times ClassName)$. For a program P with unique class names ($\vdash_{\mathsf{ij}} P \overset{\triangle}{uc}$) the hierarchy Γ^{ij}_P is defined in figure 4.16. A well-formed hierarchy is one that satisfies the conditions of well formedness ($P \vdash_{\mathsf{ij}} \Gamma \diamond$) laid out in figure 4.18.

5.2.4 Super

Function $\mathsf{super}^{\mathsf{ij}}$, is defined the same way as for ij (figure 4.19). It gives the immediate superclass of a named class C in program P provided P has well a formed hierarchy; $\mathsf{super}^{\mathsf{ij}}(P, C)$ is Udf if C is the name of an interface.

5.2.5 All fields

Function AllFields^{ij} (defined in figure 4.20) projects a sequence of all the fields that pertain to a named class. This includes those defined in the class itself and also those that it inherits from its superclasses.

5.3 Weaving

An ij program can be converted into a j program by a transformation that resolves all introductions into the classes to which they belong. This transformation is called weaving.

Weaving in the subject-observer example

Figure 5.2 shows a two class program fragment that encodes a version of the subject observer protocol [31] in ij using introduction. Class `Subject` defines only the `bool` field `flag`. This represents some state that an instance of class `Observer` is interested in observing. Class `Observer` defines the method `update(...)` and three introductions that add members to class `Subject`:

- method `update(Subject x)` takes a `Subject` object as its argument `x`, and accesses its `flag` field, in some way, in order to update the state of the observer in response to a change in the subject.
- introduction `Subject <-- Observer obs` introduces a field into class `Subject` to store a reference to the observer object which is interested in its state. Most implementations of the protocol allow multiple observers to register their interest in a subject, but here we allow one for simplicity since we have not shown how to model data structures in ij.
- introduction `Subject <-- void changed(...)...` adds method `changed(...)`; this method is intended to be called whenever a `Subject` object changes its state. It calls `update(...)` on the registered observer. If there is no registered observer this will result in a null pointer exception that could easily be avoided with an `if` statement to check for a null observer, but which are omitted for simplicity.
- introduction `Subject <-- void setObserver(...)...` adds the method by which observers are registered with a subject. An observer object can register itself as interested in the state of a subject `s` by calling `s.setObserver(this)`.

Figure 5.8 shows the j program obtained by weaving the program in figure 5.2.

Separation of concerns in the subject-observer example

Even this simple example demonstrates how introduction can improve separation of concerns. As written, the `Subject` class knows nothing about being part of the subject-observer protocol. It contains nothing more than its own model-related members represented by the `bool flag`) field. All the code pertaining to observers' ability to register

themselves with subjects and receiver updates is encapsulated in class `Observer`. A limitation of this implementation of the protocol is that there is no link between changes to the state of `Subject`'s `flag` field and calls to its introduced `update(...)` method. This link can be elegantly implemented using simple advice which is described through aj in Chapter 7. Meanwhile we must assume that clients of `Observer` know that after changing the state of some subject `s` they must call `s.update(null)`[2]

```
class Subject ext Object {

  bool flag

  Observer obs

  void changed(Object x){
    obs.update(this)
  }

  void setObserver(Observer x){
    this.obs = x;
  }

}

class Observer ext Object {

  void update(Subject x){
    ... x.flag ...
  }

}
```

Figure 5.8: The woven subject-observer example in j

There are two choices for how to drive the weaving process:

- visit each introduction and resolve it into its target class
- visit each class and resolve into it the introductions (if any) of which it is the target

The former would require a decision about what to do with with introductions whose target class cannot be found in the program. The latter simply ignores these introductions. The latter is chosen because this more accurately reflects the use of ASPECTJ, in which only introductions whose target can be found are processed. Another benefit of this choice is that every class is visited and has its introductions removed, so that all introductions in the program are removed, and those with target classes are resolved,

[2]The formal parameter `x` and actual parameter `null` are required only in order to obey the simple syntax from figure 4.3

the result is a j program. A consequence of tolerating introductions into non-extant classes is that it allows one or more classes containing introductions to be treated as a re-usable module and such a module could be appended to many programs before weaving, whether or not those classes contain target classes for all the introductions. An aspect-oriented compiler should issue a warning when it processes an introduction with no target. Since code does not get written by accident it almost certainly indicates the programmers intention to create an introduction.

Weaving a class $X = \texttt{class}\ C\ \texttt{ext}\ C'\ \texttt{imp}\ C_1..C_p\{D_1..D_nI_1..I_r\}$ is performed in the context of the program P in which X is defined. Member introductions $I_1..I_r$ are discarded and declarations from introductions that have C as their target are introduced. weaveC is defined in figure 5.9. The second function weaveP in figure 5.9 weaves

$$\mathsf{weaveC} : Prog^{\mathsf{ij}} \times Class^{\mathsf{ij}} \longrightarrow Class^{\mathsf{j}}$$
$$\mathsf{weaveC}(P, \texttt{class}\ C\ \texttt{ext}\ C'\ \texttt{imp}\ C_1..C_p\{D_1..D_nI_1..I_n\}) = \texttt{class}\ C\ \texttt{ext}\ C'\ \texttt{imp}\ C_1..C_p\{D'_1..D'_k\}$$
$$\text{where} \quad D'_1..D'_k = \mathsf{GetDecs}^{\mathsf{ij}}(P, C)$$

$$\begin{array}{lcl} \mathsf{weaveP} & : & Prog^{\mathsf{ij}} \times Prog^{\mathsf{ij}} \longrightarrow Prog^{\mathsf{j}} \\ \mathsf{weaveP}(P, \epsilon) & = & \epsilon \\ \mathsf{weaveP}(P, X :: P') & = & \mathsf{weaveC}(P, X) +\!\!+ \mathsf{weaveP}(P, P') \end{array}$$

$$\mathsf{W} : Prog^{\mathsf{ij}} \longrightarrow Prog^{\mathsf{j}}$$
$$\mathsf{W}(P) = \mathsf{weaveP}(P, P)$$

Figure 5.9: Weaving functions

a sequence of ij classes (i.e. an ij program) in the context of a program P to give a sequence of woven classes (i.e. a j program). Function W in figure 5.9 applies weaving to a whole program P by weaving each class in P in the context of P itself.

Weaving the subject-observer example

For convenience, the following variable bindings are defined:

$$
\begin{array}{ll}
F_1 & = \texttt{bool flag} \\
F_2 & = \texttt{Observer obs} \\
M_1 & = \texttt{Object update(Subject x)\{...x.flag...\}} \\
M_2 & = \texttt{Object changed(Object x)\{obs.update(this)\}} \\
M_3 & = \texttt{Object setObserver (Observer x)\{this.obs:=x\}} \\
I_1 & = \texttt{Subject} \leftarrow F_2 \\
I_2 & = \texttt{Subject} \leftarrow M_2 \\
I_3 & = \texttt{Subject} \leftarrow M_3 \\
X_s & = \texttt{class Subject ext Object } \{F_1\} \\
X_o & = \texttt{class Observer ext Object } \{M_1\ I_1\ I_2\ I_3\} \\
P & = X_s X_o \epsilon
\end{array}
$$

Now, to weave P_{so}:

$$
\begin{array}{ll}
\mathsf{W}(P) & = \mathsf{weaveP}(P, P) \\
& = \mathsf{weaveP}(P, X_s :: X_o :: \epsilon) \\
& = \mathsf{weaveC}(P, X_s) +\!\!+ \mathsf{weaveP}(P, X_o :: \epsilon)
\end{array}
$$

let $X'_s = \mathsf{weaveC}(P, X_s), \quad X'_o = \mathsf{weaveC}(P, X_o)$

$$
\begin{array}{ll}
\mathsf{W}(P) & = X'_s +\!\!+ \mathsf{weaveP}(P, X_o :: \epsilon) \\
& = X'_s +\!\!+ X'_o +\!\!+ \mathsf{weaveP}(P, \epsilon) \\
& = X'_s +\!\!+ X'_o +\!\!+ \epsilon \\
& = X'_s +\!\!+ X'_o \\
& = X'_s\ X'_o
\end{array}
$$

where

$$
\begin{array}{ll}
X'_s & = \mathsf{weaveC}(P, X_s) \\
& = \texttt{class Subject ext Object } \{D_1..D_p\}
\end{array}
\tag{5.1}
$$

and

$$
\begin{array}{ll}
D_1..D_p & = \mathsf{DecsPC}(P_{so}, \texttt{Subject}) \\
& = \mathsf{DeclarationsX}^{\vec{\mathsf{ij}}}\Big(P/_{\{\texttt{Subject}\}}\Big) \ +\!\!+ \\
& \qquad \mathsf{DeclarationsI}\Big(\mathsf{Intros}^{\vec{\mathsf{ij}}}(P_{so})/_{\{\texttt{Subject}\}}\Big) \\
& = \mathsf{DeclarationsX}^{\vec{\mathsf{ij}}}(X_s) +\!\!+ \mathsf{DeclarationsI}\Big(I_1\ I_2\ I_3/_{\{\texttt{Subject}\}}\Big) \\
& = F_1 +\!\!+ \mathsf{DeclarationsI}(I_1\ I_2\ I_3) \\
& = F_1 +\!\!+ F_2\ M_2\ M_3 \\
& = F_1\ F_2\ M_2\ M_3
\end{array}
$$

substituting into 5.1 gives

$$
X'_s = \texttt{class Subject ext Object } \{F_1\ F_2\ M_2\ M_3\} \tag{5.2}
$$

and

$$
\begin{array}{ll}
X'_o & = \mathsf{weaveP}(P, X_o) \\
& = \texttt{class Observer ext Object } \{D_1..D_p\}
\end{array}
\tag{5.3}
$$

where

$$
\begin{aligned}
D_1..D_p &= \mathsf{DeclarationsX}^{\text{ij}}(P_{so}, \texttt{Observer}) \\
&= \mathsf{DeclarationsX}^{\text{ij}}\big(P_{so}/_{\{\texttt{Observer}\}}\big) \ +\!\!+ \\
&\qquad \mathsf{DeclarationsI}\Big(\mathsf{Intros}^{\text{ij}}(P_{so})/_{\{\texttt{Observer}\}}\Big) \\
&= \mathsf{DeclarationsX}^{\text{ij}}(X_o) +\!\!+ \mathsf{DeclarationsI}\big(I_1\ I_2\ I_3/_{\{\texttt{Observer}\}}\big) \\
&= M_1 +\!\!+ \mathsf{DeclarationsI}(\epsilon) \\
&= M_1 +\!\!+ \epsilon \\
&= M_1
\end{aligned}
$$

substituting into 5.3 gives:

$$X'_o = \texttt{class Observer ext Object } \{M_1\}$$

so

$$
\begin{aligned}
\mathsf{W}(P) = \ & \texttt{class Observer ext Object } \{F_1\ F_2\ M_2\ M_3\} \\
& \texttt{class Observer ext Object } \{M_1\}
\end{aligned}
$$

⌟

5.3.1 Observations on weaving

Weaving a program takes each class in the program in turn and weaves into it the members introduced into it by classes in the program. Weaving an ij class involves discarding its introductions and setting its body to be all the members defined for it by itself or other classes in the program. The result is a class with a set of methods and fields, and no introductions, i.e. a j class. Thus weaving an ij program results in a j program.

Conjecture 2. *Weaving never fails. The result of weaving any program in* ij*, is always defined as some* j *program.*

$$\forall\, P \in Prog^{\text{ij}} \bullet \mathsf{W}(P) \in Prog^{\text{j}}$$

What can go wrong, however, is that a class might end up with more than one member with the same name. This is disallowed, for j programs, by the rules for well formed programs (figure 4.22): method overloading and field shadowing are excluded from the model languages. This is handled by the rules for structural soundness in section 5.3.2.

5.3.2 Soundness

A j program is structurally sound (judgement $\vdash_{\text{j}} \ P \overset{\triangle}{SS}$) if its type hierarchy is well formed, its classes and interfaces have unique names and each class or interface has unique member names. The structural soundness judgement for j is defined by rule [*SS*] in figure 4.22).

Structural soundness is a necessary but not sufficient condition for well formedness. It is, however, sufficient to allow member lookup (by member and class name) with the functions described in section 5.3.3. Uniqueness of member names means that each class may contain at most one method or field with a particular name. In ij methods and fields can be introduced into classes which means that duplicate names might arise as a result of weaving. Therefore it doesn't make much sense to talk about a structurally sound ij program ($\vdash_{\mathsf{ij}} P \overset{\triangle}{SS}$) since the property of having unique member names might change when the program is woven. Weaving may result in name such clashes if either: a) a member is introduced into a class that already contains a member with that name, or b) two (or more) members with the same name are introduced into the same class.

A small extension of structural soundness, however, will provide a judgement about ij programs which will not give unsound j programs when woven. Rule [*wfi*] in figure

$$[wfi]\ \frac{\forall\,(C \leftarrow D)\,(C' \leftarrow D') \in \mathsf{Intros}^{\ell}(P) \bullet \begin{pmatrix} \mathsf{id}(D) = \mathsf{id}(D') \Longrightarrow C \neq C' \\ \mathsf{F}^{\ell}(P, C, \mathsf{id}(D)) = \epsilon \\ \mathsf{M}^{\ell}(P, C, \mathsf{id}(D)) = \epsilon \end{pmatrix}}{\vdash P \overset{\triangle}{\text{WFI}}}$$

Figure 5.10: Well formed introductions.

5.10 defines the judgement for well-formed introductions, which says: no two introductions in a program introduce declarations with the same name into the same class, furthermore there are no fields or methods already in class C in the program with the name of a declarations to be introduced into C.

When an ij program P_{ij} with $\vdash P_{\mathsf{ij}} \overset{\triangle}{\text{WFI}}$ is woven, declarations will only be introduced into classes that do not already contain a declaration with the same name.

Conjecture 3. *A structurally sound* ij *program with well formed introductions:*

$$\forall\, P \in Prog^{\mathsf{ij}} \bullet \begin{pmatrix} \vdash_{\mathsf{ij}} P \overset{\triangle}{SS} \\ \vdash P \overset{\triangle}{WFI} \end{pmatrix} \Longrightarrow \vdash_{\mathsf{j}} \mathsf{W}(P) \overset{\triangle}{SS}$$

5.3.3 Method and field lookup

Method select ($\mathsf{Ms}^{\mathsf{ij}}(P, C, m)$), field select ($\mathsf{Fs}^{\mathsf{ij}}(P, C, f)$) and header select ($\mathsf{Hs}^{\mathsf{ij}}(P, C, m)$) are defined for ij as per the definitions in figure 4.24. Lookup functions F^{ij}, M^{ij}, H^{ij}, and $\mathsf{C}^{\mathsf{ij}}_{\mathsf{def}}$ are defined in figure 4.24.

⌜ for example

$$\mathsf{F}^{\mathsf{ij}}(P_{list}, \texttt{Subject}, \texttt{flag}) = \texttt{bool flag}$$

⌟

5.3.4 Native semantics

The bulk of the work of weaving is done by function $\mathsf{GetDecs}^{\mathsf{ij}}$ (figure 5.7); The weaving functions in figure 5.9 serve to distribute this functionality over all the classes in the program. In practise, the weaving function would be the only way in which weaving were done, as it is for users of ASPECTJ: the weave is performed implicitly as a step in compilation. The approach taken in this presentation, however, gives another way to look at the weaving process: presenting a semantics of the unwoven program that uses $\mathsf{GetDecs}^{\mathsf{ij}}$ to weave classes at a time as required, thus the ij program itself is given a semantics wherein the definition of each class is distributed over the whole program text and gathered into one place when needed by the semantics.

The $\mathsf{GetDecs}^{\mathsf{j}}$ defined in figure 4.12 is used by all functions that access class members; When the j version is overridden with the ij version in this calculus, a new set of functions is given, with identical signatures, that do all their member lookups on the woven forms of classes.

5.4 Well formedness in ij

The result of weaving, however, is not always a well formed j program; weaving has only to do with resolving introductions. A woven program has its members in the right places, but weaving does not say anything about whether those members are type correct in their final locations. There may be dependencies among introductions, for example an introduced method may call another introduced method so that the former would not type check unless the latter was present. So in order to check the type correctness of introductions it is necessary to resolve them all and then check the resulting program as illustrated in rule [*WFIJP*] in figure 5.11.

$$[\mathit{WFIJP}]\ \frac{\vdash_{\mathsf{j}} \mathsf{W}(P)\ \diamond}{\vdash_{\mathsf{ij}} P\ \diamond}$$

Figure 5.11: Well formedness for ij programs.

5.4.1 Type checking

Environment

Type checking in ij is performed in the context of an environment γ that maps identifiers `this` and `x` onto their types. The definition of γ is the same as in j in figure 4.25. Such an environment is well formed (judgement $P \vdash_{\mathsf{ij}} \gamma\ \diamond$ defined in figure 4.26) if it maps the identifiers (`this` and `x`) to appropriate types.

Type rules

Type rules for expressions and values are defined in figure 4.27 where $\mathsf{M}^{\ddot{\mathsf{ij}}}$ and $\mathsf{F}^{\ddot{\mathsf{ij}}}$, etc. are used to look up members. Method and field declarations are checked in the context of their enclosing class, introductions are checked in the context of the class into which they are introduced.

5.4.2 Well-formed programs

A structurally sound ij program is well formed (judgement $\vdash_{\mathsf{ij}} \ P \diamond$) if each of its classes and interfaces is well formed (judgement $P \vdash_{\ell} \ C \diamond$) and all of their method ($P, C \vdash_{\ell} \ M \diamond$) field ($P, C \vdash_{\ell} \ F \diamond$) and header ($P, C \vdash_{\ell} \ H \diamond$) declarations – including those that are introduced – are well formed.

The rules for well formed declarations in j are defined in figure 4.28. In j, these rules (of the form $P, C \vdash_{\ell} \ D \diamond$) are applied to the member declarations defined in each class, each declaration is checked in the context of the enclosing program and the name of the class in which it was declared. In ij, however, things are not quite so straight forward: the full set of declarations that pertain to a class comprises those defined within the class (as in j) and, additionally, those introduced into it from other classes using the introduction mechanism. To properly validate an introduced declaration it should be checked in the context of the fully woven program, and the name of the class into which it was introduced. Resolving introductions into their target classes is handled by the weaving function (defined in figure 5.9). These properties, therefore, are not checked on unwoven ij programs.

A method declaration is well formed (rule [*WFM*]) if its argument and return types are valid variable types and if its body expression typechecks in an appropriate environment with a type compatible with the method's declared return type; when a method overrides another, the signatures of both methods must be the same. A field declaration is well formed (rule [*WFF*]) if it has a valid type and if it does not hide an inherited field. A method header is well formed (rule [*WFH*]) if its return type and argument type are valid variable types; when a header matches an inherited one, their signatures must be the identical. A class is well formed if it extends either another class in the same program (or `Object`), and implements only interfaces; all its declarations must be well formed and it must provide implementations for all method headers declared in all its super-interface, these must have matching signatures. An interface is well formed if it extends either an interface or `Object` and all its headers are well formed. A program is well formed if it is structurally sound and each of its classes and interfaces are well formed.

Weaving and well-formedness

It would be great if some property of an unwoven program P would guarantee that $\mathsf{W}(P)$ would be well formed as is the case with structural soundness. It is possible to do this with some properties of well-formedness but not all of them.

The requirements of structural soundness are intended to allow the program to be used to lookup fields and methods, thus it is required that member names be unique within a class; it does not deal with software properties of the program such as the fact that field shadowing is not allowed. Field shadowing is disallowed by rule [*WFF*] in figure 4.28. It is possible to capture a condition of unwoven programs that guarantees there will be no field shadowing in the woven program:

$$\forall P, C, C', f, T, T' \bullet \begin{pmatrix} \mathsf{Fs}^{\mathbb{ij}}(P, C, f) \\ C \leftarrow T\ f \in P \end{pmatrix} \Longrightarrow \text{NOT} \begin{pmatrix} C \sqsubseteq C' \\ C' \sqsubseteq C \end{pmatrix}$$

This condition says that if the field introduction $C \leftarrow T\ f$ exists in P and some class C' contains a field called f, then there must be no inheritance relationship between C' and C.

Another property of a well-formed program is that the body expressions of all methods typecheck in the context of their owning method and class. In ij, however, the set of methods and fields available in the owning class is only known after weaving therefore this type check can only be performed after weaving. Since the woven program must be calculated in order to test these properties, this becomes a post-weaving check, applied to a woven program, i.e. an ij program is well formed if it weaves into a well formed j program:

$$\frac{\vdash_{\mathbb{ij}} P \diamond}{\vdash_{\mathbb{j}} \mathsf{W}(P) \diamond}$$

5.5 Operational semantics

Programs in ij behave the same as programs in j; the languages differ in how those programs are expressed in terms of where member declarations are located.

A formal semantics can be obtained in two different ways: the semantics via weaving takes execution as a post-weaving process (as it would be in ASPECTJ since weaving is part of the processing done by the compiler); the 'native' semantics presents a set of rules similar to those for j but in which the $\mathsf{GetDecs}^{\mathbb{j}}$ function is overridden by $\mathsf{GetDecs}^{\mathbb{ij}}$ so that member lookup in a class C is performed in the full set of declarations for C, including the introduced ones.

5.5.1 Semantics via weaving

Weaving ij programs gives j programs. It is, therefore, possible to define a semantics for ij in terms of translation into j via weaving. This is illustrated in the rule in figure 5.12 (note that $h' \simeq h''$ means that h' and h'' are equal up to renaming of addresses).

The semantics of j is defined by the reduction rules presented in chapter 4.

$$\frac{h' \doteq h'' \quad \mathsf{W}(P) \vdash_{\mathsf{j}} e, h, s \rightsquigarrow v, h'}{P \vdash_{\mathsf{ij}} e, h, s \rightsquigarrow v, h''}$$

Figure 5.12: Rule for semantics via weaving

5.5.2 'Native' semantics

The semantics of ij is obtained from the semantics of j by allowing all references to $\mathsf{GetDecs}^{\mathsf{j}}$ to be replaced by references to $\mathsf{GetDecs}^{\mathsf{ij}}$. The former (defined in figure 4.12) gives the sequence of declarations defined within the definition of the given class in the given program (i.e. the value of $\mathsf{DeclarationsX}^{\mathsf{j}}$); the latter (defined in figure 5.7) gives the sequence containing the declarations defined within the definition of the given class together with those defined as introductions into that class by any class in the program.

The definition of runtime values, objects, heaps, stack frames, runtime types and reduction rules remain the same as in j with the exception that all direct or in direct references to $\mathsf{GetDecs}^{\mathsf{j}}$ are overridden with references to $\mathsf{GetDecs}^{\mathsf{ij}}$. This affects the model in the following places:

- Function GetFields (figure 4.12), which is used in the definition of AllFields (figure 4.20) used by new (figure 4.32) which creates a new object with all its declared and inherited fields appropriately initialised.
- Function $\mathsf{GetMethods}^{\ell}$ (figure 4.12) which is used in the definition of function Ms (figure 4.23), for selecting methods, which in turn is used in the definition of function M (figure 4.24) which is used in the rule for executing method call expressions in figure 4.35.

The semantics of field lookup is indirectly affected by the change: references to object fields (as performed by rules [*RWGET*] and [*RWSET*] in figure 4.35) use lookups into the heap, which contains objects created using the new function which is affected as described above.

Executing subject observer example

Consider the class definitions X_s and X_o from the weaving example earlier in this chapter. Together with the following "main" class, they form a program whose entry

point is execution of method `main` on an instance of class `Main`.

$F_3 =$ `Subject s`

$M_4 =$
```
Object main(Object x){
  this.s:=new Subject
  this.s.setObserver(new Observer)
  this.flag:=true
  this.changed(null)
}
```

$X_m =$ `class Main ext Object` $\{F_3\ M_4\}$

$P = X_s\ X_m\ X_o$

Executing this program creates a `Subject` object, attaches a new `Observer` object to it and changes the value of the subject's flag field; the final call to the subject's `changed` method causes the associated observer's `update` method to be called. Running the program means executing expression $e =$ `new Main.main()` with heap $h_0 = \emptyset$ and stack $s_0 = \emptyset$, i.e.: $P \vdash_{\mathsf{ij}} e, h_0, s_0 \rightsquigarrow v, h'$ by rule [*RWMETH*]. This reduces $P \vdash_{\mathsf{ij}}$ `new Main`$, h_0, s_0 \rightsquigarrow v, h'$ giving $v = \iota_1$ and $h' = h_1 = \{\iota_1 \mapsto [\![\mathsf{s{:}null}]\!]^{\mathtt{Main}}\}$, and then invokes the body of method M_4 (`main`) in h_1 with $s = \{\mathtt{this} \mapsto \iota_1, \mathsf{x} \mapsto \mathtt{null}\}$. The body statements are executed in turn by repeated application of [*RWSEQ*]. `this.s:=new Subject` gives $h_2 = h'[\iota_1 \mapsto h'(\iota_1)[\mathsf{s} \mapsto \iota_2]]$ where $h' = \{\iota_1 \mapsto [\![\mathsf{s{:}null}]\!]^{\mathtt{Main}}, \iota_2 \mapsto [\![\mathtt{flag{:}false}, \mathtt{obs{:}null}]\!]^{\mathtt{Subject}}\}$, i.e. $h_2 = \{\iota_1 \mapsto [\![\mathsf{s{:}}\iota_2]\!]^{\mathtt{Main}}, \iota_2 \mapsto [\![\mathtt{flag{:}false}, \mathtt{obs{:}null}]\!]^{\mathtt{Subject}}\}$

Note that: $\mathsf{new}^{\mathsf{ij}}(P, \mathtt{Subject}) = [\![\mathtt{flag{:}false}, \mathtt{obs{:}null}]\!]^{\mathtt{Subject}}$ (including introduced field `obs`) since

$$
\begin{aligned}
&\mathsf{AllFields}^{\mathsf{ij}}(P, \mathtt{Subject}) \\
&\quad = \mathsf{Fields}'\Big(\mathsf{GetDecs}^{\mathsf{ij}}(P, \mathtt{Subject})\Big) \\
&\quad \text{(using the redefined version of DeclarationsX for ij)} \\
&\quad = \mathsf{Fields}'\begin{pmatrix}\mathsf{DeclarationsX}^{\mathsf{ij}}\Big(P/_{\{\mathtt{Subject}\}}\Big) +\!\!+ \\ \mathsf{DeclarationsI}\Big(\mathsf{Intros}^{\mathsf{ij}}(P)/_{\{\mathtt{Subject}\}}\Big)\end{pmatrix} \\
&\quad = \mathsf{Fields}'\Big(F_1 +\!\!+ \mathsf{DeclarationsI}\Big(\mathsf{Intros}(P)/_{\{\mathtt{Subject}\}}\Big)\Big) \\
&\quad = \mathsf{Fields}'(F_1 +\!\!+ \mathsf{DeclarationsI}(I_1..I_3)) \\
&\quad = \mathsf{Fields}'(F_1 +\!\!+ F_2\ M_2\ M_3) \\
&\quad = F_1\ F_2 \\
&\quad = \mathtt{bool\ flag\ \ Observer\ obs}
\end{aligned}
$$

`this.s.setObserver(new Observer)` invokes introduced method M_3; rule [*RWMETH*] requires $P \vdash_{\mathsf{ij}}$ `this.s`$, h_2, s' \rightsquigarrow \iota_2, h_2$ and $P \vdash_{\mathsf{ij}}$ `new Observer`$, h_2, s' \rightsquigarrow \iota_3, h_3$ where $h_3 = \{\iota_1 \mapsto [\![\mathsf{s{:}}\iota_2]\!]^{\mathtt{Main}}, \iota_2 \mapsto [\![\mathtt{flag{:}false}, \mathtt{obs{:}null}]\!]^{\mathtt{Subject}}, \iota_3 \mapsto [\![]\!]^{\mathtt{Observer}}\}$ and $s' = \{\mathtt{this} \mapsto \iota_1, \mathsf{x} \mapsto \mathtt{null}\}$ Then M is used to look up the

method to invoke:

$$
\begin{aligned}
&\mathsf{M}^{\ddot{\mathsf{ij}}}(P, \texttt{Subject}, \texttt{setObserver}) \\
&\quad = \mathsf{GetMethods}^{\ddot{\mathsf{ij}}}(P, \texttt{Subject})/_{\{\texttt{setObserver}\}} \\
&\quad = \mathsf{Methods}'\Big(\mathsf{GetDecs}^{\ddot{\mathsf{ij}}}(P, \texttt{Subject})\Big)/_{\{\texttt{setObserver}\}} \\
&\quad \text{(using the redefined version of DeclarationsX for ij)} \\
&\quad = \mathsf{Methods}'\left(\begin{array}{l}\mathsf{DeclarationsX}^{\ddot{\mathsf{ij}}}\Big(P/_{\{\texttt{Subject}\}}\Big) +\!\!+ \\ \mathsf{DeclarationsI}\Big(\mathsf{Intros}^{\ddot{\mathsf{ij}}}(P)/_{\{\texttt{Subject}\}}\Big)/_{\{\texttt{setObserver}\}}\end{array}\right) \\
&\quad = \mathsf{Methods}'(F_1 +\!\!+ \mathsf{DeclarationsI}(I_1..I_3))/_{\{\texttt{setObserver}\}} \\
&\quad = \mathsf{Methods}'(F_1 +\!\!+ F_2\ M_2\ M_3)/_{\{\texttt{setObserver}\}} \\
&\quad = (M_2\ M_3)/_{\{\texttt{setObserver}\}} \\
&\quad = M_3
\end{aligned}
$$

⌟

5.5.3 Equivalence of Semantics

The two ways of expressing the semantics should give equivalent results. One gives the semantics of a program after weaving has been applied to the whole, the other gives the same semantics but applying the essential function of weaving to class definitions as needed. This expectation is captured by conjecture 4 below.

Conjecture 4.

$$\forall\, P, C, f, m \bullet P \vdash_{\ddot{\mathsf{ij}}} e, h, s \rightsquigarrow v, h' \iff \mathsf{W}(P) \vdash_{\mathsf{ij}} e, h, s \rightsquigarrow v, h'$$

5.6 Concept List

Term	Meaning	Location
$\mathsf{Introductions}^{\ddot{\mathsf{j}}}(X)$	Projects (sequence of) introductions from a class or interface definition X	figure 5.4
$\mathsf{Intros}^{\ddot{\mathsf{j}}}(P)$	(Sequence of) all introductions in program P	figure 5.5
$\mathsf{GetDecs}^{\ddot{\mathsf{j}}}(P, C)$	(Sequence of) all declarations for class C in program P, this includes those defined in class C and those introduced into C by other classes in P	figure 5.7
$\mathsf{GetFields}^{\ddot{\mathsf{j}}}(P, C)$	All fields defined within or introduced to C	figure 4.12
$\mathsf{GetMethods}^{\ddot{\mathsf{j}}}(P, C)$	All methods defined within or introduced to C	figure 4.12
$\vdash_{\ddot{\mathsf{j}}} P \overset{\triangle}{SS}$	Structural soundness	figure 4.22
$\mathsf{Fs}^{\ddot{\mathsf{j}}}, \mathsf{Ms}^{\ddot{\mathsf{j}}}, \mathsf{Hs}^{\ddot{\mathsf{j}}}$	Field and method select	figure 4.23
$\mathsf{F}^{\ddot{\mathsf{j}}}, \mathsf{M}^{\ddot{\mathsf{j}}}, \mathsf{H}^{\ddot{\mathsf{j}}}, \mathsf{C}^{\ddot{\mathsf{j}}}_{\mathsf{def}}$	Defined or inherited fields, methods	figure 4.24
γ	Type checking environment	figure 4.25
$P \vdash_{\ddot{\mathsf{j}}} \gamma \diamond$	well formed context	figure 4.26
$P, \gamma \vdash_{\ell} e : T$	type inference	figure 4.27
$P, C \vdash_{\ddot{\mathsf{j}}} D \diamond$	Well formed member declarations	figure 4.28
$P \vdash_{\ddot{\mathsf{j}}} C \diamond$	Well formed class or interface	figure 4.28
$\vdash_{\ddot{\mathsf{j}}} P \diamond$	Well formed program	figure 4.28
$[\![f{:}v \ldots]\!]^C$	Objects	figure 4.31
$\mathsf{new}^{\ddot{\mathsf{j}}}(P, C)$	New object of class C	figure 4.32
h, s	Heaps and stack frames	figure 4.33
$P \vdash_{\ddot{\mathsf{j}}} e, h, s \rightsquigarrow v, h'$	Rewrite relation	figure 4.35
$\mathsf{W}(P)$	Weave introductions into their target classes	figure 5.9
$P, C \vdash_{\ddot{\mathsf{j}}} D \diamond$	Well formed member declarations	figure 4.28
$P \vdash_{\ddot{\mathsf{j}}} C \diamond$	Well formed class	figure 4.28
$\vdash_{\ddot{\mathsf{j}}} P \diamond$	Well formed programs in j	figure 4.28

Chapter 6

A language with parent introduction

6.1 The language pj

pj is an extension of ij that supports parent introduction, that is to say, it allows a class to contain a declaration that changes the superclass, or implemented interface list, of another class.

6.2 Syntax

The introduction concept of ij is kept and an additional construct *parent introduction* ($ParentIntro^{\mathsf{pj}}$) is added. Figure 6.1 defines $Class^{\mathsf{pj}}$ and $Prog^{\mathsf{pj}}$ together with $ParentIntro^{\mathsf{pj}}$. The base syntax is given in figure 4.3.

$P \in Prog^{\mathsf{pj}}$	::=	X^*
$X \in Class^{\mathsf{pj}}$	::=	`class` C `ext` C `[ imp` C^* `]{` D^* I^* S^* `}`
$S \in ParentIntro^{\mathsf{pj}}$	::=	C `ext` C | C `imp` C

Figure 6.1: Syntax of pj; Class type $Class^{\mathsf{j}}$ is introduced, with parent introduction $ParentIntro^{\mathsf{pj}}$, all other productions remain unchanged.

Two new kinds of introduction are added:

- superclass introductions of the form: C `ext` C'
- super-interface introductions of the form: C `imp` C'

These introduce parent relationships. Formally, they introduce edges into the graph that constitutes the class hierarchy. Arbitrary changes to the class hierarchy might result in broken programs:

- if a program requires instances of some class C to be substitutable, by polymorphism, in contexts where C' is expected, it will fail if C's ancestry is modified so that C' is no longer a supertype or super-interface of C.
- if a class's methods depend on attributes or methods inherited from a superclass, removing that superclass would break the class
- a program will break if a parent-introduction introduces a cycle in the inheritance graph

Therefore, pj must not allow arbitrary changes to the class hierarchy. Parent introduction must be additive, so that each class keeps all its declared superclasses and super-interfaces in addition to any that are introduced into it. Superclass and super-interface relationships can be introduced into a pj program, provided that they do not break the existing type hierarchy.

Examples of parent introduction

According to the syntax of pj, we may write, for example:

```
interface I ext Object {}
class C ext Object {}
class A ext Object { C ext A  C imp I  }
```

which is equivalent to

```
interface I ext Object {}
class C ext A imp I {}
class A ext Object { }
```

The two kinds of introduction (member introduction and parent introduction) can be combined in a compelling way: a class can be made to extend some interface and at the same time given the methods necessary to implement the interface's methods. In order to write the example below, a simple extension to pj is assumed which allows the type `int` for integer values. In figure 6.2 the purpose of class `A` is merely to package together the two introductions to class `Point` thus it is behaving like an aspect in ASPECTJ. The program above is equivalent to the valid program below; figure 6.3 note that this program is valid because class `Point` declares that it implements `Clonable` and it also declares an implementation for method `clone` defined in `Clonable`. Thus, the two kinds of introduction work together to define a valid program.

6.2.1 Functions on the syntax

In figure 6.4, the following functions are defined on the syntax of pj: $\mathsf{classId}^{\mathsf{pj}}(X)$, $\mathsf{superClass}^{\mathsf{pj}}(X)$, $\mathsf{DeclarationsX}^{\mathsf{ij}}(X)$, $\mathsf{Interfaces}^{\mathsf{pj}}(X)$, $\mathsf{Headers}^{\mathsf{pj}}(X)$ and $\mathsf{Introductions}^{\mathsf{pj}}(X)$.

```
interface Clonable ext Object {
  Object clone(Object x)
}
class Point ext Object {
  int x
  int y
  Point setX(int x){
    this.x = x
    this
  }
  Point setY(int y){
    this.y = y
    this
  }
}
class A ext Object {
  Point <--  Object clone(Object x){
                new Point .setX(this.x).setY(this.y)
             }
  Point imp Clonable
}
```

Figure 6.2: An aspect makes Point clonable.

```
interface Clonable ext Object {
  Object clone(Object x)
}
class Point ext Object imp Clonable {
  int x
  int y
  Point setX(int x){
    this.x = x
    this
  }
  Point setY(int y){
    this.y = y
    this
  }
  Object clone(Object x){
    new Point.setX(this.x).setY(this.y)
  }
}
class A ext Object {}
```

Figure 6.3: Clonable point class.

In addition $\mathsf{P}_{\mathsf{i}}^{\mathsf{pj}}(X)$ is defined, which projects the parent introductions form a class or interface.

For a class : $Y = \texttt{class}\ C\ \texttt{ext}\ C'\ \texttt{imp}\ C''_1..C''_k\{D_1..D_n\ I_1..I_p\ S_1..S_q\}$:

$$\begin{aligned}
\mathsf{classId}^{\mathsf{pj}}(Y) &= C \\
\mathsf{superClass}^{\mathsf{pj}}(Y) &= C' \\
\mathsf{DeclarationsX}^{\mathsf{pj}}(Y) &= D_1..D_n \\
\mathsf{Interfaces}^{\mathsf{pj}}(Y) &= C''_1..C''_k \\
\mathsf{Headers}^{\mathsf{pj}}(Y) &= \epsilon \\
\mathsf{Introductions}^{\mathsf{pj}}(Y) &= I_1..I_p \\
\mathsf{Pi}^{\mathsf{pj}}(Y) &= S_1..S_q
\end{aligned}$$

Figure 6.4: Functions that project components of class terms in language pj

For an interface: $W = \texttt{interface}\ C\ \texttt{ext}\ C'\ \{H_1..H_k\}$:

$$\mathsf{Pi}^{\ell}(W) = \epsilon$$

Figure 6.5: Generic function for projecting the empty sequence of parent introductions from an interface

6.2.2 Declarations, fields and methods

$\mathsf{DeclarationsX}^{\mathsf{pj}}$ is the same as $\mathsf{DeclarationsX}^{\mathsf{ij}}$ (defined in figure 5.7) and used in the definition of $\mathsf{GetFields}^{\mathsf{pj}}$ and $\mathsf{GetMethods}^{\mathsf{pj}}$ (in figure 4.12).

6.2.3 Unique names and field lookup

Judgement $\vdash_{\mathsf{pj}} P\ \overset{\triangle}{uc}$ (defined in figure 4.13) states that the classes and interfaces in program P have unique names. Domain restriction can be used to select classes by name from such programs.

6.2.4 Types

Types in pj work the same way as in j and ij. Judgements $P \vdash_{\mathsf{pj}} C\ \overset{\triangle}{ct}$, $P \vdash_{\mathsf{pj}} C\ \overset{\triangle}{it}$, $P \vdash_{\mathsf{pj}} C\ \overset{\triangle}{ot}$ and $P \vdash_{\mathsf{pj}} T\ \overset{\triangle}{vt}$ defined in figure 4.14, say that the given identifier is a class, interface, object or variable type, respectively.

6.2.5 Class and type hierarchy

Parent relationships between classes and interfaces in ij arise as the result of two mechanisms:

– using the the `ext` and `imp` keywords in class and interface headers (just as in j).
– parent introductions in the body of classes.

Parent introduction

Parent introduction allows one class to affect the ancestry of another. This is different from the situation in most class based languages wherein the type hierarchy is derived only from declarations in the class headers (as is the case in j and ij). The j family of calculi are based on JAVA and as such they permit only single inheritance, this means that every class has a unique superclass (though it might have many super-interfaces), as enforced by the rule in figure 4.18. In j and ij a well formed program may list only the name of one class as its superclass and may list only the names of interfaces as its super-interfaces as enforced by rule [*WFC*] in figure 4.28. In pj, however, there may be multiple parent introductions for a single class, in addition to that class's own declared superclass. It is necessary to prescribe, therefore, what it means for a ij program to contain multiple statements about a class's parents.

– One possibility is to extend pj to support multiple inheritance. This would be a departure from the intention to model ASPECTJ. It would also require several decisions to be made regarding which inheritance hierarchies are well formed, and which are not. By making these decisions for a language with single inheritance, a more insightful model of ASPECTJ is obtained, thus pj does not allow multiple inheritance.
– Another possibility is to allow only single inheritance and to reject as invalid any program in which there are multiple conflicting statements about what the parent of a class is. This approach could be encoded by a rule that considers pairs of introductions or superclass clauses and rejects the program if any pair disagree on the superclass of some class. The syntax of pj prescribes that each class declares its superclass in its header; thus no introduction may disagree with any class header about the superclass of a class. This disempowers parent introduction, preventing it from saying anything useful about classes; the only thing that can usefully be said in this case is that a class has additional interfaces.
– Another possibility is to allow only single inheritance and to allow sets of parent introductions and superclass clauses that, considered together, give a well formed hierarchy in which each class has a single superclass and all a classes superclasses and super-interfaces, as defined by the class header, are still superclasses and super-interfaces after weaving. This option requires all the parent introductions and superclass clauses to be considered together as a whole for the purpose of checking well-formedness, or more accurately, to consider the hierarchy that they define (the hierarchy is the transitive closure of these declarations).

The last option is the one chosen for pj: single inheritance is required; parent introduction is permitted; a program P containing a collection of parent introductions is valid, with respect to subtyping, provided the result of weaving those introductions is a program P' (with single inheritance) in which all subtype relations defined in class and interface headers in P still hold. The essence of this approach is that whereas traditionally in JAVA-like languages the class header is considered to declare, absolutely,

the *immediate* superclass, in pj the class header is is considered merely as a list of subclass relationships to be taken into account when calculating the subclass hierarchy. Additional relationships are provided by parent introductions. All such relationships must be taken into account to calculate the hierarchy.

For example the program in figure 6.6 contains superclass declarations that appear to disagree about the superclass of class B: class B declares that it has C as its superclass and class A contains a parent introduction that says C is B's superclass. The parent

```
class C ext Object {}
class D ext Object {}
class B ext C {}
class A ext Object {
  B ext D
}
```

Figure 6.6: A program with an ill-formed hierarchy.

declarations (in the class headers) define the hierarchy in figure 6.7(a). (Class A is not included in the diagram). The parent introduction in A adds the relation between B and D as shown in figure 6.7(b). The contradiction arises because B now seems to have

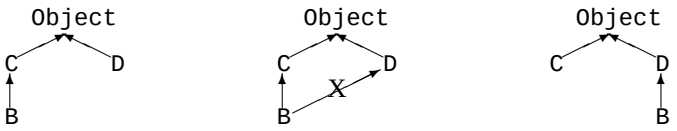


Figure 6.7: Three class hierarchies (a) left, (b) centre and (c) right

two distinct superclasses, C and D, which is not allowed with single inheritance. One way to proceed would be to allow parent introductions to override the conventional declarations then, in the example, the resulting hierarchy would be that in figure 6.7(c). This,however, is not type safe. It would be problematic if the program contained code which relied on the fact that B is a subclass of C, for example if an object of class B were passed as the actual parameter to a method with formal parameter of type C. A pessimistic solution to this is to reject, as badly formed, any parent introduction for that contradicts the hierarchy defined in class and interface headers. This would correctly reject as invalid, the program in figure 6.6 but it would also reject the program in figure 6.8 which, as described below, is well formed if its hierarchy is considered as a whole.

In the following, similar, example there is no problem associated with making B a subclass of D.

The declarations in class headers declare the same hierarchy as in the previous example (see figure 6.9(a)); the parent introductions assert that B be a subclass of D but also introduce a relation between D and C such that D is a subclass of C and therefore subclass of D will also be subclasses of C. In this way the assertion (from the class headers) that B is a subclass of C, is not contradicted. The resulting hierarchy is shown in figure 6.9(b).

```
class C extends Object {}
class D extends Object {}
class B extends C {}
class A extends Object {
  B ext D
  D ext C
}
```

Figure 6.8: A program with a hierarchy that is valid.

Figure 6.9: Two class hierarchies: (a) left and (b) right.

ASPECTJ has a forgiving approach that will not reject the program in figure 6.8: parent introductions are allowed to change the hierarchy provided that every class remains a subclass of each of its superclasses, and super-interfaces, as defined in the class and interface headers. In order to implement this policy, ASPECTJ acts upon a number of classes together as compilation unit (referred to as a program in this thesis). Considering the program as a whole, if the hierarchy defined by all the headers and introductions is valid, then the whole set of introductions are accepted. ⌟

Subtype hierarchies

The same approach is taken to validating the hierarchy in pj as in ij: the hierarchy Γ_P is projected from the program and validation rules are applied to the hierarchy. The validation rules are the same in pj as they are in j, since the parent introductions change the hierarchy but should define a valid j program. The difference is in how the hierarchy is obtained from the program.

The definition of type hierarchy Γ_P given in figure 4.15 for programs in j does not take into account subtype relations defined by parent introduction. Judgement $P \vdash_{\mathsf{pj}} C \leq C'$ (defined in figure 6.10) asserts that class C is a subtype of class C' because of a subtype relation introduced by a parent introduction.

$$\frac{\exists\, C'' \bullet \mathsf{Pi}(P, C'') = \ldots C \texttt{ ext } C' \ldots}{P \vdash_{\mathsf{pj}} C \leq C'} \qquad \frac{\exists\, C'' \bullet \mathsf{Pi}(P, C'') = \ldots C \texttt{ imp } C' \ldots}{P \vdash_{\mathsf{pj}} C \leq C'}$$

Figure 6.10: Additional subtype rules for pj

A partial hierarchy, called the introduced hierarchy Γ^i_P, is defined (in figure 6.11) in terms of the rules in figure 6.10. This partial hierarchy represents the abstraction of the introduced subtype relation; it must be combined with the hierarchy that represents the abstraction of the definitions in class headers in order to obtain the complete program hierarchy for a pj program.

$$\Gamma^i_P = \{(C, C') | P \vdash_{\mathsf{pj}} C \leq C'\}$$

Figure 6.11: Introduced subtype relation

The complete subtype relation Γ_P for a pj program is derived from the union of the class header hierarchy Γ^{j}_P with the introduced hierarchy Γ^i_P. Since Γ^i_P contains class relationships additional to those in the class header hierarchy it is necessary to take the transitive closure of this union in order to ensure transitivity of the resulting hierarchy Γ_P. This is done in the definition, in figure 6.12 of Γ^{pj}_P. It is the complete hierarchy,

$$\Gamma^{\mathsf{pj}}_P = \mathsf{transitiveClosure}\Big(\Gamma^{\mathsf{j}}_P \cup \Gamma^i_P\Big)$$

Figure 6.12: The combined subtype hierarchy.

Γ^{pj}_P, that will be used in the rest of the semantic model.

Well formed hierarchies

The rules for well-formedness of a complete pj hierarchy, Γ^{pj}_P, are the same as those for j, as defined in figures 4.17 and 4.18: the hierarchy must be a complete partial order with a unique superclass for each class and `Object` as the top element.

Super

Function $\mathsf{super}^{\mathsf{pj}}$, is defined in figure 4.19. It gives the unique superclass for the given class, this is derived from the hierarchy Γ_P. If this function is applied to a pj program, and the pj complete hierarchy Γ^{pj}_P is used, then the unique superclass after weaving parent introductions is given. This is true provided that the hierarchy is well formed, which is to say that there is a unique superclass for each class in the program.

All fields

Function $\mathsf{AllFields}^{\mathsf{pj}}(P, C)$ is defined in figure 4.20. It uses $\mathsf{GetFields}^{\mathsf{pj}}(P, C)$ to get the fields, and therefore, indirectly, $\mathsf{Declarations}^{\mathsf{ij}}$ which takes into account the introduced fields. Furthermore, it is defined in terms of $\mathsf{super}^{\mathsf{pj}}$, and so uses the pj hierarchy that takes into account parent introduction.

6.2.6 Soundness

Structural soundness

The total subtype relation for a pj program is required to honour the same conditions as for a j program. This requirement is encoded in the rule for structural soundness (defined in figure 4.22) which requires $P \vdash_{\mathsf{pj}} \Gamma_P \diamond$ and $\vdash \Gamma \overset{\triangle}{po}$. The other requirements for $\vdash_{\mathsf{pj}} P \overset{\triangle}{SS}$ are unique class and member names.

6.2.7 Method and field lookup

Functions $\mathsf{Fs}^{\mathsf{pj}}(P, C, f)$, $\mathsf{Ms}^{\mathsf{pj}}(P, C, m)$ and $\mathsf{Hs}^{\mathsf{pj}}(P, C, m)$ are defined in figure 4.23 and use $\mathsf{Declarations}^{\mathsf{pj}}$ (defined in figure 5.7). They are used in the definition of lookup functions $\mathsf{Ms}^{\mathsf{pj}}(P, C, m)$, $\mathsf{Fs}^{\mathsf{pj}}(P, C, f)$ and $\mathsf{Hs}^{\mathsf{pj}}(P, C, m)$ (figure 4.24). Lookup functions F^{pj}, M^{pj}, H^{pj}, and (defined in figure 4.24) search among the declared and introduced members of the named class for the named member. If it is not found, they search up the type hierarchy using $\mathsf{super}^{\mathsf{pj}}(P, C)$ (figure 4.19) which implicitly uses Γ_P^{pj}, the augmented hierarchy that includes the effect of parent introduction.

6.3 Well formedness in pj

Type checking

Type checking for pj is similar to that for j. Environments, γ, define type mappings for `this` and x just as in j:

$$\gamma^{\mathsf{pj}} : (\{\texttt{this}\} \longrightarrow ClassName) \cup (\{x\} \longrightarrow Type)$$

with update and select operations as defined in figure 4.25 and well-formedness defined in figure 4.26.

The rules for type checking are syntactically the same as those for j and ij (figure 4.27) where references to Γ_P refer to the complete pj hierarchy Γ_P^{pj}. Similarly, subtype judgements ($P \vdash C \leq C'$) are made in the context of the complete pj hierarchy Γ_P^{pj}.

Well formed programs

Just as in ij, a structurally sound pj program is well formed if all its classes' defined and introduced members are well formed according to the rules in figure 4.28. A structurally sound program will have a sound type hierarchy that allows inheritance and overloading of members as resolved by the lookup functions F^{pj}, M^{pj}, etc.

6.4 Weaving

Weaving in pj is the same as weaving in ij except that it also has to resolve the changes in the supertype hierarchy. The total hierarchy Γ_P must be mapped onto the extends and implements declarations in j. The weaving function W (defined in figure 5.9) is defined in terms of $\mathsf{weaveP}(P, P)$ which is defined in terms of $\mathsf{weaveC}(P, X)$. Function weaveC is redefined for pj in figure 6.13.

$$\mathsf{weaveC}^{\mathsf{pj}} : Prog^{\mathsf{pj}} \times Class^{\mathsf{pj}} \longrightarrow Class^{\mathsf{j}}$$

$$\mathsf{weaveC}^{\mathsf{pj}}(P, \texttt{class}\ C\ \texttt{ext}\ C'\ \texttt{imp}\ C_1..C_p\{D_1..D_j I_1..I_n\}) = \texttt{class}\ C\ \texttt{ext}\ C''\ \texttt{imp}\ C'_1..C'_p\{D'_1..D'_k\}$$

where
$$\begin{array}{lll} D'_1..D'_k & = \mathsf{GetDecs}^{\mathsf{pj}}(P, C) \\ C'' & = \mathsf{super}^{\mathsf{pj}}(P, C) \\ S & = \{C''' | P \vdash_{\mathsf{pj}} C''' \overset{\triangle}{_{it}}, C \sqsubseteq C'''\} \\ C'_1..C'_q & \in \{C_1..C_{\#S} | \{C_1..C_{\#S}\} = S\} \end{array}$$

$$\mathsf{weaveC}^{\mathsf{pj}}(P, \texttt{interface}\ C\ \texttt{ext}\ C'\ \{H_1..H_n\}) = \texttt{class}\ C\ \texttt{ext}\ C''\ \texttt{imp}\ C'_1..C'_p\{D'_1..D'_k\}$$

where
$$\begin{array}{lll} S & = \{C''' | C \sqsubseteq C'''\} \setminus \{C\} \\ C'_1..C'_q & \in \{C_1..C_{\#S} | \{C_1..C_{\#S}\} = S\} \end{array}$$

Figure 6.13: Weaving in pj

Weaving a pj class C, in the context of program P, gives a j class, also called C, that contains all the member definitions declared for C, whether directly or by introduction. The immediate superclass of the resulting class is $\mathsf{super}^{\mathsf{pj}}(P, C)$, and the super-interfaces of the resulting class are the sequence $C'_1..C'_q$: all the interfaces that are supertypes of C in the resultant hierarchy.

6.5 Operational semantics

The semantics of pj is essentially the same as that for ij; the change to the behaviour of programs is that changing a class's parents might change the inherited version of methods that can be invoked upon its instances. Changing parents also changes the potential polymorphism of objects: since a class cannot lose supertypes, no assignments will break as a result of introducing parents, but adding a parent to some class C might increase the set of places in the program where an instances of some other class is expected but where an instance of C may be safely substituted [49].

The runtime values and objects are the same as those defined in figures 4.29 and 4.31.

Heap and stack frame operations are defined in figure 4.33. Function new (figure 4.32) can be used to give correctly initialised new objects.

6.5.1 Reduction rules

The reduction rules for ij, defined in figure figures 4.35, effectively define judgement $P \vdash_{\mathsf{pj}} e, h, s \rightsquigarrow v, h'$ which states that expression e reduces, according to the native pj semantics, in the context of pj program P, to value v and new heap h'. Rules for errors are defined in figures 4.36 and 4.37.

6.5.2 Semantics via weaving

The semantics of pj programs can also be defined in terms of weaving. Weaving, as is defined in figures 5.9 and 6.13, The pj program (P^{pj}) is first woven ($\mathsf{W}(P^{\mathsf{pj}})$), to give a j program, then the execution of the resulting j program will proceed according to the rules in defined in figure 4.35:

$$\mathsf{W}(P^{\mathsf{pj}}) \vdash_{\mathsf{j}} e, h, s \rightsquigarrow v, h'$$

6.5.3 Observations

Parent introduction in ASPECTJ is static: the introductions are declarations that are checked and resolved by the compiler in other languages. It is possible to modify the superclass hierarchy dynamically too:

- In *Fickle* [23] a special expression is allowed to change the superclass of an object.
- In dynamic object based languages like *SELF* [21] there are no classes but the behaviour of an object is defined in part by the other objects to which it delegates method invocations. These delegate paths can be updated dynamically by assigning to the delegate slot.

6.6 Concept List

Term	Meaning	Location
$\mathsf{GetDecs}^{\mathsf{pj}}(P, C)$	All member declarations of class C in program P	figure 5.7
$\mathsf{GetFields}^{\mathsf{pj}}(P, C)$	All fields defined in class C in program P	figure 4.12
$\mathsf{GetMethods}^{\mathsf{pj}}(P, C)$	All methods defined in class C in program P	figure 4.12
$\vdash_{\mathsf{pj}} P \overset{\triangle}{uc}$	Unique class and interface names	figure 4.13
$P \vdash_{\mathsf{pj}} C \overset{\triangle}{ct}$	Class type	figure 4.14
$P \vdash_{\mathsf{pj}} C \overset{\triangle}{it}$	Interface type	figure 4.14
$P \vdash_{\mathsf{pj}} C \overset{\triangle}{ot}$	Valid object type (class or interface name)	figure 4.14
$P \vdash_{\mathsf{pj}} T \overset{\triangle}{vt}$	Type suitable for variable declaration	figure 4.14
$P \vdash_{\mathsf{pj}} C \leq C'$	Subtype (by declaration)	figure 6.10
Γ^{pj}_P	The type hierarchy for program P	figure 6.12
$\vdash \Gamma \overset{\triangle}{po}$	Well formed type hierarchy (top, reflexive, antisymmetric)	figure 4.18
$P \vdash_{\mathsf{pj}} \Gamma \diamond$	Valid hierarchy for program (covers all types, reflexive single class inheritance and separate interface hierarchy)	figure 4.18
$\mathsf{super}^{\mathsf{pj}}(P, C)$	Find unique superclass of C in P	figure 4.19
$\mathsf{AllFields}(P, C)$	Gives sequence of all fields supported by objects of class C	figure 4.20
$\vdash_{\mathsf{pj}} P \overset{\triangle}{SS}$	Structurally sound: unique member names and well formed hierarchy.	figure 4.22
$\mathsf{Fs}^{\mathsf{pj}}(P, C, f)$	Select field f of class C in P	figure 4.23
$\mathsf{Ms}^{\mathsf{pj}}(P, C, m)$	Select field f of class C in P	figure 4.23
$\mathsf{F}^{\mathsf{pj}}(P, C, f)$	Defined or inherited field f in C	figure 4.24
$\mathsf{M}^{\mathsf{pj}}(P, C, m)$	Defined or inherited method m in C	figure 4.24
$\mathsf{H}^{\mathsf{pj}}(P, C, m)$	Defined or inherited header m in C	figure 4.24
$\mathsf{C}_{\mathsf{def}}{}^{\mathsf{pj}}(P, C, id)$	Reverse lookup: class in which id is declared	figure 4.24
$P \vdash \gamma \diamond$	Well formed type contexts	figure 4.26
$P, \gamma \vdash\ e\ : T$	Expression e has type T	figure 4.27
$P, C \vdash_{\mathsf{pj}} D \diamond$	Well formed member declarations (including overloading/shadowing)	figure 4.28
$P, C \vdash T\ m(T') \diamond$	Well formed method headers	figure 4.28
$P \vdash_{\mathsf{pj}} X \diamond$	Well formed classes or interfaces	figure 4.28
$\vdash_{\mathsf{pj}} P \diamond$	well formed programs	figure 4.28
h	Heaps	figure 4.33
s	Stack frames	figure 4.33
$[\![f{:}v \ldots]\!]^C$	Objects	figure 4.31
$\mathsf{new}^{\mathsf{j}}(P, C)$	New object of class C	figure 4.32
$\mathsf{type}(h, v)$	Dynamic type value v	figure 4.34
$P \vdash e, h, s \rightsquigarrow v, h'$	Expression e rewrites in heap h and stack s to value v and modified heap h'	figure 4.35
$\mathsf{weaveC}^{\mathsf{pj}}(P, C)$	Weaving programs in pj	figure 6.13

Chapter 7

A language with advice

7.1 The language aj

aj extends j with support for *before* and *after advice*, that is to say, expressions that can be run before and/or after the occurrence of certain events – known as *join points* – during the execution of a program.

In aj a join point might be a method call, field access or field update. Advice allow expressions to be executed either before or after a join point. An advice declaration has a body – a normal j expression – and refers to a set of join points in the program by means of a *pointcuts* expression. The idea of a pointcuts is that it defines a set of join points that are relevant to a particular concern; it cuts across the space of all method calls, field accesses and field updates, and touches those which relate to the particular concern. In aj, pointcuts are defined with pointcuts expressions which comprise a keyword (one of `call`, `get` or `set`) that nominates the kind of join point, and a signature comprising a set of type patterns used to establish whether a join point matches the pointcuts.

The objects involved include the currently active object (the one executing the code that is calling the method or accessing the field), the receiver of the expression (the object that receives the method call, or that owns the field). Values involved include the value of the field (for field access and update) or the method's parameter (for method calls).

Unlike method bodies, which are explicitly invoked by statements in the program, advice body expressions run implicitly when matching join points occur. Advice body expressions execute in the context of the special variables: `s`, `r`, `v`, `x` and `this`; that are bound to objects and values involved with the join point: `s` gives the currently active object, `r` gives the owner of the field or method being accessed, in advice before field updates `v` gives the value to be assigned to the field (the current value can be observed via `r.`f , where f is the name of the field) and, in method call join points, `x` is bound to the method's parameter. The last of these variables, `this`, is used to give access to a private per-aspect state space by binding to an instance of the class in which the advice

was defined. A choice is required, when advice runs, as to which object should be used as its execution context (i.e. bound to `this` in the body of the advice block)

The chosen solution for aj is as follows. Any class that contains an advice declarations is, by definition, a singleton and it is that single instance which is chosen when advice is executed. To facilitate this in a way that is consistent with ASPECTJ, a new kind of class is introduced, called an *aspect*; only aspects may contain advice declarations; aspects are singletons and thus may not be instantiated in the normal way using `new`. For simplicity, in this model, aspect classes are not allowed to specify their superclasses; all aspect classes implicitly directly extend `Object` [1] though they may implement interfaces like normal classes.

Example

The example program in figure 7.1 demonstrates a simple use of advice. Class `Subject` contains a field `bool flag` and a method `changed(...)` which is assumed to take some action upon changes to the flag. The aspect `A` defines an advice which ensures that the `changed` method is run on a `Subject` object after each update that changes the value of a Subject's `flag` field. The change in value is detected just before the update and the result stored in the aspect instance's field `changed`; the value of this field is checked just after the update and if it's value is `true`, denoting that the value changed, the `changed` method is called on the target object.

This example is written in the syntax of aj (apart from the not-equal operator (`!=`) on boolean values, which is not defined as part of the calculus) defined in figure 7.3 The labels `a1:` and `a2:` before the advice definitions are unique identifiers for the advice. Such identifiers are not required in ASPECTJ, but are included for technical reasons. The pointcuts expression (`set ( * Subject.* )`) denotes all field-update join points wherein any field (of any type) belonging to an instance of `Subject` is being updated. The variable `r` referred to in the body of the advice is a special variable which is bound implicitly to the target object (`r` stands for receiver) of the join point expression, that is to say it refers to the instance of `Subject` whose field has been updated. The `v` variable is another special variable which is defined only in before advice on field updates, it is bound to the value that is about to be assigned to the field. It is used in this example to check whether the update expression will change the value of the field or leave it the same.

The call to `changed(...)` in this example is intended to suggest the subject-observer protocol; if the rest of the protocol were implemented for the `Subject` class, this method might inform all registered observers that the subject's `flag` field has changed state.

Another example

The program in figure 7.2 illustrates use of after-advice on a method call to count the number of times the method has been called. The advice on pointcut `call(* C.foo(*))` runs after all calls to method `foo` and increments the counter variable. Once again a simple extension to aj is assumed: to include the integer type, `int`, with postfix increment operator, `++`, in order to write this example. The aspect class `A` will have a

[1] This would be unacceptable in a model that includes abstract pointcuts since then aspects must be able to extend other aspects in order to make concrete the abstract pointcuts defined therein

```
P1 = class Subject extends Object {
       bool flag

       void changed(Object x){
         ...
       }

     }
     aspect A {
       bool changed;

       a1: before set(* Subject.*){
         this.changed = (v != r.flag);
       }

       a2: after set(* Subject.*){
         if (this.changed){
           r.changed(null)
         } else {}
       }

     }
```

Figure 7.1: After advice on field update to detect state change.

```
P2 = class C extends Object {
       Object foo(Object x){
         this
       }
     }
     aspect A {
       int foocount;
       a: after call(* C.foo(*)) {
         foocount++;
       }
     }
```

Figure 7.2: After advice on method call to count calls.

singleton instance and it is that object whose `foocount` field should be increased after a call to method `foo` on any instance of class `C`. Rules for default initialisation would ensure that the aspect's `foocount` field was initially be set to `0`; Method `foo`, in this example, does nothing. It ignores its (mandatory) parameter `x` and returns `this` so that its body typechecks. In order to test the program there must be an entry point expression, such as:

```
new C.foo(null)
```

After execution of this expression it is expected that the `foocount` field be set to `1` on a special singleton instance of aspect class `A` (instantiation of aspect classes is discussed later in this chapter). ⌟

7.2 Syntax

The syntax of aj is an adaptation of the syntax of j defined in figure 4.4. The differences are defined in figure 7.3; figure 7.4 defines additional definitions used on figure 7.3. A new kind of class called an `aspect` is introduced; aspects implicitly have `Object` as their superclass but may implement interfaces. Aspects may contain advice in addition to normal declarations. Advice are formed from an identifier a, a modifier α (one of `before` or `after`), a pointcuts expression π and a body which is a block containing an expression e.

$P \in Prog^{\mathsf{aj}}$	::=	X^*
$X \in Class^{\mathsf{j}} \cup Aspect \cup Inter$		
$Z \in Class^{\mathsf{j}}$		
$Y \in Aspect^{\mathsf{aj}}$	::=	`aspect` C `[` `imp` C^* `]` `{` D^* A^* `}`
$A \in Advice^{\mathsf{aj}}$	::=	a: α π `{` e `}`
$\pi \in PCE^{\mathsf{aj}}$	::=	`call` `(` Gm `)` |
		`get` `(` Gf `)` |
		`set` `(` Gf `)`
$var \in Var^{\mathsf{aj}}$	::=	`this` | `x` | `s` | `r` | `v`

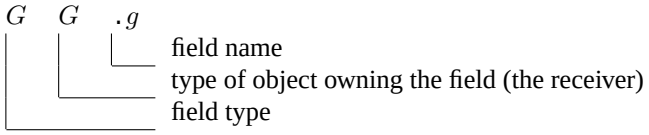
Figure 7.3: Syntax of aj as an extension of j including aspects, advice and pointcuts.

$\alpha \in Mod$	::=	`before` | `after`
$G \in Tpat$	::=	T | `*`
$g \in IDpat$	::=	id | `*`
$Gs \in GSig$	::=	Gm | Gf
$Gm \in GSig_m$	::=	G G`.` g `(` G `)`
$Gf \in GSig_f$	::=	G G`.` g
$a \in Ident$		

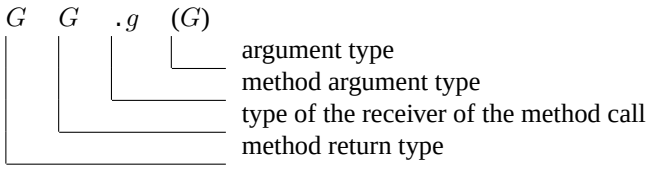
Figure 7.4: Preterms used in the definition of the syntax of aj

Point-cuts

Point-cuts expressions are one of `call`, `get` or `set` followed by a signature made of type patterns (G) or identifier patterns (g). A pattern is either an identifier or a wildcard; identifiers match identical identifiers, wildcards match all identifiers. Each patterns is matched against a value associated with a join point depending upon its position in the signature tuple as indicated below. For `set` and `get` pointcuts the signatures have the following structure:



For `call` pointcuts the signatures have the following structure:



The pointcuts from the two examples at the start of this chapter are `set(* Subject.*)` and `call(* C.foo(*))`

Advice identifiers

Each advice declaration is given an identifier a which is required to be unique within the entire program. Note that this uniqueness requirement is different from the one for field or method names. Fields and methods only need to have unique names within a class (together with the rules about overriding and shadowing). Advice do not have such identifiers in ASPECTJ, they are introduced in aj for technical reasons and not for modelling purposes (Function applicable (from section 7.21) extracts a set of applicable advice in which duplicates would be lost). There is no reason why a programmer should be required to supply these identifiers and to ensure that they are unique. Instead identifiers would be supplied by a simple transformation from a surface syntax in which advice are unnamed.

In examples the symbols a, a', a_n, etc. are used to stand for advice labels.

7.2.1 Functions on the syntax

The functions in figures 7.5 and 7.6 project values from aspect classes and advice respectively.

For an aspect Y of the form: $Y = \texttt{aspect}\ C\{D_1..D_k A_1..A_n\}$ and a class Z:

$$\mathsf{classId}^{\ell}(Y) = C$$
$$\mathsf{superClass}^{\ell}(Y) = \texttt{Object}$$
$$\mathsf{DeclarationsX}^{\ell}(Y) = D_1..D_n$$
$$\mathsf{Advices}^{\ell}(Y) = A_1..A_n \qquad \mathsf{Advices}^{\ell}(Z) = \epsilon$$

Figure 7.5: Functions on aspects in $[\ell]$

For an advice of the form $A = a : \alpha\ \pi\texttt{\{}e\texttt{\}}$:

$$\begin{aligned}\mathsf{advice}(A) &= a\\ \mathsf{pce}(A) &= \pi\\ \mathsf{mod}(A) &= \alpha\end{aligned}$$

Figure 7.6: functions on advice

$$\begin{aligned}\mathsf{advice}^{\ell} &: Prog^{\ell} \longrightarrow Advice^{*}\\ \mathsf{advice}^{\ell}(\epsilon) &= \epsilon\\ \mathsf{advice}^{\ell}(Z :: P) &= \mathsf{advice}^{\ell}(P)\\ \mathsf{advice}^{\ell}(Y :: P) &= \mathsf{Advices}^{\ell}(Y) +\!\!+ \mathsf{advice}^{\ell}(P)\end{aligned}$$

Figure 7.7: All advice in defined in a program in $[\ell]$

A sequence of all the advice declarations in a program P can be obtained with function $\mathsf{advice}^{\mathsf{aj}}(P)$ (defined in figure 7.7). Function Aspects (defined in figure 7.8) filters out the normal classes from a program to give a sequence of the names of all the aspect classes in the program.

$$\begin{aligned}\mathsf{Aspects}^{\ell} &: Prog^{\ell} \longrightarrow Ident^{*}\\ \mathsf{Aspects}^{\ell}(\epsilon) &= \epsilon\\ \mathsf{Aspects}^{\ell}(Z :: P) &= \mathsf{Aspects}^{\ell}(P)\\ \mathsf{Aspects}^{\ell}(Y :: P) &= \mathsf{classId}^{\ell}(Y) :: \mathsf{Aspects}(P)\end{aligned}$$

Figure 7.8: Identifiers of all aspects in a program in $[\ell]$

⌜ In terms of the example given earlier in this chapter, $\mathsf{advice}^{\mathsf{aj}}$ gives the advice defined in all aspects in the program:

$$\begin{aligned}\mathsf{advice}^{\mathsf{aj}}(\texttt{P1}) = &a_1 : \texttt{before set(* Subject.*)\{...\}},\\ &a_2 : \texttt{after set(* Subject.*)\{...\}}\end{aligned}$$

and $\mathsf{Aspects}^{\mathsf{aj}}$ gives the name of the single aspect A:

$$\mathsf{Aspects}^{\mathsf{aj}}(\texttt{P1}) = A$$

⌟

Unique names and lookup

Class names are required to be unique in aj as they are in j. An aspect class is just a special kind of class, so the uniqueness constraint $\vdash_{\mathsf{aj}} P \overset{\triangle}{uc}$ applies across both aspect classes and regular classes.

Also, in aj, advice identifiers must be unique within a program as defined in figure 7.9.

$$[UA]\ \frac{\mathsf{advice}^{\ell}(P) = A_1..A_n \quad \forall\, i,k \in 1..n \bullet \mathsf{advice}(A_i) = \mathsf{advice}(A_k) \Longrightarrow i = k}{\vdash P \overset{\triangle}{ua}}$$

Figure 7.9: Unique advice labels.

For a program P with unique advice ($\vdash P \overset{\triangle}{ua}$) function $\mathsf{owningAspect}^{\mathsf{aj}}(P,a)$ is defined, which gives the name of the aspect class that contains the declaration of advice a. (figure 7.10).

$$\mathsf{owningAspect} : Prog \times Ident \rightarrow\!\!\!\!\!\!+\ \ ClassName$$

$$\mathsf{owningAspect}^{\mathsf{aj}}(P,a) = \begin{cases} C & \text{if } \exists_1 C \bullet \begin{pmatrix} \mathsf{Advices}^{\ell}(P/_{\{C\}}) = A_1..A_n \\ \exists\, i \in 1..n \bullet \mathsf{advice}(A_i) = a \end{pmatrix} \\ Udf & \text{otherwise} \end{cases}$$

Figure 7.10: The name of the aspect class in which named advice was defined

For example in terms of the example program given earlier in this chapter:

$$\mathsf{owningAspect}^{\mathsf{aj}}(\mathtt{P1}, a_1) = A$$

7.2.2 Types

In aj variables are annotated in the same way as in j but new rules are needed for aspect classes. Type judgements $P \vdash_{\mathsf{aj}} C \overset{\triangle}{ct}$, $P \vdash_{\mathsf{aj}} C \overset{\triangle}{ct}$, $P \vdash_{\mathsf{aj}} C \overset{\triangle}{it}$, $P \vdash_{\mathsf{ij}} C \overset{\triangle}{ot}$, $P \vdash_{\mathsf{aj}} T \overset{\triangle}{vt}$ are defined by the rules in figure 4.14 for classes and interfaces, together with the additional one in figure 7.11 for aspects which also introduces the new judgement $P \vdash_{\mathsf{aj}} C \overset{\triangle}{at}$ for aspect types.

$$\frac{\vdash_\ell \; P \overset{\triangle}{uc} \qquad P/_{\{C\}} = \texttt{aspect}\; C \ldots}{P \vdash_\ell \; C \overset{\triangle}{ot}}$$

$$P \vdash_\ell \; C \overset{\triangle}{at}$$

Figure 7.11: Class judgements for a language $[\ell]$ with aspects

7.2.3 Class and type hierarchy

The subtype relation ($P \vdash_{\overline{\mathsf{aj}}} C \leq C'$) for aj is defined in figure 4.15 with the additional rule defined in figure 7.12.

$$[\textit{Aspects}]\ \frac{P \vdash_\ell \; C \overset{\triangle}{at}}{P \vdash_\ell \; C \leq \texttt{Object}}$$

Figure 7.12: Supertypes of aspects

The subtype hierarchy $\Gamma \subseteq (ClassName \times ClassName)$. For an aj program P with unique class names ($\vdash_{\overline{\mathsf{aj}}} \; P \overset{\triangle}{uc}$) is defined in figure 4.16. As before, $C \sqsubseteq C'$ is written in place of $(C, C') \in \Gamma$ when Γ is clear from the context.

A well formed hierarchy ($P \vdash_{\overline{\mathsf{aj}}} \; \Gamma \diamond$) is one with the properties defined in figure 4.18: that it be a partial order with `Object` as the top element and the other elements being the names of classes, interfaces or aspects in the program, each class having a unique superclass and interfaces having only extending other interfaces (aspects implicitly have `Object` as their superclass).

Function $\mathsf{super}^{\mathsf{aj}}$, which gives the superclass of a class, is defined in figure 4.19.

Function AllFields gives a sequence containing of all the fields that should be present in an instance of the named class. It is defined in figure 4.20,

7.2.4 Soundness

A program is structurally sound (judgement $\vdash_{\mathsf{J}} \; P \overset{\triangle}{SS}$) if its type hierarchy is well formed and its classes and interfaces have unique member names (rule [SS] in figure 4.22).

7.2.5 Method and field lookup

Functions $\mathsf{Fs}^{\mathsf{pj}}(P, C, f)$ and $\mathsf{Ms}^{\mathsf{aj}}(P, C, m)$ look up the definition of named fields and methods in a named class in a program. They are defined in figure 4.23. Functions

$\mathsf{F}^{\mathsf{j}}(P, C, f)$ and $\mathsf{M}^{\mathsf{j}}(P, C, m)$ do the same but allowing for inheritance. They are defined in figure 4.24.

Member definitions are inherited by subclasses. The class in which a named member was defined might be a superclass of the class of an object owning that member. The defining class can be obtained using function $\mathsf{C}_{\mathsf{def}}$defined in figure 7.13.

$$\mathsf{C}^{\ell}_{\mathsf{def}}(P, C, id) = \begin{cases} Udf & \text{if } C = \texttt{Object} \\ C & \text{if } \#\left(\mathsf{GetDecs}^{\ell}(P, C)/_{\{id\}}\right) = 1 \\ \mathsf{C}^{\ell}_{\mathsf{def}}(P, C', id) & \text{otherwise} \end{cases}$$
$$\text{where } C' = \quad \mathsf{super}^{\ell}(P, C)$$

Figure 7.13: Defining class for a feature id defined or inherited by class C

7.3 Well formedness in aj

7.3.1 Type checking

Environment

In aj, as in j, type checking is performed in the context of an environment γ that maps selected identifiers to their types. In aj, identifiers are $Var^{\mathsf{aj}} = \{\texttt{this}, \texttt{x}, \texttt{s}, \texttt{r}, \texttt{v}\}$ Like `this`, the sender (`s`) and receiver (`r`) always have object types. Like the argument (`x`) of a method call, the new value (`v`) of a field may have any variable type.

Not all these variables are always available, for example: advice bodies have a sender and receiver but method bodies do not; method bodies and advice on method calls have an argument, but other kinds of advice bodies do not. Operations on γ^{aj} are defined in

$$\gamma : (\{\texttt{this}, \texttt{s}, \texttt{r}\} \longrightarrow (Ident \cup \{Udf\})) \cup (\{x, \texttt{v}\} \longrightarrow (Type \cup \{Udf\}))$$

Figure 7.14: Type environment for aj

figure 4.25.

Type rules

The rules in figure 7.15 define judgements $P, C \vdash A \diamond$ and $P, C \vdash A_1..A_n \diamond$ for a well typed advice. Rule [*WFFA*] says that a field access advice (on set or get join points)

is well formed if the advice body expression e has some type T in the context of its enclosing program P and a stack frame in which the special variable `s`, is bound to `Object` (since field advice places no constraint on the type of the object 'sending' a field access, the most general type must be assumed), `r` is bound to the type T_r of the receiver of the field access (the type of the object that owns the field being accessed), and `this` is bound to the type C of the aspect in which the advice is defined. Rule [*WFCA*] says that method call advice (on call join points) is well formed if the advice body expression e has some type T in the context of its enclosing program P a stack frame in which the special variable `s` is bound to `Object` (as above), `r` is bound to the type T_r of the object that owns the method being called, `x` is bound to type type T_x of the formal parameter of the method and `this` is bound to the type of the aspect owning the advice. Rule [*WFA*] says that a sequence of advice is well formed in the context of a program P and the name of a class (the aspect class in which they are defined) if each of the advice in that sequence are well formed for the same program and class.

$$[\textit{WFFA}]\ \frac{\beta \in \{\texttt{set}, \texttt{get}\} \quad P, \{\texttt{s} \mapsto \texttt{Object}, \texttt{r} \mapsto T_r, \texttt{this} \mapsto C\} \vdash\ e\ : T}{P, C \vdash a : \alpha\ \beta(T_f\ T_r.f)\{e\} \diamond}$$

$$[\textit{WFCA}]\ \frac{P, \{\texttt{s} \mapsto \texttt{Object}, \texttt{r} \mapsto T_r, \texttt{x} \mapsto T_x, \texttt{this} \mapsto C\} \vdash\ e\ : T}{P, C \vdash a : \alpha\ \texttt{call}(T'\ T_r.m(T_x))\{e\} \diamond}$$

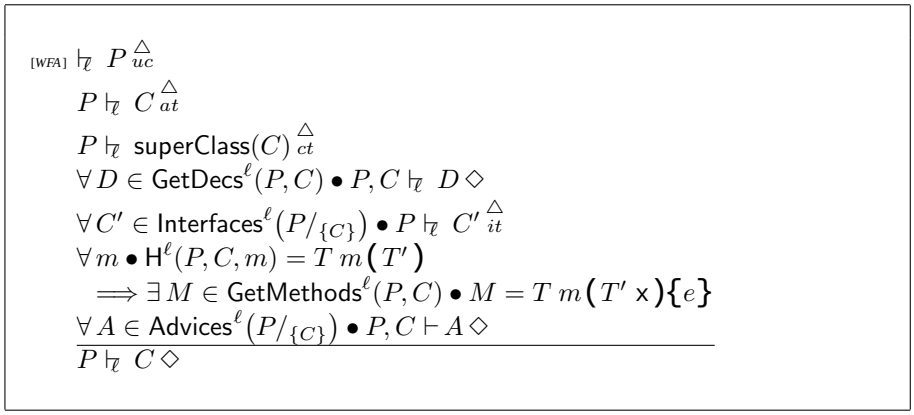
$$[\textit{WFA}]\ \frac{\forall\, i \in 1..n \bullet P, C \vdash A_i \diamond}{P, C \vdash A_1..A_n \diamond}$$

Figure 7.15: Type inference for expressions in $[\ell]$

7.3.2 Well-formed Programs

In order for an aj program to be well typed, the bodies of all methods and advice must type-check. The rules for well typed expressions remain as defined in figure 4.27. There is an apparently additional requirement that aspect classes, being implicitly singletons, are not instantiated via `new` expressions. This requirement is similar to the situation with interfaces: instantiating interfaces is not permitted either. Both of these conditions are, in fact, handled by the rule for `new` expressions in figure 4.27, which requires $P \vdash_{\mathsf{aj}}\ C \overset{\triangle}{ct}$ for `new` C to have a type.

Figure 7.16 defines a new rule that defines well formedness $P \vdash_{\ell}\ C \diamond$ for aspects. It is the same as rule [*WFC*] for classes in figure 4.28 except for the additional final line: $\forall\, A \in \mathsf{Advices}^{\ell}(P/_{\{C\}}) \bullet P, C \vdash A \diamond$ which tests that the advice defined in the aspect are well typed.



$$
\begin{array}{l}
{\scriptstyle [WFA]} \vdash_\ell \; P \overset{\triangle}{_{uc}} \\
\quad P \vdash_\ell \; C \overset{\triangle}{_{at}} \\
\quad P \vdash_\ell \; \mathsf{superClass}(C) \overset{\triangle}{_{ct}} \\
\quad \forall\, D \in \mathsf{GetDecs}^\ell(P, C) \bullet P, C \vdash_\ell \; D \diamond \\
\quad \forall\, C' \in \mathsf{Interfaces}^\ell\big(P/_{\{C\}}\big) \bullet P \vdash_\ell \; C' \overset{\triangle}{_{it}} \\
\quad \forall\, m \bullet \mathsf{H}^\ell(P, C, m) = T\; m\big(T'\big) \\
\qquad \Longrightarrow \exists\, M \in \mathsf{GetMethods}^\ell(P, C) \bullet M = T\; m\big(T'\; \mathsf{x}\big)\{e\} \\
\quad \forall\, A \in \mathsf{Advices}^\ell\big(P/_{\{C\}}\big) \bullet P, C \vdash A \diamond \\
\hline
P \vdash_\ell \; C \diamond
\end{array}
$$

Figure 7.16: Well formed programs in languages $[\ell]$ with aspect.

7.3.3 Join points and pointcuts

Join points

Join points are events in the execution of a program around which advice bodies can be executed. In aj a join point is the execution of one of the three kinds of member access expression: field access, field update and method call. Join points are modelled with values of type JP (defined in figure 7.17) which comprise a designator (`get`, `set` or `call`) and the signature of the a member being accessed.

$S \in Sig$	::=	$Sm \mid Sf$
$Sm \in Sig_m$	::=	$T\ C$`.` m `(` T `)`
$Sf \in Sig_f$	::=	$T\ C$`.` f
$jp \in JP$	::=	`(get` $\mid$ `set)(` Sf`)` $\mid$ `call(` Sm`)`

Figure 7.17: Signatures and join points.

Point-cuts

In order to decide what advice, if any, should run before and after a potential join point expression, the actual join point value is compared with the pointcuts expressions from all advice in the program. Point-cuts expressions also comprise a designator and a signature but unlike a join point signature, any of the types in a pointcuts may be replaced by a wildcard.

The patterns are either generic type names G or generic identifiers g; a pattern may be either an identifier of the appropriate kind or a wildcard `*`(see the syntax definition in figures 7.3 and 7.4).

In the example program `P1`, the pointcuts expression (of both advice definitions in the aspect `A`) is: `set ( * Subject.* )`. In program `P2` the pointcuts used is: `call ( * C.foo ( * ) )`

7.3.4 Pattern matching

Patterns are expressions that express constraints over set of identifiers or class names. Syntactically a pattern is either an identifier (or class name) or an asterisk denoting a wildcard. In ASPECTJ patterns may be more general regular expressions, modelling the regular expression matching is not vital to an understanding of the semantics of advice so in pj patterns match actual types or identifiers according to these simple rules: a wildcard matches any identifier or class name; an identifier or class name matches another if they are equal.

7.3.5 Advice order

It is possible for more than one advice to be defined for the same join point in a program. In this case when the join point occurs, all applicable advice bodies should execute. But in what order should they execute? The rules for ordering the execution of advice bodies in aj, follows those of ASPECTJ: to determine the order relation between two advice, A and A', consider the following:

- If A and A' are defined in the same aspect class then the order in which they are defined in that aspect class.
- If A and A' are defined in different aspect classes then their order is undefined.

ASPECTJ has some additional rules that make it easier for programmers to predict the behaviour of their programs in the case where the advice are defined in different aspects:

- If there is an inheritance relationship between the classes in which A and A' are defined, then the advice defined in the subclass precedes that defined in the superclass.
- When the classes in which A and A' are unrelated by inheritance, but the programmer requires a specific ordering on the execution of advice, an additional relation called *dominates* may be used. Dominates relationships are introduced by the `dominates` keyword, used in class headers in a similar way to `extends` and `implements` to indicate that the class in question dominates some other class. Then advice defined in a dominating class precedes that in a dominated class.

Aspect classes in aj do not participate in inheritance relationships so inheritance does not affect advice order in aj. The dominates relationship has been excluded from the model for reasons of simplicity. It would be appropriate to include it in a model that included inheritance among aspect classes.

The order described above is used to determine the order of execution for advice running before a join point. For advice running after a join point the order is precisely the opposite of that described above.

In aj ordering among advice is defined only if those advice are defined in the same class, in which case it is the lexical order of the advice declarations. This means that

in aj, as in ASPECTJ, it may happen that the execution order of advice defined on the same join point, but in separate classes is undefined. To reflect this, aj is deliberately non-deterministic in such cases. In general, there may be many advice on a join point, and some combinations of those may have been defined in the same aspect class and thus there may be constraints on which sequences of applicable advice are valid. The mechanism for selecting advice for a join point produces the set of all valid sequences of applicable advice, and one of those valid sequences is selected, nondeterministically, for execution.

For example, the program in figure 7.18 three advice, `a1`, `a2` and `b1`, are defined on the same join point. Since `a1` and `a2` are defined in the same aspect class they must be executed in the order `a1` then `a2`. But `b1` is also applicable and so when that join point occurs, all three advice must run in some valid order. Since `b1` is defined in a different aspect class from `a1` and `a2` there are no order constraints among `a1` and `b1` or between `a2` and `b1`, hence the allowable sequences of advice to execute before a call to `C.foo(...)` are: $\{(\texttt{a1}, \texttt{a2}, \texttt{b1}), (\texttt{a1}, \texttt{b1}, \texttt{a2}), (\texttt{b1}, \texttt{a1}, \texttt{a2})\}$. Any one of these sequences of advice may be chosen and executed when the join point happens.

```
class C ext Object {
Object foo(Object x){}
}
aspect A {
  a1: before call(Object C.foo(Object)){
     ... body a1 ...
  }
  a2: before call(* *.*(*)){
     ... body a2 ...
  }
}
aspect B {
  b1: before call(* C.foo(*)){
     ... body b1 ...
  }
}
```

Figure 7.18: Three advice on the same join point.

Ordering advice

The ordering for advice in aj is defined as follows: If A and $A'$are advice declarations in program P then, as defined in figure 7.19, A precedes A' ($A \underset{P}{\leq} A[']$) if A and A' are defined in the same aspect class and the declaration of A precedes that of A'.

A sequence of advice declarations $A_1..A_n$ is well ordered for a program P ($P \vdash A_1..A_n \overset{\triangle}{}_{ord}$) if the sequence respects the advice order for P as defined in figure 7.19.

$$[AB4]\ \frac{\vdash P \overset{\triangle}{ua} \quad \exists C \bullet \mathsf{advice}^{\ell}\left(P/_{\{C\}}\right) = A_1..A_n \quad 1 \le i < j \le n}{A_i \underset{P}{\le} A_j} \qquad [AORD]\ \frac{\forall i, j \in 1..n \bullet A_i \underset{P}{\le} A_j \Longrightarrow i < j}{P \vdash A_1..A_n \overset{\triangle}{ord}}$$

Figure 7.19: Partial order on advice

7.3.6 Selecting advice

Coverage

The $\sqsubset$ relation:

$$\sqsubset \subseteq (\{\texttt{call}, \texttt{set}, \texttt{get}\} \times Sig) \times PCD$$

represents the coverage of join points by pointcuts. $\sqsubset$ is defined to be the minimal relation that satisfies the rules in figure 7.20.

$$[Mwild]\ \frac{}{id \sqsubset \texttt{*}} \qquad [Mequal]\ \frac{}{id \sqsubset id}$$

$$\frac{T \sqsubset \texttt{*}}{T \sqsubset G} \qquad \frac{T \sqsubset T}{T \sqsubset G}$$

$$\frac{T \sqsubset G \quad C \sqsubset G_o \quad m \sqsubset g}{\texttt{get}(T\ C\texttt{.}f) \sqsubset \texttt{get}(G\ G_o\texttt{.}g)} \qquad \frac{T \sqsubset G \quad C \sqsubset G_o \quad m \sqsubset g}{\texttt{set}(T\ C\texttt{.}f) \sqsubset \texttt{set}(G\ G_o\texttt{.}g)}$$

$$\frac{T \sqsubset G \quad C \sqsubset G_o \quad m \sqsubset g \quad T_x \sqsubset G_x}{\texttt{call}(T\ C\texttt{.}m(T_x)) \sqsubset \texttt{call}(G\ G_o\texttt{.}g(G_x))}$$

Figure 7.20: Rules for coverage of join points by basic pointcuts

For example, the pointcuts

$$\texttt{set(bool Subject.flag)} \sqsubset \texttt{set(* Subject.*)}$$

since: `bool` $\sqsubset$ `*` by [*Mwild*], `Subject` $\sqsubset$ `Subject` by [*Mequal*] and `flag` $\sqsubset$ `*` by [*Mwild*].

Applicability

An advice A is applicable before or after a join point jp if the advice's pointcuts covers jp and if A has the given `before` or `after` modifier. Function applicable (defined in figure 7.21) returns the set of all sequences of advice that contain exactly all the applicable advice for the given join point,

$$\mathsf{applicable}^{\ell} : Prog^{\ell} \times Mod \times JP \rightarrow \wp(Advice)$$
$$\mathsf{applicable}^{\ell}(P, \alpha, jp) = \{A \in \mathsf{advice}(P) \mid jp \sqsubset \mathsf{pce}(A), \mathsf{mod}(A) = \alpha\}$$

Figure 7.21: Advice applicable to the given pointcuts

For example, consider the example program, `P1`, given earlier in this chapter. Let:

$jp =$ `set(bool Subject.flag)`

$\pi =$ `set(* Subject.*)`

$A_1 =$
```
a1 : before π{
        this.changed :=( v != r.flag )
}
```

$A_2 =$
```
a2 : after π{
     if ( this.changed ){ r.changed(null) }else { }
}
```

then $\mathsf{applicable}^{\mathsf{aj}}(\texttt{P1}, \texttt{before}, jp) = \{A_1\}$ since $\mathsf{advice}(\texttt{P1}) = A_1\ A_2$ and $\mathsf{pce}(A_1) = \pi$; as shown earlier, $jp \sqsubset \pi$ and $\mathsf{mod}(A_1) =$ `before`, whereas $\mathsf{pce}(A_2) = \pi$ also, but $\mathsf{mod}(A_2) =$ `after`. ⌟

Choosing advice bodies

The function advices takes a program (P), a modifier (α, one of `before` or `after`) and a join-point (jp), and returns a set of sequences of advice declarations. The returned set of sequences contains all permutations of applicable advice for the specified join point.

$\mathsf{applicable}(P, \alpha, jp)$ gives the set S of applicable advice for jp and α, S is equal to a set: $S = \{A_1..A_{\#S}\}$ formed from the elements of a sequence with no duplicate elements. The set comprehension that forms the body of the definition of advices gives all permutations of the members of S that satisfy the order constraint: $P \vdash A_1..A_{\#S} \overset{\triangle}{ord}$.

In the example program `P1`, for the join point $jp =$ `set(bool Subject.flag)` applicable returns the set $\{A_1\}$, as described earlier, since the returned set is a singleton,

$$\mathsf{advices}^{\ell} : Prog^{\mathsf{aj}} \times Mod \times JP \longrightarrow \wp(\ Exp^{*})$$
$$\mathsf{advices}^{\ell}(P, \alpha, jp) = \{A_1..A_{\#S} \mid \mathsf{applicable}^{\ell}(P, \alpha, jp) = S,$$
$$S = \{A_1..A_{\#S}\},$$
$$P \vdash A_1..A_{\#S} \overset{\triangle}{}_{ord}\}$$

Figure 7.22: The set of all sequences of advice that are applicable to the given join point

only the singleton sequence satisfies the order constraints and therefore the result is:

$$\mathsf{advices}^{\mathsf{aj}}(\texttt{P1}, \texttt{before}, jp) = \{\langle A_1 \rangle\}.$$

the set containing the singleton sequence $\langle A_1 \rangle$ (sequence brackets added for clarity).

⌟

7.4 Operational semantics

7.4.1 Runtime values

The semantics of aj uses the same runtime values as j; addresses, values, deviations and objects are defined in figure 4.29.

7.4.2 Objects

Objects in aj are the same as in j: $o = [\![f_1{:}v_1..f_n{:}v_n]\!]^C$. a list of field bindings, and a class name. The syntax of objects is defined in figure 4.29, object field select ($o(f)$) and update ($o[f \mapsto v]$) are defined in figure 4.31, $\mathsf{new}^{\mathsf{j}}$ is defined in figure 4.32.

7.4.3 The store: heaps and stack frames

Stores in aj comprise a heap (h) and a stack frame (s), as in j, but their types are slightly enhanced: Heaps are extended to include mappings from class names to addresses, so that singleton aspect instances can be looked up by their class name. The domain of stack frames is extended to include the variables defined to be in scope during the execution of advice bodies. The definition of heap update ($h\ [\ \iota \mapsto\ o\ ]$) remains as in figure 4.33, note that update to the mapping of aspect class names to addresses is not permitted, this is appropriate as the singleton aspect instances remain in memory throughout execution of the program. The definition of stack composition ($s \oplus s$') also remains as in figure 4.33.

$$h \in Heap = (Addr \longrightarrow Object) \cup (ClassName \longrightarrow Addr)$$
$$s \in Stack = \{\mathtt{this}, \mathtt{x}, \mathtt{s}, \mathtt{r}, \mathtt{v}\} \rightharpoonup Val$$

Figure 7.23: Stack frames and heaps for aj

7.4.4 Aspect instances

The expression that forms the body of advice a should, when it runs, execute in the context of an instance of a's owning aspect: an instance of the aspect class in which a is defined. The rule for aspect instantiation is that each aspect has a singleton instance throughout the execution of a program. To model this aj programs must be executed in a *prepared heap* rather than an empty one as is the case for programs in j. A prepared heap, like a prepared piano[14], is one into which certain objects have been inserted. Specifically, the prepared heap h_p for program P, contains singleton instances of all aspect classes in P, indexed by their class names. A prepared heap for a program is defined in figure 7.24.

A prepared heap for program P (denoted h_P) is defined as:

$$\begin{aligned} h_P = \{&C_1 \mapsto \iota_1, ..., C_n \mapsto \iota_n, \\ &\iota_1 \mapsto \mathsf{new}(P, C_1), ..., \iota_n \mapsto \mathsf{new}(P, C_n)\} \end{aligned}$$

where

$\mathsf{Aspects}(P) = C_1..C_n$

ι_1 ... ι_n are unique

Figure 7.24: Prepared heap

The prepared heap for the example program `P1` would be as defined below:

$$\begin{aligned} h_{\mathtt{P1}} &= \{\mathtt{A} \mapsto \iota_1, \iota_1 \mapsto \mathsf{new}(\mathtt{P1}, \mathtt{A})\} \\ &= \{\mathtt{A} \mapsto \iota_1, \iota_1 \mapsto [\![\mathtt{changed:false}]\!]^{\mathtt{A}}\} \end{aligned}$$

for some ι_1 .

7.4.5 Reduction rules

Reduction rules for normal execution

The rewrite relation has the following type

$$\rightsquigarrow : Prog^{\mathsf{aj}} \longrightarrow \left(Exp \cup Advice^{\mathsf{aj}}\right) \times Heap \times Stack \longrightarrow ((Val \cup Deviation) \times Heap)$$

The new rule [RWAD], introduced in figure 7.25), defines execution semantics for advice $A = a : \alpha\ \pi\{e\}$. Execution is defined for a whole advice construct, rather than just an advice body expression e, because the advice identifier a is needed to find the owning aspect class in order to determine the aspect instance in whose context the body should be executed. Function $\mathsf{owningAspect}^{\mathsf{aj}}(P, a)$ gives the name of the aspect class to which the advice belongs, in the prepared heap this aspect class name maps to the address of the singleton instance of that class. This address is bound to `this` in the stack frame for executing the advice body so that `this` refers to the appropriate enclosing object.

$$[\textit{RWAD}]\ \frac{P \vdash e, h, s \oplus \{\texttt{this} \mapsto h(\mathsf{owningAspect}^{\ell}(P, a))\} \rightsquigarrow v, h'}{P \vdash a : \alpha\ \pi\{e\}, h, s \rightsquigarrow v, h'}$$

Figure 7.25: Execution of an advice body as an expression

Figure figure 7.26 presents the aj versions of rules [*RWGET*], [*RWSET*] and [*RWMETH*] for field access, field update and method call expressions. These are the three ways to access a member (method or field) of an object, they are also the expressions that form join points in aj. In each rule, the applicable advice is collected and executed before, and after, execution of the join point expression. In each rule, the target object (designated by an expression e) and the member name (an identifier, m or f, following a dot) are given. Finding the applicable advice requires the member's name, defining class and signature. This information is inferred from what is available at the execution context.

Recall from the definition of coverage in figure 7.20 that the name of a class, member or return type is covered by a pattern in a join point if and only if either the pattern and the name are identical, or the pattern is a wildcard. In all these rewrite rules, the target expression is evaluated first to give an address ι. The type of this address is obtained using function $\mathsf{type}(P, \iota)$. This gives the target object's dynamic class C_t. Due to polymorphism, this might be a subclass of the class in which the member defined. $\mathsf{C}^{\mathsf{aj}}_{\mathsf{def}}(P, C_t, id)$ is used to give the defining class C_d. The defining class, together with the member name, is used to look up the member definition and obtain the signature. This information is used to form a join point designator to use as the argument of $\mathsf{advices}^{\mathsf{aj}}$ which gives the set of all valid sequences of applicable advice to execute (either before or after, depending on another argument of $\mathsf{advice}^{\mathsf{aj}}$) the join point. Any member of this set, i.e. any valid sequence of applicable advice may be executed.

Rule [*RWGET*] reduces the target expression e to give target object reference, ι. Field f is looked up in the target object to find the result, v of the whole expression. A joinpoint for the field access is constructed from the name, signature and defining class of the field. This join point is used to look up valid sequences of advice applicable before (and after) the field access. If no advice is applicable, the empty sequence (ϵ) will be the only member of this set. Otherwise the set will contain all permutations of sequences of advice where all advice in the sequence are applicable and the sequence itself satisfies the ordering constraints. There might be only one such sequence but there might be several. One advice sequence, from this set, must be executed before,

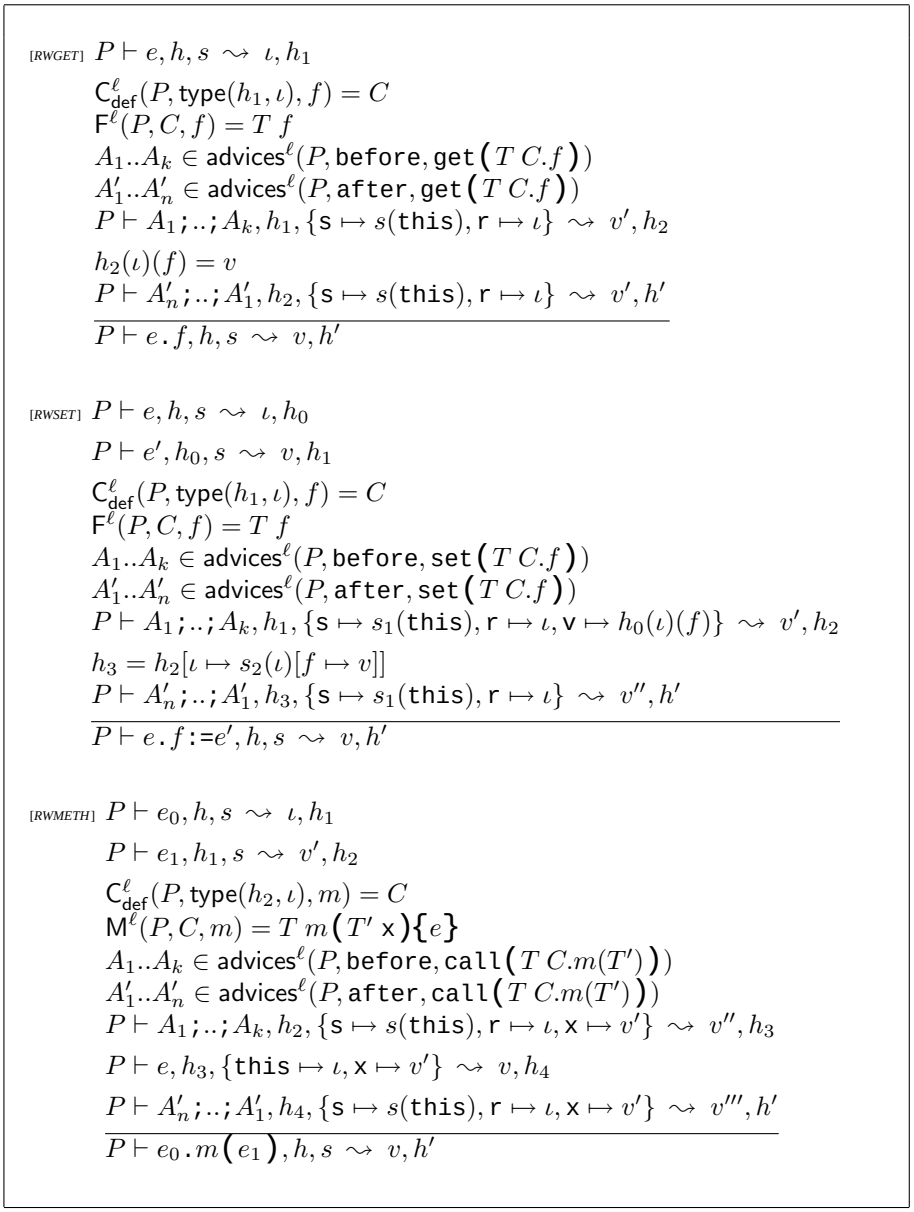


Figure 7.26: Modified rewrite rules for $[\ell]$

and one after, the field access is executed. The semantics of aj are non-deterministic with respect to this choice.

Advice are executed in a stack frame in which the special variables s and r hold references to the active object and the object being accessed respectively. The value of the field being accessed is available in both before and after advice as r.f .

Rule [*RWSET*] for aj is similar to the one for j except that:

- heap h is updated with the new value of the updated field f in object o between execution of the before advice and the after advice
- before advice bodies are executed in a stack frame containing an additional variable `v`, which holds the value to be assigned to the field

Rule [*RWMETH*] reduces the target expression e_0 to a reference ι, and the actual parameter expression e_1 to a value v'. It constructs a join point defined from the type C_s of the executing active object, the type C_t of the receiver and the signature of the method. This join point is used to select the set of sequences of applicable advice before and after the method call. One of the valid sequences applicable before is executed, then the method body expression e, and then one of the valid sequence applicable after advice. Note that the target expression and actual argument are evaluated before the `before` advice so the advice bodies cannot affect the evaluation of these expressions. The method body is executed in a context of a stack frame that binds variables `this` and `x` to a reference to the target object, and the actual parameter value, respectively. The advice bodies are executed with a stack frame that binds `s`, `r` and `x` to the calling object, the target object, and the actual parameter value, respectively.

Each of these three rules uses function `advices` (figure 7.22) which returns a set containing all sequences of applicable advice (for the specified join point) that satisfy the advice order conditions in the program. When more than one advice are applicable there are likely to be many such sequences; selection of a sequence from the set is non deterministic.

Reduction rules for getting stuck

In the event that no owning aspect instance can be found in which to execute an advice body, execution may not proceed. Such a failure is not a property of the program, however, but of the heap. If the heap is prepared according to the definition in figure 7.24, this eventuality should not arise.

Reduction rules for errors arising from dereferencing null pointers

In this model, the type of the target object is used decide what member is being invoked, and therefore what advice is applicable before and after it. In order to achieve this, the rules in figure 7.26 first reduce the target object sub-expression to give an address ι; then obtain the dynamic type of the object at that address ($\mathsf{type}(h_1, \iota)$), which is used to determine the defining class of the member being accessed. If the target object expression evaluates to a null pointer, there is no object of which to find the type; consequently, the applicable advice cannot be determined. The result of reducing a member access whose target object is null is, of course, a null pointer error; execution will not continue. In fact the program execution will halt without executing any advice which ought to be run before the member access, since the advice cannot be chosen. The rules for null pointer errors, therefore, are the same in aj as they are in j, as defined in figure 4.37.

$$\text{[DVBGET]}\quad \frac{P \vdash e, h, s \rightsquigarrow dv, h'}{P \vdash e.f, h, s \rightsquigarrow dv, h'}$$

$$\text{[DVBGET]}\quad \frac{\begin{array}{l} P \vdash e, h, s \rightsquigarrow \iota, h_1 \\ \mathsf{C}^{\ell}_{\mathsf{def}}(P, \mathsf{type}(h_1, \iota), f) = C \\ \mathsf{F}^{\ell}(P, C, f) = T\ f \\ A_1..A_k \in \mathsf{advices}^{\ell}(P, \texttt{before}, \texttt{get}(T\ C.f)) \\ P \vdash A_1\texttt{;..;}A_k, h_1, \{\mathsf{s} \mapsto s(\texttt{this}), \mathsf{r} \mapsto \iota\} \rightsquigarrow dv, h_2 \end{array}}{P \vdash e.f, h, s \rightsquigarrow dv, h_2}$$

$$\text{[DVAGET]}\quad \frac{\begin{array}{l} P \vdash e, h, s \rightsquigarrow \iota, h_1 \\ \mathsf{C}^{\ell}_{\mathsf{def}}(P, \mathsf{type}(h_1, \iota), f) = C \\ \mathsf{F}^{\ell}(P, C, f) = T\ f \\ A_1..A_k \in \mathsf{advices}^{\ell}(P, \texttt{before}, \texttt{get}(T\ C.f)) \\ A'_1..A'_n \in \mathsf{advices}^{\ell}(P, \texttt{after}, \texttt{get}(T\ C.f)) \\ P \vdash A_1\texttt{;..;}A_k, h_1, \{\mathsf{s} \mapsto s(\texttt{this}), \mathsf{r} \mapsto \iota\} \rightsquigarrow v', h_2 \\ h_2(\iota)(f) = v \\ P \vdash A'_n\texttt{;..;}A'_1, h_2, \{\mathsf{s} \mapsto s(\texttt{this}), \mathsf{r} \mapsto \iota\} \rightsquigarrow dv, h' \end{array}}{P \vdash e.f, h, s \rightsquigarrow dv, h'}$$

$$\text{[DVBSET]}\quad \frac{\begin{array}{l} P \vdash e, h, s \rightsquigarrow \iota, h_0 \\ P \vdash e', h_0, s \rightsquigarrow v, h_1 \\ \mathsf{C}^{\ell}_{\mathsf{def}}(P, \mathsf{type}(h_1, \iota), f) = C \\ \mathsf{F}^{\ell}(P, C, f) = T\ f \\ A_1..A_k \in \mathsf{advices}^{\ell}(P, \texttt{before}, \texttt{set}(T\ C.f)) \\ P \vdash A_1\texttt{;..;}A_k, h_1, \{\mathsf{s} \mapsto s_1(\texttt{this}), \mathsf{r} \mapsto \iota, \mathsf{v} \mapsto h_0(\iota)(f)\} \rightsquigarrow dv, h_2 \end{array}}{P \vdash e.f\texttt{:=}e', h, s \rightsquigarrow dv, h_2}$$

$$\text{[DVASET]}\quad \frac{\begin{array}{l} P \vdash e, h, s \rightsquigarrow \iota, h_0 \\ P \vdash e', h_0, s \rightsquigarrow v, h_1 \\ \mathsf{C}^{\ell}_{\mathsf{def}}(P, \mathsf{type}(h_1, \iota), f) = C \\ \mathsf{F}^{\ell}(P, C, f) = T\ f \\ A_1..A_k \in \mathsf{advices}^{\ell}(P, \texttt{before}, \texttt{set}(T\ C.f)) \\ A'_1..A'_n \in \mathsf{advices}^{\ell}(P, \texttt{after}, \texttt{set}(T\ C.f)) \\ P \vdash A_1\texttt{;..;}A_k, h_1, \{\mathsf{s} \mapsto s_1(\texttt{this}), \mathsf{r} \mapsto \iota, \mathsf{v} \mapsto h_0(\iota)(f)\} \rightsquigarrow v', h_2 \\ h_3 = h_2[\iota \mapsto s_2(\iota)[f \mapsto v]] \\ P \vdash A'_n\texttt{;..;}A'_1, h_3, \{\mathsf{s} \mapsto s_1(\texttt{this}), \mathsf{r} \mapsto \iota\} \rightsquigarrow dv, h' \end{array}}{P \vdash e.f\texttt{:=}e', h, s \rightsquigarrow dv, h'}$$

Figure 7.27: Rules for propagation of deviations occurring in advice before or after field access or update

[DVBMETH]
$$\frac{\begin{array}{l} P \vdash e_0, h, s \rightsquigarrow \iota, h_1 \\ P \vdash e_1, h_1, s \rightsquigarrow v', h_2 \\ \mathsf{C}^{\ell}_{\mathsf{def}}(P, \mathsf{type}(h_2, \iota), m) = C \\ \mathsf{M}^{\ell}(P, C, m) = T\ m\texttt{(}T'\ \mathsf{x}\texttt{)\{}e\texttt{\}} \\ A_1..A_k \in \mathsf{advices}^{\ell}(P, \texttt{before}, \texttt{call(}\ T\ C.m(T')\ \texttt{)}) \\ P \vdash A_1\texttt{;..;}A_k, h_2, \{\mathsf{s} \mapsto s(\texttt{this}), \mathsf{r} \mapsto \iota, \mathsf{x} \mapsto v'\} \rightsquigarrow dv, h_3 \end{array}}{P \vdash e_0\texttt{.}m\texttt{(}e_1\texttt{)}, h, s \rightsquigarrow dv, h_3}$$

[DVMETH]
$$\frac{\begin{array}{l} P \vdash e_0, h, s \rightsquigarrow \iota, h_1 \\ P \vdash e_1, h_1, s \rightsquigarrow v', h_2 \\ \mathsf{C}^{\ell}_{\mathsf{def}}(P, \mathsf{type}(h_2, \iota), m) = C \\ \mathsf{M}^{\ell}(P, C, m) = T\ m\texttt{(}T'\ \mathsf{x}\texttt{)\{}e\texttt{\}} \\ A_1..A_k \in \mathsf{advices}^{\ell}(P, \texttt{before}, \texttt{call(}\ T\ C.m(T')\ \texttt{)}) \\ P \vdash A_1\texttt{;..;}A_k, h_2, \{\mathsf{s} \mapsto s(\texttt{this}), \mathsf{r} \mapsto \iota, \mathsf{x} \mapsto v'\} \rightsquigarrow v'', h_3 \\ P \vdash e, h_3, \{\texttt{this} \mapsto \iota, \mathsf{x} \mapsto v'\} \rightsquigarrow dv, h_4 \end{array}}{P \vdash e_0\texttt{.}m\texttt{(}e_1\texttt{)}, h, s \rightsquigarrow dv, h_4}$$

[DVAMETH]
$$\frac{\begin{array}{l} P \vdash e_0, h, s \rightsquigarrow \iota, h_1 \\ P \vdash e_1, h_1, s \rightsquigarrow v', h_2 \\ \mathsf{C}^{\ell}_{\mathsf{def}}(P, \mathsf{type}(h_2, \iota), m) = C \\ \mathsf{M}^{\ell}(P, C, m) = T\ m\texttt{(}T'\ \mathsf{x}\texttt{)\{}e\texttt{\}} \\ A_1..A_k \in \mathsf{advices}^{\ell}(P, \texttt{before}, \texttt{call(}\ T\ C.m(T')\ \texttt{)}) \\ A'_1..A'_n \in \mathsf{advices}^{\ell}(P, \texttt{after}, \texttt{call(}\ T\ C.m(T')\ \texttt{)}) \\ P \vdash A_1\texttt{;..;}A_k, h_2, \{\mathsf{s} \mapsto s(\texttt{this}), \mathsf{r} \mapsto \iota, \mathsf{x} \mapsto v'\} \rightsquigarrow v'', h_3 \\ P \vdash e, h_3, \{\texttt{this} \mapsto \iota, \mathsf{x} \mapsto v'\} \rightsquigarrow v, h_4 \\ P \vdash A'_n\texttt{;..;}A'_1, h_4, \{\mathsf{s} \mapsto s(\texttt{this}), \mathsf{r} \mapsto \iota, \mathsf{x} \mapsto v'\} \rightsquigarrow dv, h' \end{array}}{P \vdash e_0\texttt{.}m\texttt{(}e_1\texttt{)}, h, s \rightsquigarrow dv, h'}$$

Figure 7.28: Rules for propagation of a deviation occurring in advice before, during or after a method call

In ASPECTJ, advice is chosen based on the static type of the target object expression; calls to advice bodies are inserted by a weaving process applied to the code before execution. In ASPECTJ, when a member access results in a null pointer error, the before advice, already selected by the weaving process, will be executed before the program flow is interrupted by throwing a `NullPointerException`.

Reduction rules for propagating deviations

The rule in figure 7.29 propagates deviations arising in the body of an advice. If a deviation arises in the execution of an advice body it will be propagated up by the rule for propagation of deviations through a sequence of expressions (recall that a sequence of applicable advice is executed at each join point using the standard rule for sequential composition of expressions using ' ;') defined in figure 4.38.

$$[RWAD]\ \frac{P \vdash e, h, \{\texttt{this} \mapsto \mathsf{owningAspect}^{\ell}(P, a)\} \rightsquigarrow dv, h'}{P \vdash a : \alpha\ \pi\{e\}, h, s \rightsquigarrow dv, h'}$$

Figure 7.29: Rule for propagation of deviation through advice

7.5 Observations

7.5.1 Selecting advice

In order to select advice to run at a join point, it is necessary to consider each advice in the program and see if its pointcuts expression matches the details of the join point. For member access pointcuts, ASPECTJ compares the static information about the join points with the advice and pre-selects advice candidates for each join point (final selection is made at run time for those join points that contain `target`, `this` or `args` pointcuts, since these constrain the dynamic type of objects). aj, on the other hand select advice at execution-time, when static type information is not available. As pointed out in section 7.4.5, this means that ASPECTJ will execute before advice before generating a null pointer exception whereas aj will not. In the context of aj, this difference is slight and only visible when a program fails due to dereferencing a null pointer. ASPECTJ is, however, a derivative of JAVA, and in JAVA the occurrence of a null pointer exception does not necessarily signify failure of the program since exceptions can be caught.

The behaviour of ASPECTJ could be modelled mode accurately by, for example:

– specifying a source to source transformation that weaves the code for advice bodies into the program before and after the join points where they are applicable.

– using an augmented syntax that keeps relevant type information together with member accesses

The former approach would be similar to the technique used to build early implementations of ASPECTJ. This choice would not serve the purpose of making a clear presentation of semantics since a source transformation would not directly model the behaviour of the advice construct. Additionally, advice bodies may contain expressions that are themselves join points and may, therefore, have before and after advice of their own; the programs resulting from such a transformation might, therefore be complex and hard to follow. This complexity would be exacerbated by the fact that each advice body must be executed in the context of a set of appropriately bound variables to expose information from the join point.

The latter approach would be similar to models of the JAVA byte code [26]. This would involve creation of an augmented version of the languages in the j family to store the additional type information. This additional information would be used only in solving the problem of selecting advice.

A technique similar to this is used to relate advice back to the aspect class in which it was defined for the purpose of finding the singleton instance of the aspect class to bind to `this` in the advice body: each advice declaration in aj has a unique identifier which does not appear in ASPECTJ. The additional syntactical clutter introduced in this case is limited to the syntax of aj and does not permeate the other languages in the j family.

The chosen approach is to write a simple reduction rule for the execution of member access expressions that form join points using a simple model that closely resembles the ASPECTJ source code. Since static type information is not available in the reduction rule used the dynamic type of the target object which serves as a close model.

Note also ASPECTJ includes a pointcuts designator `target` which can be used to select join points based on the dynamic type of the target.

7.5.2 Advice context

In aj, all advice bodies run in the context of the singleton instance of the aspect class in which the advice was defined. In ASPECTJ there are there are a number of ways to instantiate aspect classes. Aspect definitions can be annotated to indicate that instances should be created when certain pointcuts (defined within that aspect) occur. For example, an aspect can be declared `percflow` π (for some pointcuts π). Which means that an instance is created each time the flow of control passes the point indicated by π. Alternatively, an aspect may be declared `perthis` π in which case an instance is created for each distinct object that plays the role of `this` in an occurrence of pointcuts π. There is also an annotation `issingleton` which indicates that there should be one instance per virtual machine, this is the default case and the one chosen for this model.

7.5.3 Other kinds of join point

ASPECTJ has several additional kinds of join points that act on language features in JAVA that are not present in aj. For example in ASPECTJ it is possible to place advice on calls to class constructors, static methods, static and non- static initialiser blocks and try/catch blocks. It is also possible to use an aspect to add exception handling code to a program that contains none and, as a result, would not compile in JAVA due to uncaught exception messages. Constructors, initialisers, static methods and exception handling were deliberately left out of the j family of languages to keep them simple. The inference rules for the rewrite relation in figure 4.35 are decomposed, in the conventional way, into roughly one rule per case of the syntax of expressions (figure 4.3). The rules in figure 7.26 define the execution of advice [*RWMETH*], and redefine execution of the three kinds of expression that form join points in aj: field access [*RWGET*], field update [*RWSET*] and method call [*RWMETH*]. Each of these resembles the corresponding rule from figure 4.35 but with the addition of lines that select an appropriate sequence of applicable advice and execute these before and after the join point expression. Although the rules differ in their details, the clearly follow a pattern: fetch the applicable advice, execute the before-advice, then the target expression, then the after-advice. This pattern would apply equally well to other expressions, were they to become join points.

7.5.4 Signatures in pointcuts

The model presented here does not quite demonstrate everything that can be achieved with the `call` pointcuts in ASPECTJ. This is because j does not allow method overloading: a decision that was made deliberately, to keep the model simple. In ASPECTJ, `call` pointcuts select join points by the name and signature of the called method. A method can be identified by its name and defining class, unless the language permits overloading, when the type of the parameters is necessary. The signature information in aj's `call` pointcuts is not redundant, however, since the method name may be replaced with a wildcard, in which case the type information is used to choose joint points. Consider, for example, a pointcuts of the form: `call(bool` C`.*(*))` this will match any call to a boolean function method defined in class C.

7.5.5 Modularity

Aspects are introduced in this chapter. In ASPECTJ, aspects are the special module that allows code relating to cross-cutting concerns, whose separation is facilitated by introduction and advice, to be encapsulated. This work presents the behaviour of advice and introduction by means of operational semantics. It has not addressed the issues of modularity and encapsulation of separated concerns. In aj it is possible to define advice only inside an aspect so, in a sense, advice definitions will be encapsulated within aspects. As seen in example program `P1` at the beginning of this chapter, advice bodies may update variables belonging to the aspect in which they are defined. In the example program two advice bodies use a field to communicate: the before advice sets a flag and

the after advice consults it. This use of state is separated from the rest of the program because the field belongs to an instance of the aspect. The aspect instance is bound to the special `this` variable when the advice body executes. This was the motivation for introducing aspects in aj: making aspect classes means that there is a way to denote which objects should be used as the context for advice bodies. Recall that aspects may not be instantiated with `new` expressions but, instead, singleton instances are provided in a prepared heap, with mappings from the aspect class names to the addresses of their respective instances. The reduction rule for executing advice looks up the advice's containing aspect class and uses that to locate its instance to bind to `this`.

7.5.6 Non-termination

Expressions involving the execution of advice might be non-terminating if the advice body entails an occurrence of the join point in which the advice is defined.

For example the after advice in aspect `A` in figure 7.30 would not terminate if it were invoked because of, for example, executing the field access to `A.flag` in method `getFlag()`: the body of the advice contains another access to the field which would

```
aspect A {
  bool flag
  bool getFlag(){
    this.flag;
  }
  a1: after get(bool A.flag){
    ...  this.flag; ...
  }
}
```

Figure 7.30: Non-terminating advice

invoke the advice body again when it executes, and so on.

In ASPECTJ the programmer would be required to avoid this consequence by defining the pointcut precisely to be not all accesses to `A.flag`, but those outside the aspect. This would be done using ASPECTJ's `within` pointcut designator combined using the `&&` and `!` combinators like this:

```
after(): get(bool A.flag) &&! within(A) { ... }
```

These additional combinators and designators are not included in aj.

7.6 Concept List

Term	Meaning	Location
$\mathsf{advice}^{\mathsf{aj}}(P)$	All advice defined in program P	figure 7.7
$\mathsf{Aspects}^{\ell}(P)$	(Sequence of) names of aspects in program P	figure 7.8
$\vdash P \overset{\triangle}{ua}$	Advice identifiers are unique within P	figure 7.9
$\mathsf{owningAspect}^{\mathsf{aj}}(P, a)$	Name of aspect class name owning definition of a	figure 7.10
$P \vdash_{\ell} C \overset{\triangle}{at}$	C is an aspect class in P	figure 7.11
$\vdash_{\mathsf{J}} P \overset{\triangle}{SS}$	Structurally sound program	figure 4.22
$\mathsf{Fs}^{\mathsf{pj}}(P, C, f)$, $\mathsf{Ms}^{\mathsf{aj}}(P, C, m)$	Field and method select	figure 4.23
$\mathsf{F}^{\mathsf{j}}(P, C, f)$, $\mathsf{M}^{\mathsf{j}}(P, C, m)$	Field and method lookup	figure 4.24
$\mathsf{C}^{\ell}_{\mathsf{def}}(P, C, id)$	Class where a member was defined	figure 7.13
γ^{aj}	Type checking environment	figure 7.14
$P, C \vdash A \diamond$	Type typed advice	figure 7.15
$P \vdash_{\ell} C \diamond$	Well formed aspect classes	figure 7.16
jp	Join points	figure 7.17
$P \vdash A_1..A_n \overset{\triangle}{ord}$	Advice order	figure 7.19
$* \sqsubset *$	Coverage and matching	figure 7.20
$\mathsf{applicable}^{\ell}(P, \alpha, jp)$	Advice applicability	figure 7.21
$\mathsf{advices}^{\ell}(P, \alpha, jp)$	Sequences of advice applicable to jp	figure 7.22
$[\![f{:}v \ldots]\!]^{C}$	Objects	figure 4.31
$\mathsf{new}^{\mathsf{aj}}(P, C)$	New object of class C	figure 4.32
h, s	Heaps and stack frames	figure 7.23
h_P	Prepared heap	figure 7.24
$P \vdash A, h, s \rightsquigarrow v, h'$	Advice execution	figure 7.25
$P \vdash_{\mathsf{jj}} e, h, s \rightsquigarrow v, h'$	Rewrite relation	figure 7.26

Chapter 8

Conclusion

8.1 Observations

The substance of this thesis is a precise, faithful and understandable model of the principal novel features of ASPECTJ. This section presents some observations that have arisen as a result of building such a mode.

8.1.1 On the nature of aspect-oriented modularity

One the goals of aspect-oriented programming is to design new language features that support modular separation of concerns better than traditional[1] languages have done. The better traditional languages allow separation of concerns into modules that align with the structure of code artifacts. Code, being formed of text that is essentially sequential, can be decomposed easily, in one dimension, into files, classes/units/modules, etc., functions, lines, etc. Aspect-oriented techniques transcend this dependency on a single dimension, known as the tyranny of the dominant decomposition [75], in the case of ASPECTJ, this is done by introducing aspects: modules with non-local affects: the contents of an aspect may affect the behaviour of a program elsewhere than just inside that aspect.

In traditional languages, the behaviour of a statement may be described by a rule which is written with knowledge of that expression alone. ASPECTJ, however, the behaviour of one statement may be affected by all the aspects in the program to which it belongs. The rewrite relation ($P \vdash e, h, s \rightsquigarrow v, h'$), therefore, has the program (P) as part of its context, allowing all the program's advice to be taken into account when each expression executes. Thus the resulting semantics is unconventional in that it requires a global perspective on the program at every step. This is a result of the requirement to

[1] The word *'traditional'* is used here to mean non-aspect oriented, so relatively recent developments like object-oriented programming, which might in other contexts not be considered traditional, are included.

handle cross-cutting concerns. Since the concerns cut across the program as a whole (scattering) and at any point in the program's execution, potentially any subset of the concerns in the program might apply, it is necessary either:

- to keep the whole program available at each step
- to perform a weaving step before execution to bring all relevant code together at each points.

This is an insight about the nature of aspect oriented programming: its aim – to separate cross-cutting concerns – requires the relation of the local (point of execution) to the global (the context of cross-cutting) to be taken into account. Some researchers, for example Devereux [22], have tackled the problem of reasoning about aspect-oriented modules in a modular way.

8.1.2 Illuminating language design decisions

The motive for some of the changes made by the ASPECTJ development team, during the language's development, can be understood as a result of this model.

Pointcut semantics

During the early development of this model ASPECTJ included the old two-argument `calls` joinpoint designator (described in section section 3.1.1). To model this the `call` basic pointcut expression in aj had a pattern with five positions for type patterns corresponding to the parts of a call specification. (Recall that the old two-argument calls form looks, in general, like this: `calls(`$C, T_r[T].m\big(T_x\big)$`)` Where C is the dynamic type of the receiver, T_r is the return type, T is the (optional) class in which the method was defined, m is the method name and T_x is the type of the formal parameter.) The type patterns in this structure were treated somewhat differently from one another: C was matched dynamically against the type of the receiving object (just like a JAVA `instanceof` test) whereas T_r was used to statically determine the type in which the method was defined. Since `instanceof` tests for inclusion in the polymorphic type family defined by its argument type the effect was that all subtypes of C matched, whereas T matched exactly. That the aj model highlighted these differences illustrates the effectiveness of the formal model.

At about this time, the ASPECTJ team made changes to the language specification that simplified the syntax of ASPECTJ and the conceptual model that underpins it. This was reflected in a consequent simplification of the formal model aj.

In fact the coverage of aj was slightly reduced by this simplification as at this time ASPECTJ gained the `target`, `this` and `args` designators which are not modelled in aj. Including them, however, would be conceptually much simpler than modelling the original two argument version of `calls` because the separate handling of the type patterns is separated between the two designators.

Similarly, the different handling of type patterns with respect to whether they specify a single type or a subtype family to match with was simplified in ASPECTJ by in introduction of a syntactic modifier (a '+' placed after the type name – though when first introduced this was a function `subtypes(T)`) to indicate that subtypes should match.

Advice variables

In early versions of ASPECTJ, variables that bind to join point values were declared inside advice headers and inserted in place of type patterns in pointcut specifications where they stood for their types, from the perspective of type patterns, and their positions determined what values should be bound to them when the join points occurred. This mixing of function – the slots in the pointcut designator serve to hold either variables or type patterns – was potentially confusing to programmers, especially so when the type pattern specified a pattern (using wildcards) and not just a type.

In aj the two are kept separate. On of the aims of the j family is simplicity; some features are omitted from the model for the sake of this aim. For binding variables, a simplified but different mechanism is used for binding variables to joinpoint values. The binding is implicit, not explicit: certain variables are always in scope inside an advice body, whether or not the programmer needs to use them. The programmer has no control over the names of these. This leaves the definition of pointcuts uncluttered by variable definition. Any mechanism for binding variables to these values would require matching of the values to the names in some way. The complexity of such a mechanism outweighs the advantage of including variable naming in the model.

The aj model clearly illustrates the declaration and matching of pointcut patterns uncluttered by additional syntax for declaring and binding variables.

8.2 The aims of the model

ASPECTJ extends JAVA; adding a number of features to support aspect-oriented programming. The j family of languages provide a model of the core features of ASPECTJ– introduction and advice – in the context of a simple formal language-model. The aim of creating the j family of languages was to describe, in a precise, faithful and understandable way, the features that underpin aspect-oriented programming in ASPECTJ. A set of features is modelled that is large enough to cover the significant innovations of ASPECTJ but small enough to remain simple to comprehend; in other words to distill the essence of ASPECTJ. This section discusses how the model achieves its aims.

8.2.1 Precision

The model is a precise one. Each language in the j family is a calculus whose preterms are defined formally in Bacus Naur [58] Well formedness judgements define by inference rules describe the valid terms. Consistently named sets of functions are defined to

project values from terms in each language. Type inference rules for terms are defined together with a type hierarchy defined as a bounded partial order on type names. Execution is precisely described by a set of inference rules that define a reduction system for expressions in the context of a program, including rules for getting stuck due to type errors or null pointer errors.

8.2.2 Simplicity

The models have been kept simple. j models only features of JAVA that are necessary to maintain its object nature and to support the description of aspect-oriented features in its descendants. Concepts such as static members, inner classes, object constructors, object initialisers, visibility control modifiers (`public`, `private`, etc.), loops and exception handlers have been excluded, together with strings, arrays and all but one built-in type. The features that remain capture the essence of object-oriented programming in JAVA: classes, interfaces, (non-static) methods and fields, inheritance, sequential expression composition, assignment and object references. Similar decisions have guided the selection of features in the descendant languages: ij adds to j only introduction of (implicitly public) methods and fields into other classes. pj adds to ij only introduction of superclasses and super-interfaces. aj adds to j advice and aspects (because they are needed to support the execution of advice bodies).

Java has many more features and so does ASPECTJ. The complexity of ASPECTJ arises from the cross product of these feature sets. Each additional join point expression in JAVA increases the number of execution cases by the number of primitive join points kinds. Each new join point increases the complexity by the number of expressions.

The features are presented separately in a modular form as increments on the base language j. This approach could be described as an aspect-oriented one since it separates the major concerns in aspect oriented programming and encapsulates them in separate chapters. The burden of repetition between chapters is reduced by means of the parameterised definition of reusable formal notation. Thus it has been able to concentrate on the mechanisms behind introduction and advice, without the complexity that arises from the interplay of ASPECTJ features with JAVA features. For example, ASPECTJ has a number of designators for specifying join points on try and catch blocks, constructors, etc. Each of these would require a reduction rule that selected and executed the relevant sequence of before and after advice in the same way as do the rules for method call, field access and update in aj.

8.2.3 Faithfulness

The j family are reasonably faithful to the subset of the features of JAVA and ASPECTJ that they model. The behaviour described here matches that described by the language documentation wherever the documentation is clear. As of version 1.0 of ASPECTJ, there are three documents that describe the language: a programming guide, a quick reference guide and the guide to porting programs written in previous versions of the language. None of these describes, for example, precisely when before advice runs:

does it run before of after evaluation of the expression that gives the target object. Where necessary, (and where possible) to clarify these issues, the compiler has been used as a reference: short test programs were written that reveal the behaviour of the language implementation. These can be found in appendix A.1.

As explained in section 7.5.1, there is a difference between the behaviours of my model and ASPECTJ when a null pointer is dereferenced in a field access expression that has advice defined on it: in ASPECTJ the before advice will run before the exception is raised. In aj the null pointer error will be propagated without executing the before advice. In either case, it is not appropriate that the after advice should run as the program execution stops when the null pointer is dereferenced.

8.2.4 Coverage

As described in section 3.1.2, the calculi presented in this thesis encode language features from each of the main groups of features in ASPECTJ: default aspect creation, before and after advice, pointcuts on member accesses, exposing values associated with join points, member introduction and parent introduction.

This work explores the behaviour of the advice and introduction features of ASPECTJ. These features alone, however, do not make an aspect-oriented language. They are included in ASPECTJ in order that it support the goal of aspect-oriented programming: separation and modularisation of cross-cutting concerns. To support this goal of modularisation, the new features of ASPECTJ are available only within aspect classes; aspect oriented features are encapsulated in aspects! ASPECTJ incorporates some clever visibility control mechanisms, for example: features introduced into a class and marked as `private` in the aspect are private to the aspect that defined them, not to the class into which they are introduced. The aspect may introduce public access methods for those features into the same class and it may define advice defined on join points in the code of that class which access those private members.

This thesis makes no attempt to deal with the problem of controlling access to introduced members, as such it does not provide a complete solution to the problem of modularising cross-cutting concerns, but rather it explain, in a clear manner, the behaviour of the introduction and advice mechanisms.

8.3 Building the model

Building a model of ASPECTJ has been an interesting challenge. The very features that make it an interesting language have, themselves, been challenging. But more than this, the way in which the language has developed has meant that the production of a formal model was rather like shooting at a moving target. This section presents the challenges of building a formal model of a language while the language itself is being developed.

ASPECTJ has been developed in an unusual way: by constant iteration with the combined driving forces of the ASPECTJ team's vision and the user community's feedback. The original vision of aspect oriented programming entailed a collection of domain-specific aspect languages[42, 44], the idea of a general purpose aspect language based on JAVA emerged soon after [79], the basic notions of introduction and advice have been in place since then. There have, however, been several significant changes to the syntax of those features. Unlike most languages, for ASPECTJ, the notes on porting code in from one version to the next make interesting reading. This means that building a model of ASPECTJ has been, to some extent, like shooting at a moving target. The ASPECTJ team did not claim to have a stable notation until version 1.0 of the language. This was made clear to the user community and enabled the team to make the sweeping changes when they did which have enabled the language to improve and grow, learning from experience. The only way to get experience with an aspect-oriented language is to use one and when ASPECTJ was the only one there wasn't a lot of choice.

Some of these improvements in language design have made my job easier, for example: the porting notes recognise that, in some previous versions of the language, the patterns used to match types in pointcut signatures were not handled in a uniform way: some were matched exactly, some on subtypes and some with wildcards. It was necessary to write some test programs[2] to find out which were which in the then-current implementation. This lack of clarity was spotted by the developers who responded by simplifying the language and making the matching consistent and introducing new pointcut designators to explicitly encode the alternative behaviours. My response was blithely to simplify my model.

8.4 Using the model

This section presents observations on the use of the model developed in this thesis.

Programming language descriptions written in English (or, I am given to believe, any other natural language) tend to be less than perfectly clear. This is due partly to the fact that natural languages are inherently ambiguous (with, for example, one word or phrase having many meanings) and partly because of the way humans think (very often, we think we have finished when much remains to be done). Also, it is sometimes the case that, for reasons of economy, language specifications are written to serve more than one purpose: to serve as a general introduction to the concepts of the language, to serve as a guide for programmers, to be submitted for publication; as well as to precisely define the language syntax and semantics. When the time comes to supplement the natural language description of a language with something to make its syntax and semantics precise, a faithful, simple and precise model of the language and its operational semantics it a good candidate.

This is true of ASPECTJ. During its five year development period, the documentation has changed many times. Sometimes there were separate documents introducing the concepts, teaching programmers and specifying the language, sometimes these were combined. At the time of writing, the language is at version 1.1. The documentation

[2] Similar to those in appendix A

comprises: a two page ‘Quick Reference’ guide, a ‘Programming Guide’ and a set of example programs. At no time has a formal semantic model been published by the ASPECTJ team. Their mode of working has been to release a reference implementation (the ASPECTJ compiler release) and to encourage discussion among the user community by means of an email list. So, in a sense, the compiler has been the specification. But the ASPECTJ programming guide[6] (sic) says:

> Certain elements of ASPECTJ’s semantics are difficult to implement without making modifications to the virtual machine. One way to deal with this problem would be to specify only the behaviour that is easiest to implement. We have chosen a somewhat different approach, which is to specify an ideal language semantics, as well as a clearly defined way in which implementations are allowed to deviate from that semantics. This makes it possible to develop conforming ASPECTJ implementations today, while still making it clear what later, and presumably better, implementations should do tomorrow.

This means that no version of the ASPECTJ compiler can be taken as the definitive reference. This makes a very case for a formal definition of the language.

The work presented here would also be suitable for developing as part of a tool-set for reasoning about aspect-oriented programming. An operational semantics is a tool that can be used in verifying an axiomatic semantics. With some work (i.e. extending the work, suggested above, to integrate the separate features and prepare a proof of soundness) and if a fuller set of ASPECTJ’s features were modelled, the operational semantics could be a useful tool for those who continue to develop ASPECTJ; to help detect possible conflicts between proposed features and those of JAVA or ASPECTJ.

8.5 Directions for future work

8.5.1 Adding features

The shape and scope of the languages presented in this thesis has been determined in an effort to distill the essence of ASPECTJ while still covering its major areas of functionality. The work presented here contains a selected set of ASPECTJ features supported by a simple subset of JAVA features. Roughly speaking, the scope of this work could be increased by extending either of these feature sets. Although it is tempting to see the feature coverage of JAVA and ASPECTJ as independent axes, in many cases there are dependencies between features:

– Adding, for example, static methods to j would entail some work in aj to ensure that accesses to static and non-static members can be distinguished in pointcuts. It would also be necessary to ensure that the special variables in advice bodies are bound appropriately since there is, for example, no instance in the role of target to bind to r when static members are accessed.

– Adding, the `cflow` primitive join point to aj would entail some work on the treatment of the semantics of j to make the whole call stack (not just the current frame) available so that advice can determine whether each join point is in the control flow of the `cflow` pointcuts expression (an expression e is in the control flow of a given pointcuts π if there is a method call matching π somewhere deeper in the stack when e is executed). It also raises the issue of whether to include ”per cflow” aspect instantiation.

Due to these dependencies, some features of ASPECTJ cannot be added without first adding other, necessary, features of JAVA. For example, ASPECTJ's exception handling functionality requires exceptions to be present in the base language, and putting advice on constructors and/or initialisers would require those features to be present. Some of these dependencies are more subtle. To distinguish between `call` and `execution` pointcuts, it is necessary to add method invocations via `super` (although j currently supports method overriding, this feature is not present). An `execution` pointcuts occurs every time a matching method body is executed, whereas a `call` pointcuts occurs only when a method is called (and not when overridden method bodies are invoked via `super`).

8.5.2 Developing the model

It would be possible to bring the branches of the j family together to produce one language with all the features: feature and member introduction. Such a language would be more complex than those presented here. It would, however, be a good starting point for constructing a proof of soundness for the language: because all the modelled features would be included, the soundness proof would build confidence that there are no undesirable conflicts between them. If kept consistent with the presentation of j, ij, pj and aj, and presented together with their individual presentations given in this thesis, the merged language would be easy to comprehend.

The weaving functions of ij and pj could be combined to give a single function. This suggests that it might also be possible to extend the work of this thesis by creating a weave function for aj that pre-selects advice to run at join points in the way that ASPECTJ does.

8.6 Related work

As part of a project called the Aspect Sand Box, Mitchell Wand [81] has prepared a denotational semantics for an aspect-oriented language with dynamic join points. The base language used for the semantic study is not object-oriented: it does not include types, classes or objects. The emphasis of this work is on capturing the behaviour of the `cflow` dynamic pointcuts designator, which allows selection of join-points at run time based on whether or not they have occurred within the control flow of a method (specified by another pointcuts). The pointcuts language, therefore, includes designators for procedure calls and control flows, but not field access or update. This model

also supports the `around` form of advice which replaces the expression with the advice body expression which may contain an occurrence of `proceed` which invokes the original pointcuts expression code. In order to match `cflow` pointcuts the model has a rich runtime representation that includes details of the call stack.

Sipma [68] has modelled aspect-oriented programming using semantic transformations on modular transition systems. His model is built on a simple (non object-oriented) language which translates to a transition system, together with aspects modelled as a set of *aspect facets*, one for each module in the program, that define new variables and also abstraction and restriction rules that modify the behaviour of the module being touched. This model is not object-oriented, and does not model closely any existing aspect-oriented language. It does, however, model inserting one or more statements before or after a given statement.

Tucker [76, 77] defines pointcuts and advice for higher order functional languages. This work tackles the problems of designing advice on function invocations where functions are first-class entities that might not have a name in the sense that methods do in an object-oriented language. An implementation in a functional language is presented.

Lämmel has done considerable work on the semantics of aspect-oriented languages. In [46] he develops aspect oriented extensions for a declarative language in which weaving is performed using functional meta-programs.

In [47] Lämmel uses an operation called *superimpose* to intercept method-call joinpoints and attach additional behaviour to them. The superimpose construct is added to a natural semantics of an object-oriented language. The use of natural semantics and an object-oriented language makes this a very similar treatment to that of aj in Chapter 7. Lämmel uses a novel construct for registering advice that elegantly addresses the problem of making the advice body type safe. By contrast, aj models the mechanism described by ASPECTJ. The only join- points in this system are method calls, specified (dynamically) by an object location (the target of the call) and a method name. Superimpositions can be registered dynamically into a run-time registry, in fact they must be since actual object addresses are used to specify the method-call join points. No notion of pattern or type matching is included (though some simple matching on the type of method arguments could easily be added). Lämmel's conclusions, that treating advice-like mechanisms as language features whose semantics are described abstractly (as opposed to by means of coding styles or specific encodings or program transformations) is consistent with the aims of this thesis to present a simple comprehensible model of the language feature in a way that does not prescribe implementation and allows all kinds of optimisations.

In [48] Lämmel adds *superimposition* to a denotational semantics in a language parametric manner. This involves elaborating the semantic functions to carry advice. The denotations used are made reflective so that they can be altered, and an advice binding construct is added.

Bibliography

[1] M. Abadi and L. Cardelli. An imperative object calculus: basic typing and soundness. *Theory and Practise of Object Systems*, 1(3):151–166, 1995.

[2] M. Abadi and L. Cardelli. *A Theory Of Objects*. Springer, 1996.

[3] D. Ancona, G. Lagorio, and E. Zucca. Jam - a smooth extension of Java with mixins. In *Proceedings of ECOOP'00*, 2000.

[4] E. P. Andersen. *Conceptual Modeling of Objects: A Role Modeling Approach*. PhD thesis, Department of Informatics, Faculty of Mathematics and Natural Sciences, University Of Oslo, Nov. 1997.

[5] E. P. Andersen and T. Reenskaug. System design by composing structures of interacting objects. In *Proceedings of ECOOP '92*, Lecture Notes In Computer Science, pages 133–152. Springer, 1992.

[6] The AspectJ programming guidel. `http://www.eclipse.org/aspectj`, 2001.

[7] AspectJ lannguage specification. `http://aspectj.org`, 1999. Modified 10 August 1999.

[8] L. M. J. Bergmans. *Composing Concurrent Objects - Applying Composition Filters for the Development and Reuse of Concurrent Object-Oriented Programs*. PhD thesis, Department of Computer Science, University of Twente, Netherlands, June 1994.

[9] L. M. J. Bergmans. The composition-filters object model. Technical report, Department of Computer Science, University of Twente, 1994.

[10] L. M. J. Bergmans and M. Aksid. Composing crosscutting concerns using composition filters. *Commun. ACM*, 44(10):51–57, Oct. 2001.

[11] D. G. Borrow, L. DeMichiel, R. P. Gabriel, G. Kiczales, D. Moon, D. Moon, and S. Keene. The Common Lisp Object System specification: Chapters 1 and : Chapters 1 and 2. Technical Report 88-002R, X3J13 standards committee, 1988.

[12] N. Boyen, C. Lucas, and P. Steyaert. Generalised mixin-based inheritance to support multiple inheritance. Technical report vub-prog-tr-94-12, Programming Technology Lab, Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussel, Belgium, 1994.

[13] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP '90*, pages 303–311. ACM, Oct. 1990.

[14] J. Cage. Concerto for prepared piano and chamber orchestra. Edition Peters 6706, Henmar Press, 1960.

[15] L. Cardelli. A semantics of multiple inheritance. *Information and computation*, 76(2 /3):138–164, Feb./Mar. 1988.

[16] L. Cardelli. Program fragments, linking and modularization. In *Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages, Paris, France*. ACM, January 1997.

[17] W. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.

[18] W. Cook. A proposal for making Eiffel type safe. In *Proceedings of ECOOP'89the 3rd European Conference on Object-Oriented Programming.*, 1989.

[19] W. Cook and J. Palsberg. A denotational demantics of inheritance and its correctness. In *OOPSLA '89 Conference on Object-Oriented Programming Systms Languages and Applications*, pages 433–443. ACM, October 1989.

[20] B. J. Cox. *Object-Oriented Programming - an Evolutionary Approach*. Addison-Wesley, Reading, 1986.

[21] U. David and R. B. Smith. SELF: The power of simplicity. *SIGPLAN Notices*, 22(12):227–247, Dec. 1987.

[22] B. Devereux. Compositional reasoning about aspects using alternating-time logic. In *Proceedings of Foundations of Aspect-Oriented Languages (FOAL) '03*, March 2003.

[23] S. Drossopoulou. Program fragments, linking and modularization. In *ECOOP'01*, LNCS 2072, pages 130–149. Springer, 2001.

[24] S. Drossopoulou, S. Eisenbach, and S. Khurshid. Is the Java type system sound? *Theory and Practise of Object Systems*, 5(1):3–24, 1999.

[25] S. Drossopoulou, D. F, M. Dezani-Ciancaglini, and P. Giannini. Fickle: Dynamic object re-classification. In *ECOOP'01*, LNCS 2072, pages 130 – 149. Springer, 2001.

[26] S. Drossopoulou, T. Valkevych, and S. Eisenbach. Java type soundness revisited, 2000.

[27] D. Duggan and C. Sourelis. Mixin modules. *ACM SIGPLAN Notices*, 31(6):262–273, June 1996.

[28] K. Fisher and J. C. Mitchell. A delegation based object calculus with subtyping. In *Fundamentals of computation theory (FCT'95)*. Springer LNCS, 1995.

[29] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 171–183, New York, NY, Jan. 1998. ACM.

[30] M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. Technical Report TR 97-293, Rice University, June 1999. Correction of earlier work.

[31] E. Gamma, R. Helm, R. Johnson, and J. Vissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing. Addison-Wesley, 1994.

[32] A. Goldberg and D. Robson. *SmallTalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

[33] M. J. Gordon. *The Denotational Description Of Programming Languages*. Springer Verlag, 1979.

[34] G. Gottlob, M. Schrefl, and B. Röck. Extending object-oriented systems with roles. *ACM Transactions on Information Systems*, 14(3):268–296, 1996.

[35] J. Hannemann and G. Kiczales. Design pattern implementation in java and aspectj. In *OOPSLA '02*. ACM, November 2002.

[36] W. Harrison, H. Kilov, H. Ossher, and I. Simmonds. From dynamic supertypes to Subjects: A natural way to specify and develop systems. *IBM Systems Journal*, June 1996.

[37] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of OOPSLA '93*, pages 411–428. ACM, 1993.

[38] W. Harrison, H. Ossher, M. Kaplan, A. Katz, and V. Kruskal. Writing composable programs in C++. Review Draft 2. IBM T. J. Watson Research Centre, P.O. Box 704, Yorktown Heights, N.Y. 10598, 1995.

[39] F. Inc. Flavors. Flavors manual distributed with ALLEGRO CL (CLOS programming environment), 1999.

[40] M. Kaplan, H. Ossher, W. Harrison, and V. Kruskal. Subject-oriented design and the Watson Subject compiler. Position Paper for OOPSLA '96 Subjectivity Workshop, 1996.

[41] G. Kiczales and D. G. Borrow. The Common Lisp Object System specification: Metaobject protocol. Technical Report 88-003, X3J13 standards committee, 1988.

[42] G. Kiczales, J. Irwin, J. Lamping, J.-M. Loingtier, C. Videria Lopes, C. Maeda, and A. Mendhekar. Aspect-Oriented Programming - a position paper from the Xerox PARC Aspect-Oriented Programming project. `http://www.parc.xerox.com/spl/projects/aop/`, 1997.

[43] G. Kiczales, J. Lamping, and A. Mendhekar. Aspect-oriented programming. In *European Conference on Object-Oriented Programming ECOOP*. Springer-Verlag, June 1997. LNCS 1241.

[44] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)*. Springer-Verlag LNCS 1241, June 1997.

[45] B. B. Kristensen and K. Østerbye. Roles: Conceptual abstraction theory and practical language issues. *Theory and Practise of Object Systems*, 2(3):143–160, 1996.

[46] R. Lämmel. Declarative aspect-oriented programming. In O. Danvy, editor, *Proceedings PEPM'99, 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation PEPM'99, San Antonio (Texas), BRICS Notes Series NS-99-1*, pages 131–146, Jan. 1999.

[47] R. Lämmel. A Semantical Approach to Method-Call Interception. In *Proc. of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002)*, pages 41–55, Twente, The Netherlands, Apr. 2002. ACM Press.

[48] R. Lämmel. Adding Superimposition To a Language Semantics — Extended Abstract. In G. T. Leavens and C. Clifton, editors, *FOAL'03 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2002; Technical Report CS Dept., Iowa State Univ.*, Mar. 2003. 6 pages.

[49] B. Liskov. Data abstraction and hierarchy. *SIGPLAN Notices*, 23(5), May 1988.

[50] C. Lucas and N. Boyen. Subjectivity through mixin-methods. Position Paper for OOPSLA '94 Subjectivity Workshop, 1994.

[51] J. Martin and J. Odell. *Object-Oriented Analysis & Design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[52] J. D. McGregor and T. Korson. Supporting dimensions of classification in object-oriented design. *Journal of Object Oriented Programming*, pages 25–30, February 1993.

[53] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1st edition, 1988.

[54] B. Meyer. *Eiffel: The Language*. Advances in Object-Oriented Software Engineering. Prentice Hall, 66 Wood Lane, HP2 4RG, UK, 1992.

[55] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[56] R. Milner, T. M, and H. R. *The Definition of Standard ML*. MIT Press, 1989.

[57] D. A. Moon. Object-oriented programming with Flavors. In *Proceedings of OOPSLA '86*, pages 1–6. ACM, 1986.

[58] P. Naur. Revised report on the algorithmic language ALGOL60. *Commun. ACM*, 3(5):299–314, May 1960.

[59] H. Ossher, M. Kaplan, W. Harrison, A. Katz, and V. Kruskal. Subject-oriented composition rules. In *Proceedings of OOPSLA '95*. ACM, 1995.

[60] H. Ossher, M. Kaplan, A. Katz, H. William, and V. Kruskal. Specifying Subject-oriented composition. *Theory and Practise of Object Systems*, 2(3):179–202, 1996.

[61] H. Ossher and P. Tarr. Multi-dimensional separation of concerns in hyperspace. Research Report RC21452(96717)16APR99, IBM T.J. Watson Research Cenntre, PO Box 704, Yorktown Heights, NY 10598, 1999.

[62] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, Dec. 1987.

[63] T. Reenskaug. OOram role model analysis and synthesis. Position Paper for OOPSLA '94 Subjectivity Workshop, 1994.

[64] T. Reenskaug, E. P. Andersen, A. J. Berre, A. Hurlen, A. Landmark, O. A. Lehne, E. Nordhangen, E. Nêss-Ulseth, G. Oftedal, A. L. Skaar, and P. Stenslet. OORASS: Seamless support for the creation and maintenance of object oriented systems. *Journal of Object Oriented Programming*, pages 27–41, October 1992.

[65] T. Reenskaug, P. Wold, and O. A. Lehne. *Working with Objects: The OOram Software Engineering Method*. Manning Publications Company, 3 Lewis Street, Greenwich, CT 06830, 1995.

[66] D. Scott. Data types as lattices. *SIAM Journal on Computing*, 5(3):522–587, 1976.

[67] J. Shilling and P. Sweeney. Three steps to views: Extending the Object-Oriented paradigm. In *Proceedings of OOPSLA '89*, pages 353–361, 1989.

[68] H. B. Sipma. A formal model for cross-cutting modular transition systems. In *Proceedings of Foundations of Aspect-Oriented Languages (FOAL) '03*, March 2003.

[69] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In E. Jul, editor, *ECOOP '98—Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 550–570. Springer, 1998.

[70] Y. Smaragdakis and D. Batory. Mixin-based programming in C++. Technical Report CS-TR-98-27, University of Texas, Austin, Dec. 1, 1998.

[71] R. Smith and D. Ungar. A simple and unifying approach to subjective objects. *Theory and Practise of Object Systems*, 2(3):161–178, 1996. This paper was previously available as an unpublished draft from 1994.

[72] J. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[73] G. T. Sullivan. Aspect-oriented programming using reflection and metaobject protocols. *Commun. ACM*, 44(10):95–97, 2001.

[74] P. Tarr and H. Ossher. Hyper/J user and installation manual. on-line at `http://www.research.ibm.com/hyperspace`, 2000.

[75] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of International Conference on Software Engineering (ICSE'99)*, 1999.

[76] D. B. Tucker and S. Krishnamurthi. A semantics for pointcuts and advice in higher-order languages. Technical Report cs0213, Brown University, 2002.

[77] D. B. Tucker and S. Krishnamurthi. Pointcuts and advice in higher-order languages. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 158–167. ACM Press, 2003.

[78] M. Van Limberghen and T. Mens. Encapsulation and composition as orthogonal operators on mixins: A solution to multiple inheritance problems. *Object-oriented Systems*, 3(1):1–30, 1996.

[79] C. Videra Lopes and G. Kiczales. *Recent Developments in AspectJ*. Springer-Verlag, 1998. LNCS 1543.

[80] K. Waldén and J.-M. Nerson. *Seamless Object-Oriented Software Architecture — Analysis and Design of Reliable Systems*. The Object-Oriented series. Prentice-Hall, Englewood Cliffs (NJ), USA, 1995.

[81] M. Wand. A semantics for advice and dynamic join points in aspect-oriented programming. *Lecture Notes In Computer Science*, 2196, 2001.

[82] N. Wirth. The programming language Pascal. *Acta Infformatica*, 1:35–63, 1971.

[83] N. Wirth. *Programming in Modula-2*. Springer Verlag, 3rd edition, 1985.

Appendix A

Test programs

Here are a selection of simple programs used to reveal the behaviour of ASPECTJ. These programs use the version 1.1 syntax.

A.1 Advice order on field accesses

```
class Test {

  public static void main(String[] args){
     System.out.println(new Test(1).go());
  }

  public String toString(){
    return "Test = " + this.n;
  }

  public Test(int n){
    this.n = n;
  }

  int n = 0;

  int go(){
     return e().n;
  }

  Test e(){
    System.out.println( "this is e, returning [" + instance +"]");
    return instance;
  }

  public static Test instance = new Test(3);

  public static void setInstance(Test t){
    instance = t;
  }

}


aspect A {

  before(): get(int Test.n) && !withincode(* *.toString()) {
```

```
      System.out.print( "about to change test instance from ["
              + Test.instance + "]");
      Test.setInstance(new Test(2));
      System.out.println(" to [" + Test.instance + "]");
  }

  after(): get(int Test.n) && !withincode(* *.toString()) {
      System.out.println("test instance is [" + Test.instance + "]");
  }

}
```

The output from this test is:

```
this is e, returning [Test = 3]
about to change test instance from [Test = 3] to [Test = 2]
test instance is [Test = 2]
3
```

which indicates that, for a field access, the target expression is evaluated before the advice, then the before advice. Evaluation of the field access does not change the state of anything. Finally the after advice runs.

A.2 Advice selection on static vs dynamic type

```
class Test2 {

  public Test2(int n){
    this.n = n;
  }

  public static void main(String[] args){
     System.out.println(new Test2(1).go());
     System.out.println(new Test2(1).go());
  }


  public String toString(){
    return "["+ this.getClass().getName() +" = " + this.n + "]";
  }

  int go(){
     return e().f().n;
  }

  int n = 0;

  /* Return the instance and change it for a new instance */
  Test2 e(){
    Test2 result = instance;
    instance = new Test4(1);
    System.out.println("This is e(), returning "
            + result + "; next time " + instance );
    return result;
  }

  public static Test2 instance = new Test3(0);

  public static void setInstance(Test2 t){
    instance = t;
```

```
  }

  public Test2 f(){
    return this;
  }

}

class Test3 extends Test2 {

  public Test3(int n){ super(n); }
}

class Test4 extends Test2 {
  public Test4(int n){ super(n); }
}


aspect B {

  before(): get(int Test2.n) && target(Test3) && !withincode(* *.toString()){
    System.out.println("Before access Test2.n on test3 object");
  }

  before(): get(int Test3.n) && !withincode(* *.toString()){
    System.out.println("Before access Test3.n");
  }

  before(): get(int Test2.n) && !withincode(* *.toString()){
    System.out.println("Before access Test2.n");
  }

}
```

The output from this test is

```
This is e(), returning [Test3 = 0]; next time [Test4 = 1]
Before access Test3.n
Before access Test2.n
0
This is e(), returning [Test4 = 1]; next time [Test4 = 1]
Before access Test2.n
1
```

which indicates that for advice on the target expression of field access expressions is evaluated before the advice, followed by the advice itself and then the access itself. The access itself has no side effects.

Advice for `get` join points are selected on the static type of the receiver expression; the dynamic type of the receiver can be tested using an additional `&& target(T)`.

A.3 Null references in target expressions

```
class NullTest {

  public static void main(String[] args){
     System.out.println(new NullTest().go());
```

```
    System.out.println(new NullTest().go());
  }

  NullTest(){
  }

  int n = 0;

  static NullTest instance = new NullTest();

  int go(){
    return NullTest.instance.n;
  }

}

aspect N {

   before(): get(int NullTest.n){
     System.out.println("Setting instance to null!");
     NullTest.instance = null;
   }

}
```

The output from this test is:

```
Setting instance to null!
0
Setting instance to null!
Exception in thread "main" java.lang.NullPointerException
        at NullTest.go(NullTest.java:16)
        at NullTest.main(NullTest.java:5)
```

The first call to `go()` completes, the second one fails because the previous one set the target to `null`. It is not possible for before advice to cause a null pointer exception.