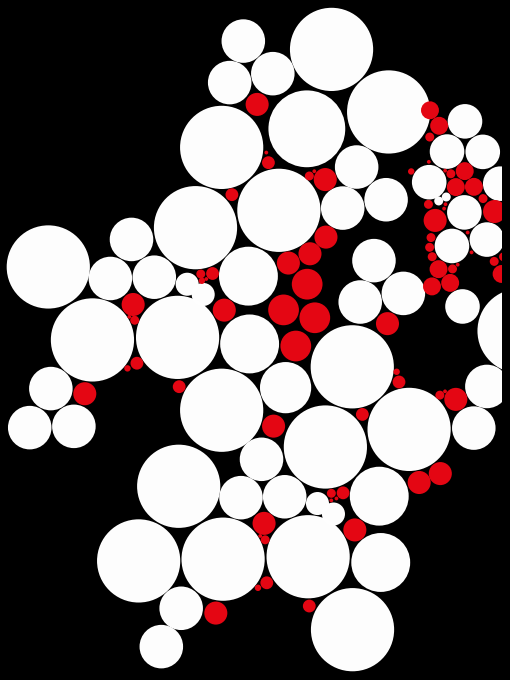


First-order function dispatch in a Java-like programming language

Steven te Brinke

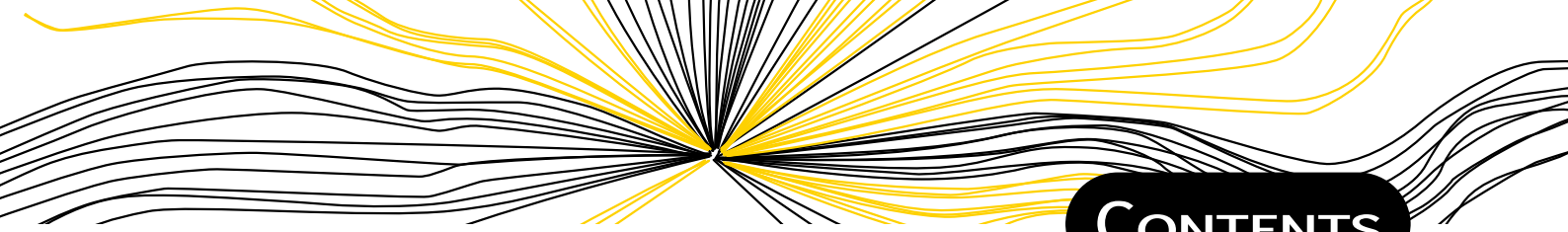
January, 2011

A dissertation submitted to the University of Twente
for the degree of Master of Science

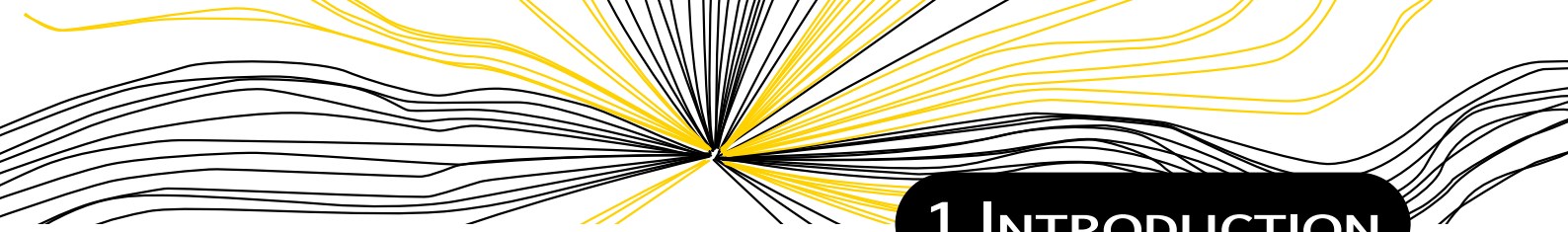
Department of Computer Science
Chair Software Engineering

Supervisors
Dr. ing. Christoph Bockisch
Dr. ir. Lodewijk Bergmans
Prof. dr. ir. Mehmet Akşit

UNIVERSITY OF TWENTE.



Contents



1 Introduction

1.1. Goals and motivation

Separation of concerns is an important concept in software development. Every software application consists of multiple concerns, examples are: what is to be computed, how the results are displayed, who is authorized to view the results, which data is logged, etc. Separation of these concerns allows a programmer to focus on a single concern, without having to consider everything at once [10]. During the software life cycle, concerns often evolve separately. Therefore, evolution scenarios are better supported when concerns are separated.

To separate concerns, software is typically decomposed in separate parts, called modules [6]. These modules are integrated in a coherent program by using composition operators. Composition operators are the language mechanisms which let programmers compose behaviour and/or data, defined as separate entities. Examples of composition operators are: function application, inheritance, delegation, pointcut-advice mechanisms, composition filters, etc. Function application allows the invocation of functionality defined separately—as a function definition—by using a call statement. Inheritance and delegation allow addressing fields and methods defined separately—as part of a super class or delegatee.

Depending on the situation, different composition operators are needed, but no language provides all possible relevant operators. When a language does not provide a composition operator with the desired compositional behaviour, workarounds have to be used by the programmer. That is, introducing new operators by writing glue code, for example as macros, libraries, frameworks or language extensions. Often, these newly introduced operators are only partially integrated with the language. Therefore, its use might require additional glue code to be written, of which an example is given in the next paragraph. Typically, such glue code solutions suffer from a lack of maintainability.

As shown in [16], delegation is a composition operator that requires additional glue code when explicit language support is lacking. In object-oriented languages that do not support explicit delegation [25], similar functionality can be simulated by

forwarding calls from one object to another. However, when forwarding a call in this way, the *self* or *this* object context changes from the object the call was originally sent to—the interface object—to the object to which the call is delegated. In contrast, in languages that support explicit delegation, the *self* context does not change when delegating an operation. This difference leads to the so-called *self problem* in languages that do not support explicit delegation [25]. That is, *self*-calls within the object to which a call is delegated, will always be handled by that object, rather than the interface object to which the call was originally sent (before being delegated). Although it is possible to work around this by passing the original interface object as an extra call parameter, this is an unsatisfactory solution. For example, this workaround necessitates changes to the interface of affected methods, which also impacts other clients of the delegatee (even if they do not use delegation but regular method invocation).

In addition, one of the motivations for using object-based languages is their built-in support for object-based encapsulation, such as the *self* object reference. If the use of *self* is restricted this limits the benefits of using a language that supports this primitive in the first place. Thus, using workarounds might not allow the expressiveness of the language to be used to its full potential.

Following the problem that every composition operator supports only a limited set of evolution scenarios, our goal is to provide a composition infrastructure for defining such operators. Because defining a new composition operator might be a non-trivial task, we would not like to put this burden on every application programmer. Therefore, it is important that composition operators can be distributed as libraries, which can be used by many applications.

Composition operators are very important in the definition of an application, they are used at many places throughout the code. This means that their execution time can have a significant influence on the execution of the application as a whole. Therefore, not only the ease of defining new operators, also the possibility for applying optimizations is important. Both concerns can be addressed by a declarative specification. The optimizations themselves are outside the scope of this thesis.

The composition infrastructure should:

- model composition operators as first-class entities
- provide a declarative way for defining composition operators
- support the definition of a variety of composition operators

- allow composition operators to be composed again
- allow multiple composition operators to be used within the same program
- reuse the technology for defining application programs in composition operator definition
- allow composition operators to be distributed as libraries

1.2. Approach

To support the composition infrastructure described in the previous section, we have designed a programming language, Co-op. Its core object language and basic syntax are inspired by the Java programming language [14]. In Co-op, method calls and field lookups are modelled as message sends [30]. These message sends can be handled using several primitive elements, from which composition operators can be constructed. This is similar to the message handling presented in [16].

The primitive elements used by Co-op are bindings and constraints. A binding selects certain messages using a selector and then rewrites these to messages which—directly or indirectly—invoke the desired behaviour. Constraints are used to express dependencies and ordering among bindings.

To allow the flexibility to deploy new composition operators at runtime, Co-op is typed dynamically. This is because composition operators, which are applied dynamically, can change the structure of a class. For example, a composition operator might change the available methods on an object. Thus, at compile time it is unknown which methods will be available during runtime, making static type checking impractical; either it cannot guarantee safety, or it will be too restrictive by disallowing uncertain cases that are actually correct.

A first Co-op prototype, partly realizing our goals, was presented by Havinga [16]. The main concepts of his work are: expressive, first class composition operators which can be composed again. Building upon his work, our main contributions are:

More declarative selector language By introducing specific selector expressions, selectors can be written in a more declarative way. This facilitates writing selectors easily and allows for more static reasoning, hence better robustness and more opportunities for optimizations. It does not reduce expressiveness, because

selector expressions can invoke any method written in the turing-complete base code.

Improved definition of constraints Havinga [16] reused the constraints specified by Nagy [26]. We introduce slightly modified versions of these constraints, increasing the expressivity of the constraints. For example, we decoupled ordering and control constraints, which means that control constraints do not imply an order, but allow the programmer to specify the desired order.

Unifying method invocation and field lookup The mechanisms for method invocation and field lookup are unified by modelling field lookup as message sends, which is how method calls were modelled already. Allowing field lookup to reuse the functionality provided for method calls improves the expressiveness of defining composition operators.

Access to dynamic message properties Message rewrite can access the result returned by executed bindings. Whenever a message send is dispatched to multiple methods, these methods can use the result returned by the previously called method.

Recursion avoidance in message selectors Because we allow using message sends in selectors, it is easy to cause infinite recursion accidentally. In the most common case, Co-op can detect and avoid this unwanted recursion.

Most important about Co-op are its concepts, i.e. the denotational semantics of the language. We consider the concrete syntax to be of lesser importance. Even though quite a bit of thought went into designing the concrete syntax, the syntax for the core object model and basic expressions are borrowed from Java [14]. We believe that many concepts of Co-op can be written using a syntax based on any object oriented language. An example of such a language, having a different concrete syntax, would be Smalltalk [12].

We have implemented a Co-op prototype in Haskell [18, 19]. Even though the prototype has a few limitations, like providing a limited number of built-in data types, it is possible to construct many composition operators using the prototype.

1.3. Results

To show that our language supports modelling a variety of composition operators, we implemented several. As a test case, we used inheritance, because it is a diverse and well classified composition operator. From the 16 properties of inheritance presented in [28], we have implemented 10. Besides these, we also implemented singletons and static method calls, showing that not only inheritance is supported. All composition operators are implemented in separate classes, allowing reuse in any program. The implementations have been tested using the prototype.

1.4. Outline

The outline of the rest of this thesis is as follows.

Chapter 2 elaborates the problem, describing the goals of Co-op in more detail.

In chapter 3 we explain the Co-op language. After discussing the structure, the different kinds of expressions are explained. The basic expressions are similar to the ones in Java, so we only discuss the differences with Java. Our main contribution is the dispatch. Therefore, we discuss dispatch in more detail, including the language features related to dispatch.

In chapter 4 we give several examples of composition operators implemented in the described prototype.

Chapter 5 discusses the implemented Co-op prototype, describing its features and limitations.

In chapter 6 our work is compared with related work.

Chapter 7 lists several features that did not make it into our language and discusses their pros and cons.

Chapter 8 concludes this thesis with a summary.

2 Problem elaboration

2.1. Approach to evaluation

In order to demonstrate the expressiveness of Co-op, we would like to show that many composition operators can be modelled in Co-op. Because every programming language has its own set of composition operators, having its own semantics, the number of available composition operators is too large to present a full overview. Inheritance is a subset of these operators, which is more clearly defined and still quite diverse [28]. Therefore, we use inheritance to test how much of it can be covered by Co-op. To show that Co-op is not limited to inheritance only, some other well-known composition operators will be shown too.

2.1.1. Inheritance

A detailed description of inheritance is presented by Taivalsaari [28]. In general, an inheritance hierarchy is a directed acyclic graph (DAG). A simple example of an inheritance DAG is shown in figure 2.1. Which DAGs represent valid inheritance hierarchies depends on the kind of inheritance. According to Taivalsaari [28], inheritance has eight main characteristics. This section will give a short description of these different characteristics of inheritance.

Inheritance type Most object oriented systems are built around *classes*, which are blueprints from which instances can be created. Another possibility are prototype based systems. These systems have no classes, but are built around *objects*. These

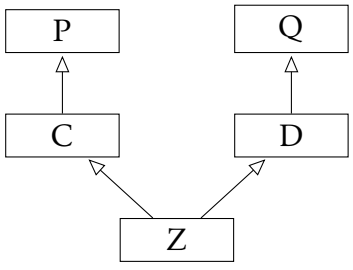
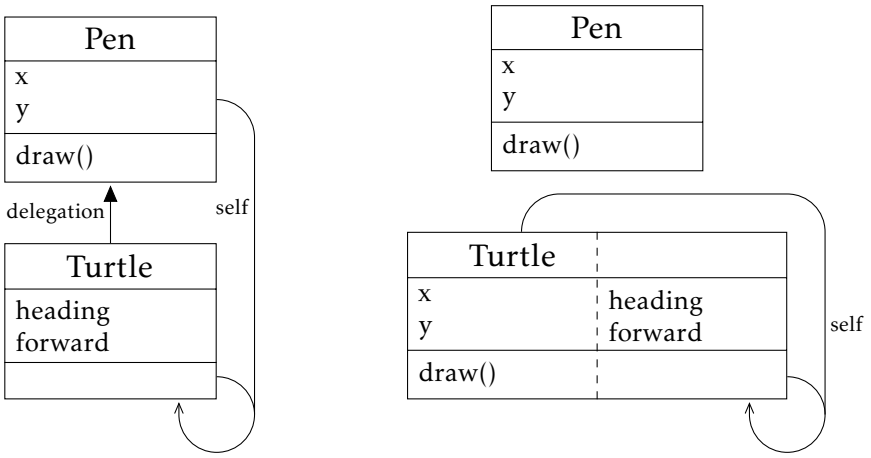


Figure 2.1.: Basic inheritance DAG



(a) Delegation (b) Concatenation

Figure 2.2.: Different sharing types

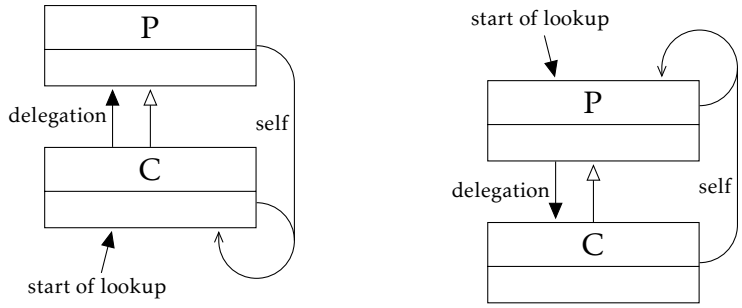
objects are prototypes providing default behaviour. Instances are created by cloning already existing prototypes. Inheritance can be achieved by modifying an individual instance, which then becomes a new prototype that can be cloned.

Sharing type In order to share common behaviour between objects, the relation between these objects must be expressed. Within computer memory, there are only two ways to express direct relationships: references and contiguity. This results in two elementary strategies for inheritance: *delegation*, which uses references, and *concatenation*, which uses contiguity. Often, common behaviour is shared by using delegation: when an object receives a message it does not understand, it will delegate this message to those objects that have been designated as its "parents". It is also possible to capture the essence of inheritance by using concatenation instead of delegation. Concatenation can be done by utilizing cloning together with the ability to add new properties to objects dynamically. Figure 2.2 illustrates the storage difference between these sharing types for a Turtle extending a Pen. An overview of the differences between the two sharing types is given in table 2.1.

Vertical combination In most main-stream object oriented languages, lookup proceeds from the most recently defined (descendant) parts of the object to the least recently defined parts (parents). This lookup order is called *descendant-driven* inheritance. Another possibility, adopted by Beta, is *parent-driven* inheritance, which

Sharing type	Delegation	Concatenation
Record combination strategy	sharing/references	copying/contiguity
Interface dependence	dependent (life-time sharing)	independent (creation-time sharing)
Inheritance DAG	preserved	flattened

Table 2.1.: Delegation versus concatenation



(a) Descendant-driven (b) Parent-driven

Figure 2.3.: Different directions of lookup

traverses the inheritance DAG in the opposite direction. An illustration of these lookup directions is given in figure 2.3. As can be seen, in both situations is the inheritance direction the same—P extends C. However, the start and direction of lookup and also the direction of the self reference are opposite.

Besides the difference in direction, vertical combination can also differ in the completion of method lookup. When the same method is defined as part of multiple classes in the inheritance DAG, calling the method can be handled in multiple ways. Simplest is disallowing this situation, which disallows any overriding, called *strict*. The most frequently adopted approach is *asymmetric*: terminating the lookup immediately after encountering the first matching method. It is also possible to execute every matching method. This *composing* completion is, for example, used by Beta.

When combining the different types of direction and completion, we have five variations of vertical combination, which are shown in table 2.2. The inheritance hierarchy for the column $x \in C \wedge x \in P$ is shown in figure 2.4. In this case, x is defined as a member of C and P, causing vertical overlap because P is an ancestor of C. The other two columns represent the situations without vertical overlap; here x only is a member of C or P respectively.

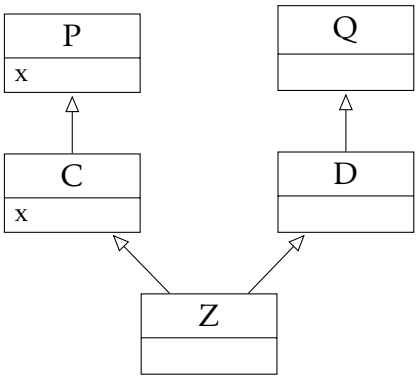


Figure 2.4.: Vertical overlapping

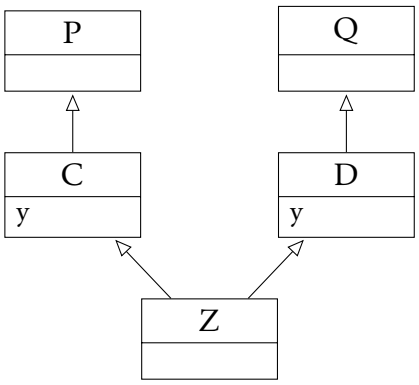


Figure 2.5.: Horizontal overlapping

C* extends *P	$x \in C \wedge x \notin P$	$x \in C \wedge x \in P$	$x \notin C \wedge x \in P$
strict	$C.x$	error	$P.x$
asymmetric descendant-driven	$C.x$	$C.x$	$P.x$
asymmetric parent-driven	$C.x$	$P.x$	$P.x$
composing descendant-driven	$C.x$	$C.x \circ P.x$	$P.x$
composing parent-driven	$C.x$	$P.x \circ C.x$	$P.x$

Table 2.2.: Variations of vertical lookup combination

Horizontal combination When the inheritance DAG contains overlapping properties along different paths, these properties are horizontal overlapping. An example of such a hierarchy is shown in figure 2.5. By using *ordered* horizontal combination, an order is defined among the different paths. This allows the execution of the first matching property. When no such order is defined, i.e. the horizontal combination is *unordered*, the presence of horizontal name collision is an error.

Multiple inheritance Single inheritance, for example provided by Java, only allows extending a single class. Multiple inheritance, for example provided by C++, also allows extending multiple classes. Thus, single inheritance limits the inheritance hierarchy to trees, whereas multiple inheritance allows the hierarchy to be any DAG.

Dynamic inheritance In general, the inheritance structure is defined statically. However, dynamic inheritance provides the ability to change parents dynamically at runtime.

Selective inheritance Not passing all information from the parent to the child is called selective inheritance. Two cases of selective inheritance can be distinguished: *selective attribute inheritance* and *selective value inheritance* [33]. Selective attribute inheritance allows inheriting only a subset of the members of the parent object. Selective value inheritance provides the ability to inherit a subset of the values of the parent object. Thus, selective value inheritance is a kind of attribute inheritance where some values are the same for the parent and child object, whereas others—the ones not using value inheritance—might differ.

Mixin inheritance Mixin inheritance provides the ability to write special mixin classes. These mixin classes are syntactically equivalent to normal classes, but have a different intention. A mixin class is defined solely for adding properties to other classes. Mixins are never instantiated and have no superclasses. By combining a mixin with a base class using multiple inheritance, the functionality of the mixin is added to the base class.

It is possible to combine every value for one characteristic with all other characteristics. However, some combinations might not be useful. For example, when using single inheritance, horizontal combination is not important, because only one parent is available. Which combination of characteristics is provided depends on the programming language. A few examples are: Smalltalk provides class-based, delegation based, asymmetric descendant-driven inheritance. Beta provides class-based, concatenation based, composing parent-driven single inheritance. Java provides class-based, delegation based, asymmetric descendant-driven single inheritance.

2.1.2. Other composition operators

Static method calls and the use of static variables are both features which can be modelled in terms of dispatch. By modelling them as composition operators, Co-op does not have to provide these as built-in concepts in order to retain its expressiveness. Thus, we will not add these concepts to our language and show that they can be modelled as composition operators.

3 Co-op language

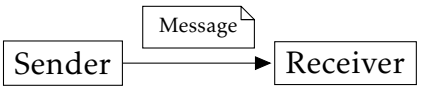
The semantics and syntax of the Co-op language will be elaborated in this chapter. The core object language and basic syntax are inspired by the Java programming language. However, we believe that many concepts of Co-op can be written using a syntax based on any object oriented language. Thus, we consider the concrete syntax of lesser importance than the semantics. Therefore, we will not elaborate all syntactic details and focus on the parts that are important for understanding the semantics.

3.1. Sending messages

In Co-op, every method call and field access is dispatched using message sends. This section will present the basic idea of message sends using a metaphor. The details of dispatch will be explained later in section 3.6.

Sending a message to a friend can be as easy as writing his address on a postcard and putting it in the postbox. Then, we assume it will be delivered to the intended receiver as shown in figure 3.1(a). In general, we do not think about the intermediate steps of the message delivery. As figure 3.1(b) shows, the message will be handled by a postal service, which is a black box to us. The postal service can influence the delivery in several ways, of which we will give a few examples:

- When the intended receiver is on holiday, the postal service might reroute the message to his holiday address.



(a) Conceptual message send

(b) Actual message send

Figure 3.1.: Sending a message

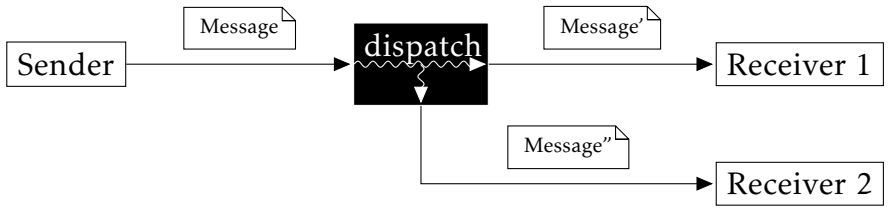


Figure 3.2.: Duplication during message send

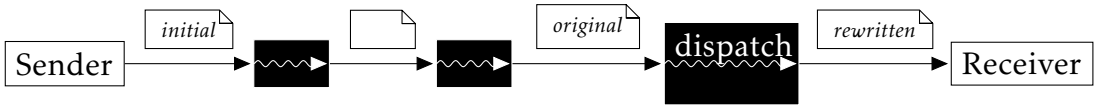


Figure 3.3.: Multiple stages during message send

- The message might be handled by another person than the intended receiver, for example his secretary.
- A security check can be done by Customs when the message crosses a country border. In such a case, the contents of the message are inspected and might be altered.

In any of these situations, there is a stakeholder that benefits from the ability to influence the message delivery process. In general, a consumer can easily subscribe to and unsubscribe from the services offered to consumers, like the rerouting to a holiday address, without thinking of the influence these subscriptions have on the process of the postal service.

In Co-op, message sends are handled similar to the process described above. The message delivery process—postal service—is called *dispatch* and can be influenced by programmers. A single influence—consumer service—is a *composition operator*. A composition operator can be activated and deactivated by an application programmer—which is similar to subscription by consumers. Ideally, an application programmer does not have to think about the dispatch once he has activated the right composition operators, in the same way we do not think about the process of the postal service.

Co-op does not have a clear distinction between the address and contents of a message—a message is more like a postcard than an enveloped letter. Any composition operator has access to the full content of the message. Thus, dispatch can influence the full message, like Customs can influence your full package.

During dispatch, a message can be copied, as shown in figure 3.2. It is also possible that the message goes through multiple stages of dispatch, shown in figure 3.3. When

we look at a specific dispatch, we say that the incoming message is the *original message* and the outgoing message is the *rewritten message*. The *initial message* is the message originally sent. Further details of dispatch will be explained in section 3.6.

3.2. Structure

The structure of a Co-op class is similar to that of a Java class. Besides variables and methods, which are well-known members in object-oriented languages, Co-op also provides a declarative syntax to specify dispatch. Dispatch declarations never specify the full dispatch process, rather are always partial—like consumer services never influence the full process of a postal service.

An example Co-op class shown in listing 3.1. This class specifies a simple string prefixer. The class can be used as follows:

```
1 var prefixer = Prefixer.new();
2 prefixer.setPrefix("[mesg] ");
3 System.println(prefixer.prefix("Hello World!"));
```

Executing this example will result in the following output:

```
[mesg] Hello World!
```

Examples of more advanced classes utilizing dispatch are shown in the next chapter.

A Co-op class can contain the following members:

- Variables: define the existence of fields like in Java.
- Methods: define methods like in Java.
- Conditions: define boolean conditions over messages.
- Bindings: define which message is bound to which dispatch.
- Constraints: define constraints between bindings.

Variable A variable declaration defines the existence of a certain field inside the class. For example, the field *prefix* is defined on line 2 of listing 3.1.

Listing 3.1: Simple Co-op class defining a string prefixer

```
1  class Prefixer {
2    var prefix;
3
4    method setPrefix(prefix) {
5      this.prefix = prefix;
6    }
7
8    method prefix(str) {
9      return this.prefix + str;
10   }
11 }
```

Method A method definition defines a method signature and its body. Two methods with a single argument are specified in listing 3.1 (on line 4 and 8).

Binding A binding is structured as follows:

```
1    BindingName = (MessageSelector)
2      {
3        // Message rewrite
4      };
```

The *BindingName* is the name of the defined binding. The *MessageSelector* selects which messages are influenced by this binding—for example only messages to a certain person. The message rewrite is used to change values of the message—for example rewriting the address to the holiday address of the selected person.

Constraint Constraints are used to express ordering and dependencies between bindings.

3.3. Typing

To allow the flexibility to deploy new composition operators at runtime, Co-op is typed dynamically. This is because composition operators, which are applied dynamically, can change the behaviour of a class. For example, a composition operator might

change the available methods on an object. Thus, at compile time it is not always known which methods will be available during runtime, making static type checking impractical; either it cannot guarantee safety, or it will be too restrictive by disallowing uncertain cases that are actually correct.

3.4. Classes

Co-op is a class-based language [32]. Thus, Co-op does not provide built-in inheritance, a concept used by object-oriented languages. Inheritance can be achieved by providing the desired dispatch, as will be explained later.

Co-op classes do not have a special constructor, but every class object has a method which returns a new instance of its type. An instance of *SomeClass* can be created as follows:

```
SomeClass.new();
```

Classes are loaded lazily in Co-op, like it is done in Java. When a class is loaded, the method *initializeClass()* will be called. This method fulfils the role of a static initializer.

In Co-op, everything is an object. Thus, Co-op does not provide separate primitive types. The keyword *null* references a single instance of the class *coop.lang.Null*.

3.5. Methods

Co-op methods are identified by their name and number of explicit parameters. Besides these properties, every method utilizes a—possibly empty—set of implicit parameters. Implicit parameters are parameters which are implicitly passed to a method. A method definition must define all used implicit parameters explicitly.

Every message property is passed to the called method implicitly, if the method declares the use of this parameter. By default, every method has the implicit parameter *this*. Which implicit parameters are used by a method can be defined by using the annotation *@ImplicitParameters* as follows:

```
1 method @ImplicitParameters(["this", "thisJoinPoint"]) someMethod() {
```

```
2     // Method body
3   }
```

Here, the method *someMethod* has two implicit parameters: *this* and *thisJoinPoint*. These two parameters can be used in the same way and have the same scope as explicit parameters. At the call side, these parameters are not given to the method, shown in the following call:

```
someObject.someMethod();
```

From the perspective of the method body, the following method definition is equal to the previous one, even though the signatures differ:

```
1   method @ImplicitParameters([]) someMethod(this, thisJoinPoint) {
2     // Method body
3   }
```

At the call side, this method differs from the previous one, because we have to pass the parameters explicitly now:

```
SomeClass.someMethod(someObject, someJoinPoint);
```

Thus, we see that implicit parameters are implicit only in the way they are passed to a method. A method definition must define all used implicit parameters explicitly. The execution of a method is only possible if all implicit parameters are available, as will be explained in more detail in section 3.6.

Side note

■ In general, the use of implicit and explicit parameters differs. However, when we look closely at the semantics, we see that the differences are small. At the call side, we often assume that the arguments become explicit parameters, and the other properties implicit ones. However, this distinction is not very clear, because a binding can easily change this behaviour.

Further, we see in the examples above that implicit parameters can be passed to a method as explicit ones. This would only require a trivial change to the message rewrite:

```
parameters = [this, thisJoinPoint]
```

We can also change the method to receive the explicit parameters as implicit ones, in the following way:

```
1   method @ImplicitParameters(["parameters"]) someMethod() {
```

```
2     var this = parameters[0];
3     var thisJoinPoint = parameters[1];
4     // Method body
5   }
```

Thus, we see that the distinction between implicit and explicit parameters mainly is syntactic sugar. However, we think that the distinction is important to allow writing software applications easily. □

The body of a method consists of the same expressions allowed by Java. The only semantic difference is that field accesses and method calls are message sends. Besides annotating elements which allow annotations in Java, it is also possible to annotate message sends. Details about these message sends are explained in the next section.

3.6. Dispatch

In Co-op, both method calls and field accesses are modelled as message sends. The dot is the message send operator, so message sends can be written in the following way:

```
this.someOtherField = this.someField;
this.someFunction();
```

This code contains three message sends, namely:

1. A message of the kind "Lookup" with name "someField" and no parameters, which commonly results in a field read of *someField*.
2. A message of the kind "Lookup" with name "someOtherField" and a single parameter, namely the value returned by message send 1, which commonly results in storing this value in field *someOtherField*.
3. A message of the kind "Call" with name "someFunction" and no parameters, which commonly results in the execution of the method *someFunction*.

As can be seen in the three message sends above, every message send defines certain properties of the message (summarized in table 3.1). Besides the properties explicitly given in these three examples, a few other properties are also defined. For example, in

property	**message send 1**	**message send 2**	**message send 3**
messageKind	"Lookup"	"Lookup"	"Call"
name	"someField"	"someOtherField"	"someFunction"
parameters	[]	[this.someField]	[]
target	this	this	this

Table 3.1.: Example message properties

all three cases the target of the message is *this*. Also, there is a sender property which is initialized to *this*. The common properties of a message are listed in table 3.2. The default properties are initialized by any message send, the other properties might be undefined, depending on the context. For example, when a message is sent from a static context—a context without a *this* reference—, the sender of the message will be undefined.

An interesting message property is the set of call annotations. Annotations provide the ability to add additional information to a message send. These annotations can be added to every message send at the call side. Annotations can be added to a call as follows:

```
this.@SomeAnnotation @OtherAnnotation("SomeValue") someFunction();
```

Besides the listed properties, a message can have any property, because the availability of properties can be influenced by message rewrites, which will be explained later. The default dispatch, explained in the next paragraph, uses certain properties of the message, as shown by the *use* column.

Side note

■ We have seen that message rewrites receive the original message and produce a rewritten message. Therefore, we have thought about providing the original message as a—possibly immutable—property of the rewritten message. In presence of recursive rewrites, bindings could access the original messages recursively, or only the initial message could be provided. However, in any case the relation between the original message and the rewritten message is defined by another rewrite, i.e. another composition operator. Thus, allowing access to the original message creates awareness of other composition operators which have been applied already.

When using properties of the original message, we can undo earlier rewrites easily. In general, this is undesired. Because we do not know the intention of the previous rewrite, we cannot decide which properties of the original message can be used safely. Thus, it is very difficult to use the original message while maintaining

name	def.	type	use	meaning (of initial value)
sender	no	*any*		originating context (null iff static)
senderType	yes	Class		type of originator
target	no	*any*	F	intended receiver (null if static)
targetType	yes	Class	M	type for method lookup
name	yes	String	MF	name of method to be invoked
parameters	yes	list of *any*	MF	explicit parameters to be passed to the method
messageKind	yes	{"Call", "Lookup"}	MF	whether the message should result in a method call or a field access
annotations		set of Annotation		annotations of the call
this	no	*any*		the original target (not used for selection, but used as implicit parameter by many methods)
thisType	yes	Class		the class of the original target (not used for selection, but used as implicit parameter by methods)
result	no	*any*		return value of last invoked method
error	no	*any*		error thrown by last invoked method

Table 3.2.: Common message properties

composability. Therefore, we have decided not to provide access to the original message. □

Default dispatch In order to achieve the common behaviour of message sends described at the beginning of this section, Co-op provides a default binding which achieves this behaviour. The default binding is always present, but only succeeds if the message addresses a declared method or field.

A message addresses a method if and only if (i) the *messageKind* is "Call" and (ii) the *targetType* has a method named *name* (iii) which has the same amount of parameters as the length of *parameters* and (iv) for this method all implicit parameters are defined. Implicit parameters are named parameters which are automatically passed to the

method when defined as message properties. For example, most methods use the implicit parameter *this*, but it is possible to require any number of implicit parameters.

A message addresses a field if and only if (i) the *messageKind* is "Lookup", (ii) the *targetType* has a field named *name* and (iii) the length of *parameters* is (a) zero—field get—or (b) one—field set. In the former case, the current value of the field will be returned. In the latter case, the value of the given parameter will be stored in the field and this value will be returned.

An overview of the properties used for addressing methods and fields in shown in table 3.2. The *use* column shows what the default dispatch uses this property for. This is for locating a method implementation (M) or a field instance (F).

We defined the default binding to be always present, but only sometimes successful. This is similar to saying a method call succeeds if and only if the called method is implemented. It is simple to imagine the default binding conforming to this definition:

```
1 binding defaultBinding = (true)
2   {
3     // The actual dispatch is performed: executing the method or accessing the field
4     // Only successful if the dispatch succeeds (i.e. method or field is defined)
5   };
```

Side note
■ Another possibility to look at method calls is by saying: a method call is only possible when the called method is declared and in that case, it is always successful. Using this view on method calls, the default binding would be defined to be always successful, but only present whenever an implementation exists. Using this definition, we would have an instance of the default binding for every method. An example for the method *System.println* is shown below.

```
1 binding defaultBinding = (targetType == System & messageKind == "Call" &
      name == "println" & arguments.length == 0)
2   {
3     // The actual dispatch is performed: executing the method or accessing the field
4     // Always successful
5   };
```

Both definitions are very similar. The only difference is that binding failure in the first definition is comparable to binding absence in the second one. It is easy to see that, by themselves, both definitions have the same execution semantics. Therefore, we conclude that binding success and presence are closely related. When

introducing other concepts of the language, like constraints, we will maintain this similarity between binding success and presence.

We saw that the first definition can be written as one generic instance, whereas the second one uses multiple parameterised instances. Because of the simplicity of understanding a single instance, we will use the first definition throughout this thesis. □

Bindings Every binding consists of a condition, the message selector, and an assignment block, the message rewrite. The message selector specifies for which messages the binding is applicable. The message rewrite creates a modified copy of the message, which is then resent. This rewritten message will be processed by all applicable bindings, exactly like the original message send. A binding is successful if and only if the message send of the rewritten message is successful.

For multi-level dispatch in general and single inheritance in particular, we can easily see that this recursive definition of success is desired. When modelling single inheritance, messages are resent to the super class recursively, as long as no implementation is present. Once an implementation has been found, the call succeeds. Thus, the call succeeds if and only if at least one super class implements the method. We will see later that more difficult situations, like Beta inheritance, can be modelled using this recursive success as well.

Every binding *b* is a special type of member of a class *C* and can access any property *p* of class *C*. This general structure is shown in the listing below. Any instance *c* of *C* has also an instance *c.b* of binding *C.b*.

```
1  class C {
2    var p;
3    binding b = (this.p) { ... };
4  }
5
6  var c = C.new();
```

We see here that we can distinguish between a binding class, *C.b*, and a binding instance, *c.b*. We talk about bindings when the distinction between binding instances and binding classes is not important.

A composition operator, which represents a concern, should be modelled by a set of binding classes. The instances of these binding classes represent the use of this

operator. For example, inheritance is modelled by several binding classes. So for every inheritance relation, instances of these binding classes are created.

Binding activation In order to use a binding, it must be activated first. The activation of a binding *c.b* can be done by calling *c.b().activate()*. When the binding is no longer needed, it can be deactivated by calling *c.b().deactivate()*.

Message sends A message send succeeds if and only if at least one binding processes the message successfully. Whereas message resends are not required to succeed, in principle message sends written by a programmer are. In general, whenever a programmer writes a method call, he expects something to happen. Thus, the call should at least be dispatched to some implementation. If all bindings fail, the call cannot be dispatched to anywhere, so an exception is thrown.

However, if the programmer would like to signal an event which does not have to be handled, he can do so by adding an annotation to the call. For example, the built-in annotation *@ImplementationOptional* allows message sends which are not handled by anyone. The working of this annotation is defined in Co-op itself, it is not part of the language itself, and will be explained in section 4.

3.7. Condition language

A condition is essentially a boolean expression over properties of the message. It can also execute message sends, for example to read fields of the object it is part of. Because in theory all message selectors, which are conditions, will be evaluated on every message send, it is important to allow optimization of these conditions. Therefore, conditions should be free of side effects. When all conditions are side effect free, the runtime environment can—without influencing the result of the program—omit evaluation of any condition if it has already identified whether this condition matches. Because message sends are allowed in conditions, Co-op can not guarantee that conditions are side-effect free. This is the responsibility of the programmer. Co-op does guarantee correct behaviour only if all conditions are side effect free.

When the evaluation of expressions is performed lazily from left to right, which is the case in many programming languages, it is up to the programmer to write

the cheap comparisons first. However, in conditions it is quite difficult to identify which operators are cheapest, because every condition only specifies a small part of the dispatch. The programmer is unaware of the total dispatch. Therefore, the programmer should not try to optimize the total dispatch, the execution environment should perform these optimizations. An example of such an optimization could be utilizing a binary decision diagram of all activated selectors, which allows evaluating as few conditions as possible.

In order to optimize, the execution environment should be able to re-order the evaluation of primitive expressions which are part of conditions. Therefore, the programmer should not make assumptions about the evaluation order based on the order in which expressions are defined. To distinguish between the semantics of Java, which guarantees an evaluation order, and unordered behaviour, we use a different syntax for the boolean *and* (`&`) and *or* (`|`) operators. These operators do not guarantee the order of evaluation, or that the evaluation is performed lazily. Because all operands are side-effect free, laziness does not influence the result, so it is an optional optimization strategy.

When unordered evaluation is used, it is desired that expressions posses the Church–Rosser property [8, 31]:

> *If an expression can be evaluated at all, it can be evaluated by consistently using normal-order evaluation. If an expression can be evaluated in several different orders, then all of these evaluation orders yield the same result.*

Boolean logic is not sufficient to guarantee this property for unordered evaluation, because dispatch can fail and we do not require that all expressions are evaluated. Thus, some evaluation orders might fail, whereas others succeed, of which an example is given in the next paragraph. To ensure that all evaluation orders yield the same result, Co-op uses ternary logic [23] for evaluating conditions. This means that we introduce an additional truth value: *unknown*. We can think of *unknown* as a sealed box containing either unambiguously true or unambiguously false. Some logical operators can yield an unambiguous result, even if they involve an *unknown* operand, as shown in table 3.3.

An example of a condition requiring ternary logic when evaluation is unordered, is the following:

```
a != null & a.length == 10
```

a	**b**	$a \wedge b$	$a \vee b$
true	true	true	true
true	false	false	true
true	unknown	unknown	true
false	true	false	true
false	false	false	false
false	unknown	false	unknown
unknown	true	unknown	true
unknown	false	false	unknown
unknown	unknown	unknown	unknown

Table 3.3.: Example of ternary operators

When *a* equals *null*, it is clear that the condition should evaluate to false and that dispatching *a.length* would fail. Using left to right short-circuit evaluation will result in $false \wedge \ldots = false$. However, we do not require left to right evaluation, so the given condition is equal to:

```
a.length == 10 & a != null
```

When evaluated in this order, *a.length* will be evaluated first, which fails. Thus, without the use of ternary logic, unordered short-circuit evaluation can yield different results for different permissible execution orders. Therefore, we define the return value of a failing dispatch to be *unknown*. Now, evaluation will result in:

$$\begin{aligned} & unknown \equiv 10 && \wedge && a \not\equiv null \\ = \quad & unknown && \wedge && a \not\equiv null \\ = \quad & unknown && \wedge && false \\ = \quad & && false. \end{aligned}$$

When looking at the unordered *or* operator, we see that the result of both orders is also the same:

```
a == null | a.length == 10
```

Using left to right short-circuit evaluation when *a* equals *null* results in $true \vee \ldots = true$. The given condition is equal to:

matching type	operators	lhs	rhs
lazy	&, |	object	object
normal	==, !=, <, >, <=, >=	object	object
annotation presence	@==, @!=	message	annotation matching expression

Table 3.4.: Binary selector operators

```
a.length == 10 | a == null
```

Evaluation of this order will result in:

$$\begin{aligned} & unknown \equiv 10 \quad \vee \quad a \equiv null \\ = \quad & unknown \quad \vee \quad a \equiv null \\ = \quad & unknown \quad \vee \quad true \\ = \quad & true. \end{aligned}$$

Besides the operators explained above, a message selector can use normal binary operators and a special annotation operator. The possible operations are listed in table 3.4. The annotation operator is provided because annotations are an important property of messages, adding annotations is the way for programmers to add additional information to a message send. In order to check for presence or absence of some annotation, use:

```
1 message @== @SomeAnnotation
2 message @!= @SomeAnnotation
```

Checking that the annotation has certain parameter values is also possible:

```
1 message @== @SomeAnnotation(name == "SomeName")
2 message @== @SomeAnnotation(priority > 4)
```

3.7.1. Recursion avoidance

It is very easy to generate infinite recursion in message selectors. For example, consider the condition:

```
messageKind == "Lookup" & target == this.child
```

This condition, used for defining inheritance, will generate a message send for reading the field *child*. In order to process that message, all message selectors, including the one just given, should be evaluated. This will result in the same message send again, causing infinite recursion.

As said in the previous paragraph, message selectors are side effect free. Thus, the evaluation of a message selectors does not change the state of the system. The behaviour of a message send is only influenced by the state of the system and the message itself. When message selectors generate messages recursively, the state remains the same. Thus, if the message being processed is generated again recursively, it will always result in infinite recursion, which is never desired.

To avoid this infinite recursion, we define that whenever during the evaluation of a message selector, a new message is generated which is exactly the same as any of the messages causing the evaluation of this selector, the result of this selector is *unknown*. This avoids the most common infinite recursion.

In the example given at the beginning of this paragraph, we see that now the given selector will be cancelled for the message send for reading the field *child*, so the field read can happen normally. This example is also shown in figure 3.4. The pieces of paper (rectangles with a folded top right corner) represent message sends. All bindings, drawn as rectangles, are matched against each message send. For all bindings, except the default binding, the applicability is defined by a selector. These selectors are shown as expression trees, where rounded rectangles depict the expressions. All constraints are shown as dashed arrows. In this figure, we have also shaded the recursion avoidance part part. The unshaded part is the desired message send, without recursion.

We see that recursion is detected when the second message is sent again. This detection—represented by the dotted arrow—cancels the message send. Therefore, the result of the message send is *unknown*. As we can see, this causes the virtualBinding to be unapplicable, allowing the defaultBinding to succeed, which is the desired behaviour.

The situation becomes more difficult when two instances, α and β, of the virtual-Binding are created. Besides self recursion, the message sends will also cause mutual

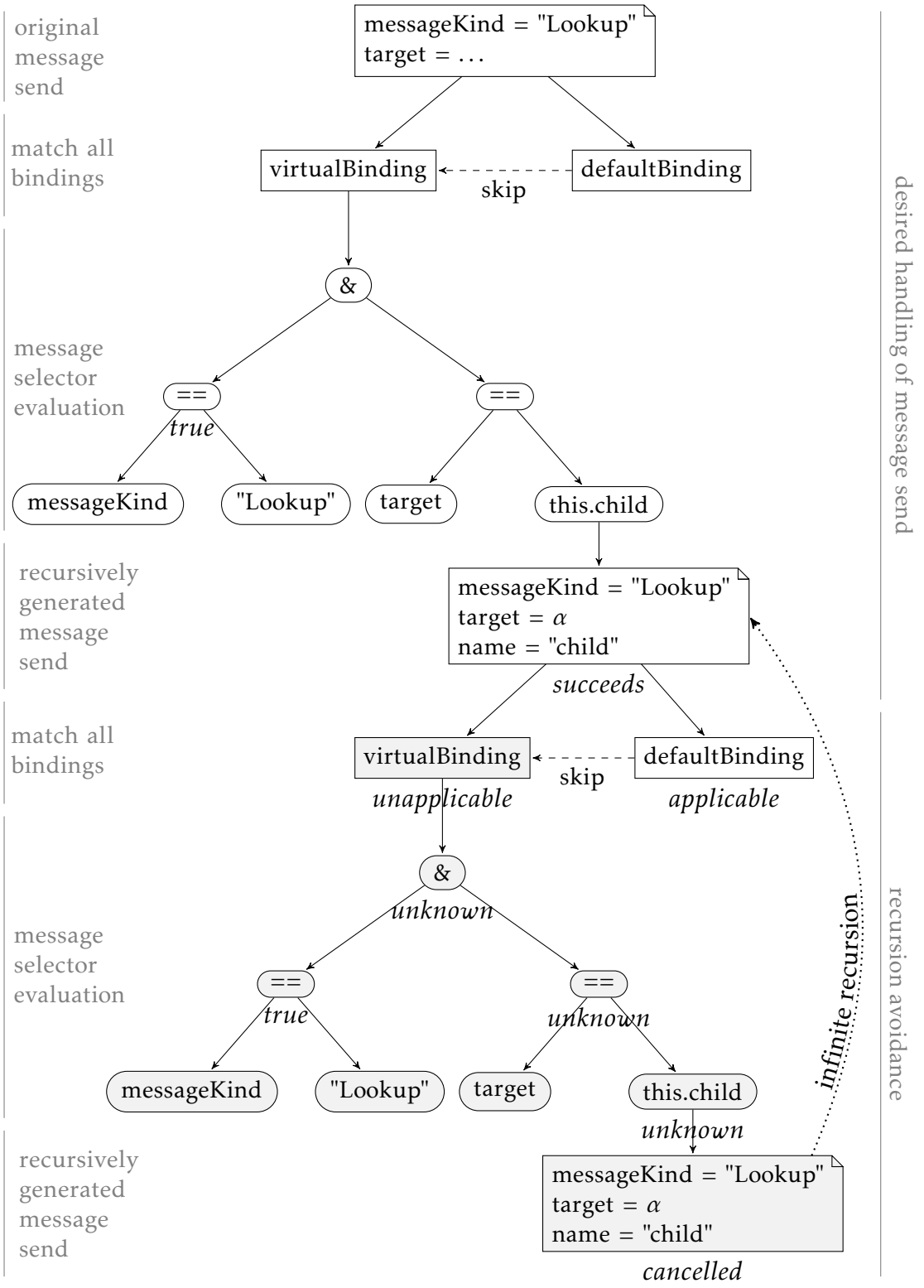


Figure 3.4.: Recursion avoidance example for `virtualBinding = (messageKind == "Lookup" & target == this.child) {...}`

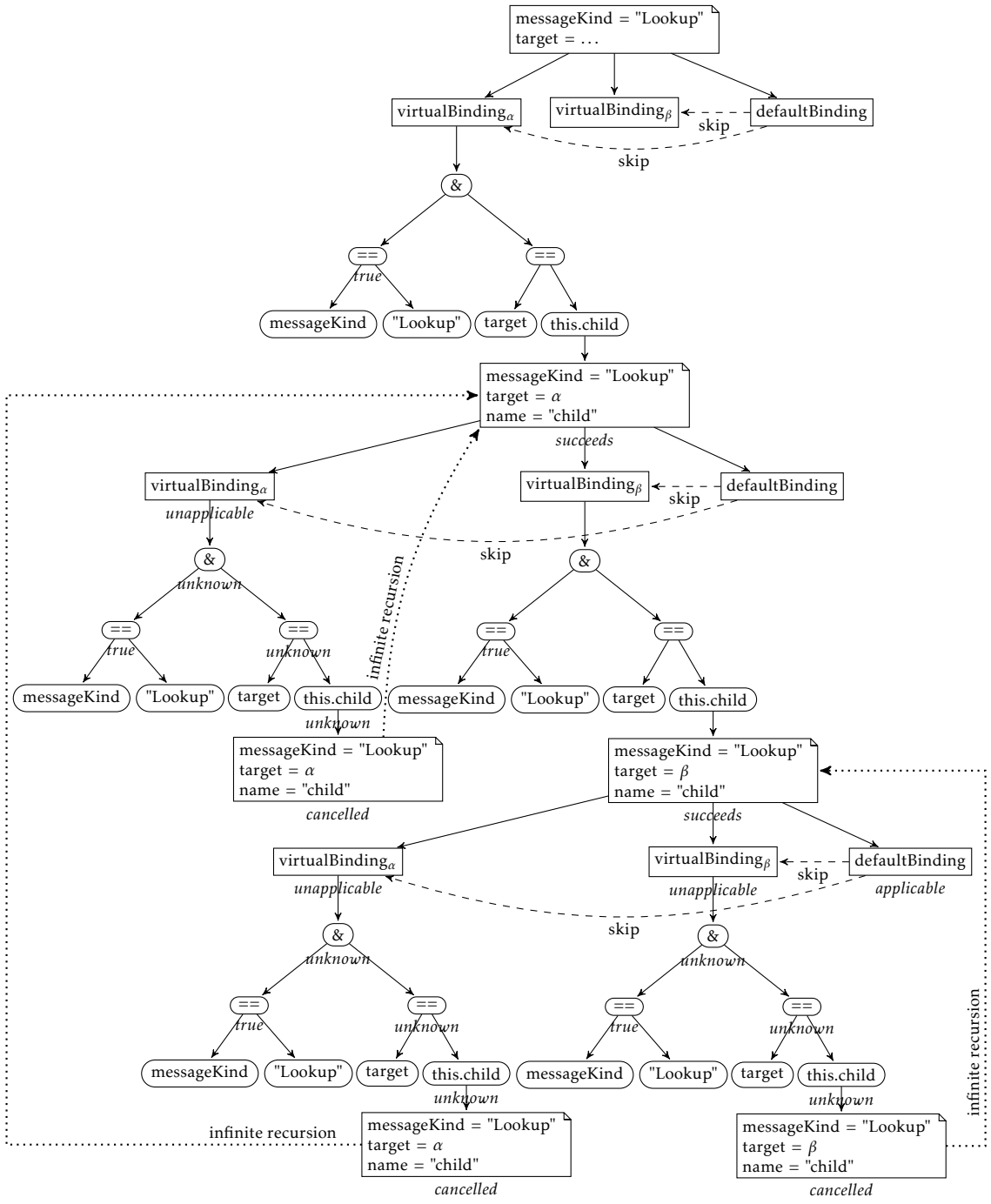


Figure 3.5.: Recursion avoidance example for two instances of `virtualBinding`

recursion. This mutual recursion can be detected and cancelled too, as shown in figure 3.5.

Side note

■ The described recursion detection can detect the most common infinite recursion,

but is restrictive in the kinds of infinitive recursion which can be detected. This is caused by the requirement that *exactly* the same message should be generated recursively. Adding message properties for debugging or tracing might break this requirement. For example, adding a counter that counts the stage of the dispatch will cause all subsequent messages to have a different stage count. Therefore, recursion avoidance will never be triggered.

In fact, only the properties which are used in conditions are important for recursion avoidance. Thus, we could define recursion avoidance to only consider these properties for message equality. Because message selectors are declarative, it is possible to generate the set of properties used in conditions and use only these properties. However, using a system wide set is not desired. This is still restrictive, because most likely many other composition operators are loaded. These operators might use other properties than the operator to which the recursion avoidance applies. Another major problem is that recursion avoidance is now influenced by which composition operators are loaded. Thus, recursion avoidance might work fine when a composition operator is tested in isolation, but might fail when used in combination with other operators. This influence between composition operators violates the desired separation of concerns.

Another possibility allowing debugging and tracing could be to allow the programmer to annotate certain properties as being unimportant for message equality. This way, it would be possible to create specific properties for debugging and tracing. Whether such a solution is desired is open for further research. □

3.8. Message rewrite language

A message rewrite creates a copy of the original message, modifies it and resends this modified message. The modification of the copy can be done using assignments:

```
mySelector = message.name;
```

The left hand side of the assignment references a property of the new message which should be set. The right hand side is an expression, in this case a message send for reading the field *name* of the original message.

For annotations, special operators are provided, which can be used as follows:

```
message @+= @MyAnnotation;
```

The left hand side is always *message*, as we are changing the annotations of the copy of the message. The right hand side is an expression which returns an annotation. In this example, we simply used an annotation literal. The possible operators are shown in table 3.5.

oper	action
@+=	Adds the given annotation to the annotations of the message.
@-=	Removes the given annotation from the annotations of the message.

Table 3.5.: Annotation operators in action selectors

Side note
■ We also thought about providing the operation @=, which replaces all annotations of the message with the given one. However, we have not seen any use case of this operator and think that its use reduces the composability of composition operators. A composition operator should only influence the annotations which are important to this operator itself. If a composition operator removes all annotations, it cannot be composed with any other operator that requires any annotation. □

Besides assigning values to properties, it is also possible to remove a property from the message:

```
remove mySelector;
```

None of these operations influence the original message. Thus, even after removing a property from the rewritten message, it will still be available on the original message.

Side note
■ Whereas it is theoretically possible to resend a message without any modification, this will never be done in practice. Resending a message unmodified will always cause infinite recursion. Therefore, we can assume that every message rewrite modifies at least one message property. □

3.9. Constraints

Multiple composition operators can apply at the same time. In order to express relations among these operators, we use constraints. Co-op models composition operators as binding classes. Therefore, constraints are expressed over binding classes.

A constraint specifies a relation between two binding classes. For example, $pre(A, B)$ specifies a precedence relation between binding A and binding B. The meaning of the different constraints will be explained in this section.

The constraints used by Co-op are based on the ones presented by Nagy [26]. Co-op provides two types of constraints: ordering and control constraints. Ordering

constraints influence the execution order only, they do not influence what will be executed. Control constraints do the complement: they express what will be executed, without influencing the execution order. Nagy does not make this distinction, his control constraints can influence the execution order and vice versa. We explicitly decoupled these two concerns, because we think it is better to address them separately.

Binding presence and success are closely related, as shown in section 3.6. Therefore, we have chosen the constraints to have the same outcome on binding absence and failure. To achieve this, we use, from the constraints presented by Nagy, as ordering constraint the soft *pre* and as control constraints the hard *cond* and hard *skip*. This results in the following constraints:

- $p\_pre(A, B)$ only allows the execution of B when A has been executed already or will not be executed at all
- $cond(A, B)$ only allows the execution of B when A is applicable
- $p\_skip(A, B)$ only allows the execution of B when A is not applicable

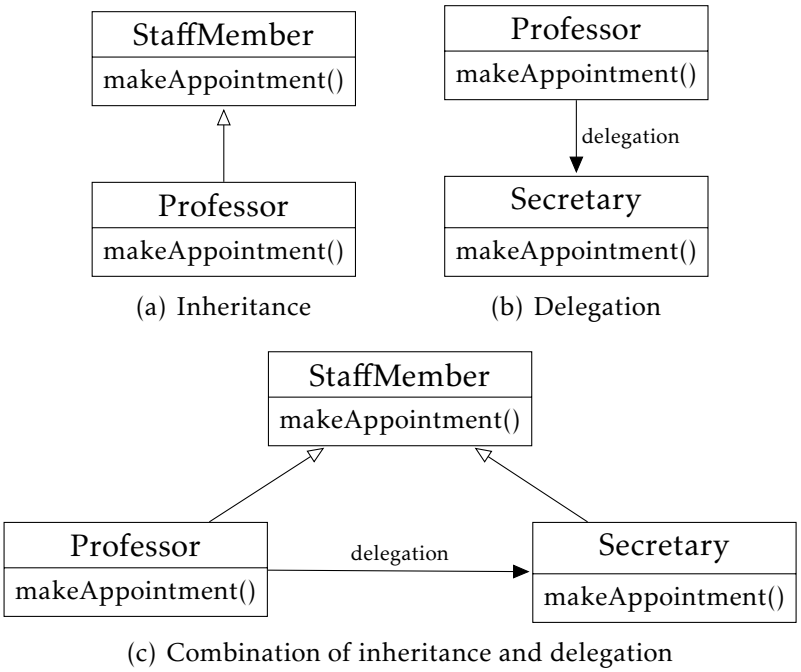
Transitivity Some constraints are not transitive. For example, p_skip is not transitive: $p\_skip(A, B) \wedge p\_skip(B, C)$ does allow the execution of A and C or the execution of B (or any subset thereof). This might not be the desired behaviour, of which we present an examples in the next few paragraphs.

A *professor* is a *staff member* at the university. Therefore, the university models a professor as a subclass of staff member, as shown in figure 3.6(a). Every staff member has the ability to make appointments, but in this example the professor has his own procedure for making appointments. In order to ensure that appointments are made using one procedure only, we use the following constraint:

```
constraint inheritanceConstraint = skip(defaultBinding, virtualBinding);
```

This ensures that whenever a professor has his own procedure for making an appointment—the defaultBinding—, the procedure of a staff member—the virtualBinding—will not be used—it is skipped.

The university has assigned a *secretary* to every *professor*, as shown in figure 3.6(b). A professor can delegate making appointments to his secretary. In fact, delegating to the secretary is preferred by the university, because the time of a professor is more expensive than that of a secretary. Therefore, we use the following constraint:



(a) Inheritance (b) Delegation

(c) Combination of inheritance and delegation

Figure 3.6.: Combination of concerns that would benefit from transitive constraints

```
constraint delegationConstraint = skip(delegationBinding, defaultBinding);
```

Whenever there is a secretary to which making an appointment can be delegated—the delegationBinding—, the professor will not try to make an appointment himself—the defaultBinding is skipped.

Both examples presented above represent a concern about professors. These concerns are described separately, but what happens when we combine them? This could, for example, result in the situations shown in figure 3.6(c), together with the constraints:

```
1 constraint inheritanceConstraint = skip(defaultBinding, virtualBinding);
2 constraint delegationConstraint = skip(delegationBinding, defaultBinding);
```

When we try to make an appointment with a professor in this situation, and *skip* is not transitive, the following happens:

1. The secretary will make an appointment for the professor.
    - The delegation constraint causes the default binding to fail.

2. Because the default binding fails, the virtual binding is not restricted by any constraints, so the procedure defined for a staff member is executed.

This means that the appointment is made twice, which is undesired. We can avoid this problem by adding the following constraint:

```
constraint transitivityConstraint = skip(delegationBinding, virtualBinding);
```

With the addition of this constraint, the behaviour would be as follows:

1. The secretary will make an appointment for the professor.
    - The delegation constraint causes the default binding to fail.
    - The transitivity constraint causes the virtual binding to fail.

Now, the appointment is made only once, which is the desired behaviour. However, we had to add a constraint between two concerns: inheritance and delegation. Because we try to separate concerns, we would like to avoid the need to specify this constraint. This constraint is exactly the constraint that transitivity would give us. Thus, using a transitive *skip* constraint can avoid the need to specify constraints between different concerns. Therefore, we think transitive constraints are desired.

Because we consider transitivity a desired property, we call the non transitive constraints *primitive*. This is denoted by using the prefix *p_* before the names of these constraints. The transitivity of the different constraints is explained below.

Cond The transitivity of *cond* follows from its definition. Assume that we have defined $cond(A, B)$ and $cond(B, C)$, then transitivity requires that $cond(A, C)$ holds. Only when A is not applicable and C is applicable, $cond(A, C)$ would not be satisfied. Therefore, we consider only situations where A is not applicable. When A is not applicable, B cannot be applicable, because of $cond(A, B)$. Similar, C cannot be applicable, due to $cond(B, C)$. All these possible situation are listed in table 3.6. Thus, when A is not applicable, C is not applicable either. This means that $cond(A, C)$ holds, so we can conclude:

- *cond* is transitive: $cond(A, B) \wedge cond(B, C) \Rightarrow cond(A, C)$

A	B	C	cond(A,C)
unapplicable	unapplicable	unapplicable	holds
unapplicable	unapplicable	absent	holds
unapplicable	absent	unapplicable	holds
unapplicable	absent	absent	holds
absent	unapplicable	unapplicable	holds
absent	unapplicable	absent	holds
absent	absent	unapplicable	holds
absent	absent	absent	holds

Table 3.6.: Possible situation when $cond(A, B)$ and $cond(B, C)$ are defined

Pre For *pre*, this is different. By itself, *p_pre* is not transitive. For example, when we define $p\_pre(A, B)$ and $p\_pre(B, C)$ and B is absent, $p\_pre(A, C)$ might not hold. A possible execution order would be: $\langle C, A \rangle$. This execution order satisfies both defined constraints, but not the constraint implied by transitivity.

However, we have defined *pre* to be transitive. This transitivity can be ensured by calculating the transitive closure over all *pre* constraints. When this is done, there is no need to add any additional kind of constraint; every *pre* constraint can be handled as a primitive one. The result is:

- *pre* is transitive: $pre(A, B) \wedge pre(B, C) \Rightarrow pre(A, C)$

Skip As we saw earlier, *p_skip* is not transitive, which might result in undesired behaviour. Therefore, we also provide a transitive version of skip.

- *skip* is transitive: $skip(A, B) \wedge skip(B, C) \Rightarrow skip(A, C)$

In total, we have defined five constraints, all listed in table 3.7. However, we have only three basic kinds of constraints, because the transitive *pre* and *skip* can be defined in terms of the primitive version of these constraints.

When there are conflicts between constraints, it can be impossible to identify which bindings should be executed in what order. An example is when $pre(A, B)$ and $pre(B, A)$ are both present. In such a case, an exception will be thrown.

Constraint	Meaning
p_pre(A,B)	Prioritizes execution of binding A over execution of binding B.
pre(A,B)	Idem, but transitive.
cond(A,B)	Allows execution of binding *B* only if binding *A* is applicable.
p_skip(A,B)	Skips execution of binding *B* if binding *A* is applicable.
skip(A,B)	Idem, but transitive.

Table 3.7.: Constraints

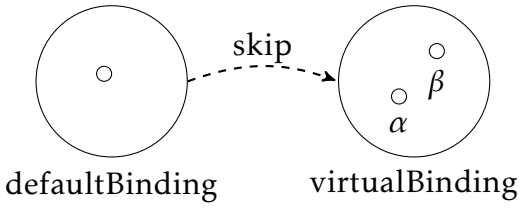


Figure 3.7.: Example of the *skip*(defaultBinding, virtualBinding) constraint in figure 3.5

3.9.1. Constraints handle binding instances

We have now seen the intention of the different constraints. However, because constraints are expressed between binding classes, they have to handle a set of binding instances at runtime. Thus, in order to define the semantics of constraints more precisely, we have to take into consideration that they work over sets.

An example of such a set can be derived from figure 3.5. In this situation, there are two instances of the virtual binding, α and β, and a single default binding. Between the two binding classes, the following constraint is defined:

```
skip(defaultBinding, virtualBinding);
```

In figure 3.5, we see this constraint and the binding instances every time all bindings are matched, which is three times. The first occurrence, directly after the first message send, is most clear. What we cannot see easily from this figure is that there are only two binding classes and is a single constraint. This is shown more clearly in figure 3.7.

Using this notion of a set of binding instances, we define the precise semantics of constraints as follows:

- $pre(A, B)$ only allows the execution of $\beta \in B$ when $\forall_{\alpha \in A}\ \alpha$ is applicable $\Rightarrow \alpha$ has been executed already

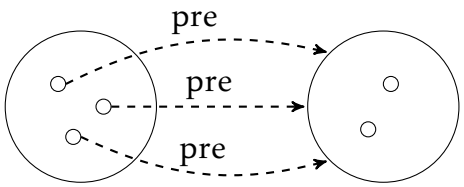


Figure 3.8.: Working of pre constraint

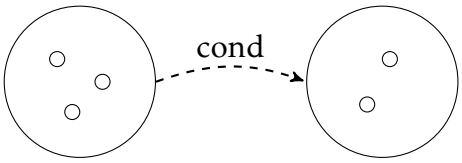


Figure 3.9.: Working of cond constraint

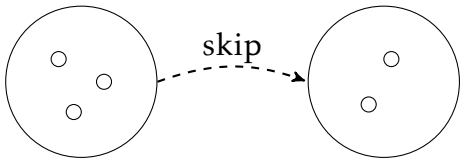


Figure 3.10.: Working of skip constraint

- $cond(A, B)$ only allows the execution of $\beta \in B$ when $\exists_{\alpha \in A}\ \alpha$ is applicable
- $skip(A, B)$ does not allow the execution of $\beta \in B$ when $\exists_{\alpha \in A}\ \alpha$ is applicable

The semantics of $pre(A, B)$ are also shown in figure 3.8. It shows that all elements from A, the left set, should have been executed before any element of B, the right set, can be executed. For $cond(A, B)$ and $skip(A, B)$ we see in figures 3.9 and 3.10 that the applicability of any element from A is enough to respectively allow or cancel the execution of all elements of B.

The *cond* and *skip* constraint both use existential quantification—$\exists_{\alpha \in A}\ \alpha$ is applicable—rather than universal quantification—$\forall_{\alpha \in A}\ \alpha$ is applicable. This is motivated by the similarity between binding presence and success. If we consider the shaded circle in figure 3.11 an unsuccessful bindings, both sets are similar. Therefore, we would like the constraints to have the same result on both sets. However, we see that only for the existential qualifier, the outcome is true regardless the used set. The universal qualifier will yield false for the left and true for the right set. Therefore, the existential qualifier is used for the definition of these constraints.

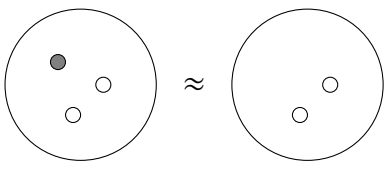


Figure 3.11.: Binding instance applicability

3.9.2. Constraint activation

Only activated constraints are used by the runtime. Constraint activation is the same as binding activation. An example is the following the constraint:

```
constraint inheritanceConstraint = skip(defaultBinding, virtualBinding);
```

This constraint can be activate by calling:

```
inheritanceConstraint().activate();
```

Deactivation works similar:

```
inheritanceConstraint().deactivate();
```

In general, it is desired to activate the constraints before activating the bindings of a composition operator. Constraints have no influence on the runtime when the bindings they are specified on are not activated. However, bindings might cause unintended behaviour when activated without activation of the desired constraints.

4 Composition operator construction

It is possible to define many different composition operators in Co-op. In this chapter, we will explain three of these composition operators, to show how composition operators can be created.

4.1. Static

Static method calls are not a built-in concept of Co-op. However, it is easy to write a static method call, because Co-op will load the class *Static* listed in section B.1 before executing any program. This class defines the composition operator for static method calls, which will be explained in this section. Some relevant code snippets from section B.1 are also listed in this section. When we do so, we use the line numbers from the full listing, so it is easy to locate the same code in the appendix.

Consider that we have the following class, defining a single method, *someMethod*, which behaves like a static method:

```
1  class SomeClass {
2    method @ImplicitParameters([]) someMethod() {
3      // Some method body
4    }
5  }
```

Now we can execute *someMethod* in the following way:

```
SomeClass.someMethod();
```

This call will result in the message send shown in figure 4.1 (irrelevant message properties are omitted). The different steps in the dispatch are explained below.

The original call has target *SomeClass*, which is an instance of *coop.lang.Class* representing *SomeClass*. Therefore, the default binding will try to locate an implementation of the method *someMethod* in *coop.lang.Class*, which fails. Thus, the default binding fails.

The class *Static* defines a single binding:

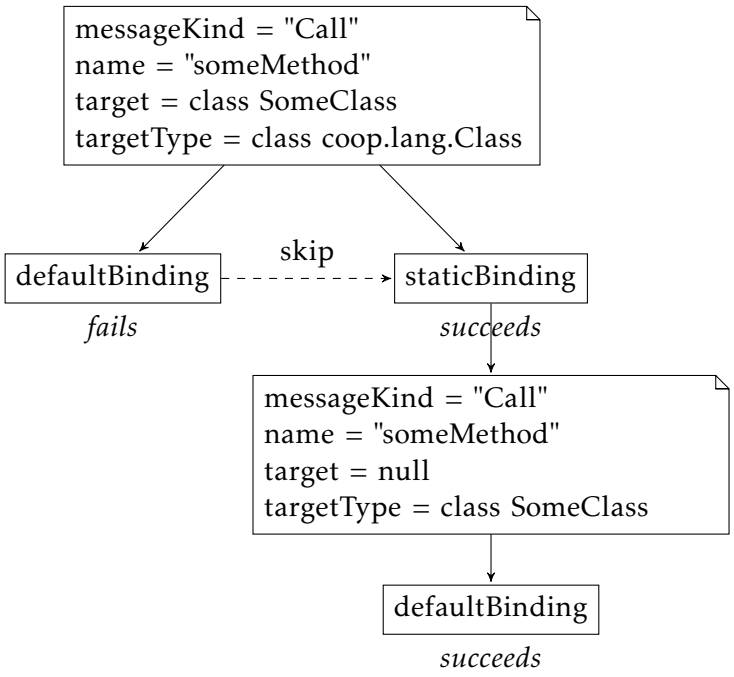


Figure 4.1.: Static dispatch example for the class shown in section B.1

```
9  binding staticBinding = (targetType == Class & messageKind == "Call" & target
       != null) {
10   targetType = target;
11   target = null;
12   remove this;
13 }
```

This *staticBinding* matches the first message and rewrites it to the second one shown in figure 4.1. We see that the target type has changed to *SomeClass*, which contains an implementation for *someMethod*. Therefore, the default binding succeeds here.

We also see that a skip constraint is present between the default and static binding:

```
15 constraint defaultBeforeStatic = skip(System.defaultBinding, staticBinding);
```

In this example, the constraint does not do anything, because the default binding fails. We can see the working of the constraint when we create a new instance of *SomeClass*:

```
SomeClass.new();
```

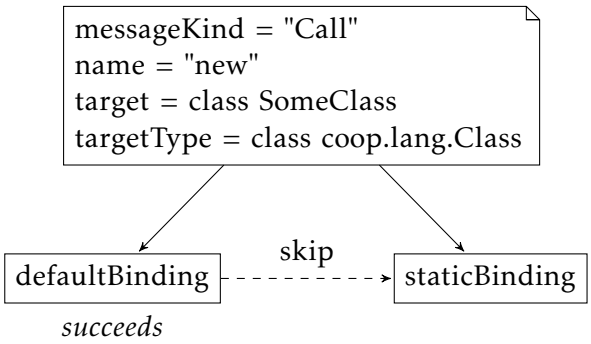


Figure 4.2.: Construction of an instance of *SomeClass*

The method *new* is implemented in *coop.lang.Class*, so the default binding succeeds. Because the default binding is successful, the static binding is skipped. Therefore, the evaluation of the static binding can be omitted, as shown in figure 4.2.

4.2. Event notification

It can be desired to create an event which does not necessarily have to be handled. For example, the message *initializeClass*, which is implicitly called during class loading, may be ignored if a class does not require initialization. In order to achieve this behaviour, Co-op loads the class shown in section B.2 by default. This class defines a single binding and constraint and the method *noDispatch* used by the binding:

```
6  binding implementationOptional = (messageKind == "Call" & message @==
       @ImplementationOptional) {
7    targetType = DispatchOptional;
8    name = "noDispatch";
9    parameters = [];
10   message @-= @ImplementationOptional;
11 }
12
13 constraint whenDefaultFails = skip(System.defaultBinding,
       implementationOptional);
14
15 method @ImplicitParameters([]) noDispatch() {
16   // Nothing to be done when no dispatch is required
17 }
```

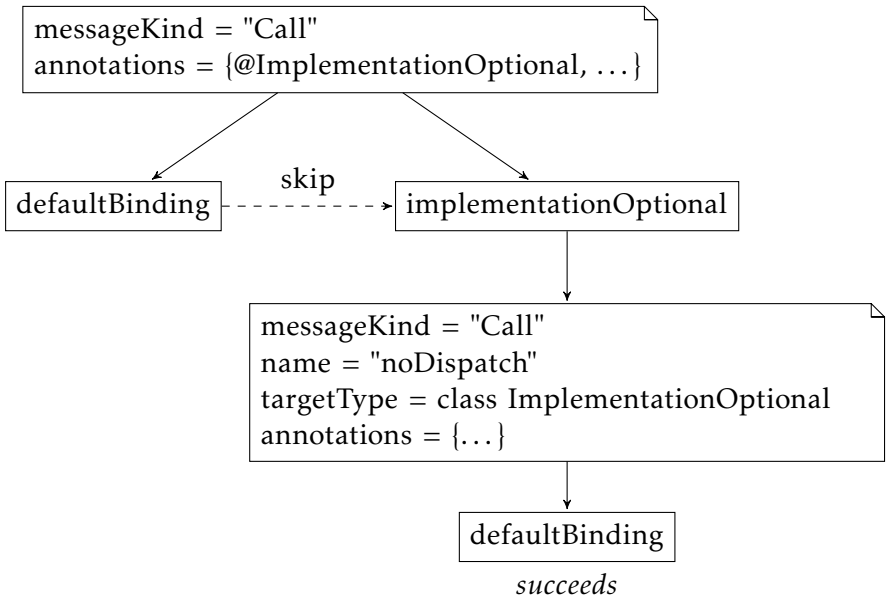


Figure 4.3.: Dispatch when a call is annotated with *@ImplementationOptional*

Also, any call to *initializeClass* is annotated with *@ImplementationOptional*:

```
SomeClass.@ImplementationOptional initializeClass();
```

When a call is annotated with *@ImplementationOptional*, the dispatch will look as shown in figure 4.3. Thus, if the default binding fails, the *implementationOptional* binding will call the empty method *noDispatch*. Therefore, the call succeeds and nothing happens (unless some other dispatch handles this call, of course).

4.3. Multiple inheritance

A more interesting example is multiple inheritance. As shown in section 2, many types of inheritance exist. In this chapter, we will show an excerpt of our inheritance implementation, which is the part that models class-based, delegation based, asymmetric descendant-driven multiple inheritance only.

Both method and field lookup are influenced by inheritance. The influence on method lookup will be explained in the next section, the influence on field lookup in the section thereafter.

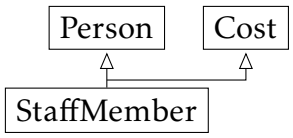


Figure 4.4.: Staff member class diagram

4.3.1. Method inheritance

The influence of inheritance on method lookup is defined by the class *MethodInheritance*. This class specifies multiple types of inheritance, of which multiple inheritance will be explained in this section. The code defining multiple inheritance is fully listed in section B.3.

Creating inheritance relations The class *MethodInheritance* specifies multiple types of inheritance, of which only multiple inheritance is listed here. The class *MultipleInheritance*, is defined as follows:

```
7  class MultipleInheritance {
8    // Call this method for every parent type
9    method @ImplicitParameters([]) inherits(childType, parentType) {
10     MethodInheritance.new().createDispatch(childType, parentType);
11   }
12 }
```

The class is a factory to create multiple inheritance easily. It allows using multiple inheritance in the following way:

```
MultipleInheritance.inherits(StaffMember, Person);
MultipleInheritance.inherits(StaffMember, Cost);
```

These two lines of code specify that *StaffMember* inherits from *Person* as well as *Cost*, which is also show in figure 4.4. For each inheritance relation between two classes, an instance of *MethodInheritance* is created using this call. This instance stores the child and parent type in the following variables:

```
14 var childType;
15 var parentType;
```

Virtual method calls The following binding specifies virtual method lookup:

```
18 binding virtualBinding = (messageKind == "Call" & targetType ==
       this.childType) {
19   targetType = this.parentType;
20 }
```

When the lookup type of a message is equal to the child type of an inheritance relation, the method is also invoked on its parent class by this *virtualBinding*. This means that two different bindings match the same message: the default binding and the *virtualBinding*. This would lead to the execution of two methods when both the parent and child type implement a method with the specified name, which is the composing vertical combination explained in section 2.1.1. However, we would like to have asymmetric vertical combination. Therefore, we add a constraint between the default binding and the *virtualBinding*:

```
0 constraint defaultSkipsVirtual = skip(System.defaultBinding, virtualBinding);
```

This constraint specifies that if the default binding succeeds, any *virtualBinding* is skipped. This way, method definitions in the child type effectively override those in the parent type, while methods not defined in the child type are forwarded to the parent type, as intended by asymmetric descendant-driven inheritance.

Note that the dispatch might be a multi-stage process: if the parent type is involved in another inheritance relation, the rewritten message may be rewritten again. These rewrites continue until the message reaches an ancestor implementing the method or until the root of the inheritance DAG is reached and the message send fails.

Super method calls Many object-oriented languages provide special keywords to refer to related classes in the program structure. For example, in Java the keyword *super* can be used by subclasses to call method implementations defined in an ancestor class, even if the subclass overrides the original method implementation. However, the keyword *super* does not refer to an object in the same sense as *this* can be considered a normal object. Keywords such as *super* are rather an indication to the message dispatch mechanism of a language that it should deviate from its normal procedure: instead, the lookup procedure followed to dispatch the message should be started at the parent of the class in which the keyword occurs.

In Co-op, annotations are used to influence dispatch. Therefore, instead of introducing a keyword—such as *super*—, we model the desired behaviour using the call annotation *@Super*. For example, this annotation can be used as follows:

```
this.@Super makeAppointment(date);
```

The *superBinding* implements this behaviour:

```
24 binding superBinding = (messageKind == "Call" & message @== @Super &
       targetType == this.childType) {
25   targetType = this.parentType;
26   message @–= @Super;
27 }
```

We see that the search for an implementation starts at the parent of the sender. Also, the annotation *@Super* is removed, so the search proceeds as a virtual method call described before.

Side note
■ If the annotation *@Super* was not removed by the *superBinding*, the rewritten call would be treated as a super call again. This process would continue until the topmost class in the inheritance DAG is reached and the implementation of that class is executed. Therefore, it is necessary to remove the annotation *@Super*. □

Again, it is possible that both the default and *superBinding* match. In this case, it is clear that only the *superBinding* should be executed, which is specified by the following constraint:

```
37 constraint superSkipsDefault = skip(superBinding, System.defaultBinding);
```

Side note
■ Multiple inheritance allows a class to have multiple parents. In such a situation, using *@Super* is not sufficient. It should be possible to address a specific parent. It is easy to achieve such behaviour. To do so, we give the annotation *@Super* a parameter that identifies the parent that should be called, for example: *@Super("DesiredParent")*. In order to rewrite to the defined parent only, we can change the message selector of the *superBinding* by appending the following code to it:

```
& message.annotations.get("Super").value == this.parentType.getName()
```

Now, only the *superBinding* for the chosen parent is applicable. Thus, this would allow calling a specific parent. □

Super constructor calls When an instance of a child class is created, it is often desired that the constructor of the super class is called. In Co-op, inheritance is realised by delegation, which means that the fields of ancestors are not stored in the child itself. Therefore, it is really necessary to create its ancestors when a child is constructed. This is done by creating the parent after the child has been created.

```
30 binding superConstructorBinding = (messageKind == "Call" & name == "new" &
       target == this.childType) {
31   targetType = MethodInheritance;
32   target = null;
33   name = "createParent";
34   parameters = [result, this.parentType]; // The newly created object is "result"
35 }
```

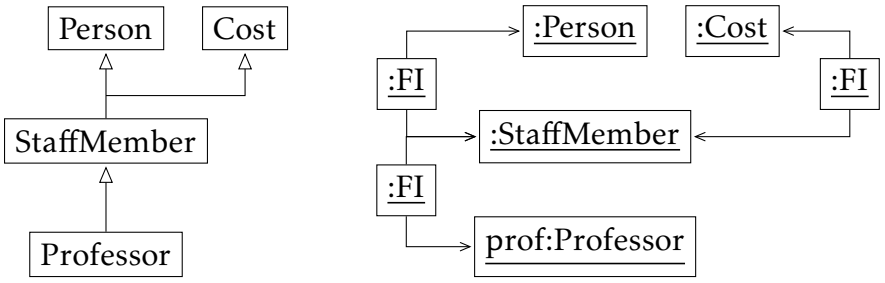
The *superConstructorBinding*, shown above, defines calling the super constructor when the child is constructed. This is achieved by calling the method *createParent*, shown below, with as arguments the newly created child—the result of the constructor called before—and the type of the parent. The method *createParent* creates a field inheritance relation between the child and a newly constructed parent instance. This relation ensures that fields of any ancestor are accessible through the child. Further, *createParent* returns the child again, ensuring that the original constructor call returns the constructed object.

```
45 method @ImplicitParameters([]) createParent(child, parentType) {
46   FieldInheritance.new().createDispatch(child, parentType.new());
47   return child;
48 }
```

We see that the *superConstructorBinding* uses the result of the original constructor. Of course, this result is only available after the original constructor executed successfully. This execution requirement is specified using two constraints:

```
42 constraint superConstructorConstraint = cond(System.defaultBinding,
       superConstructorBinding);
43 constraint superConstructorOrderConstraint = pre(System.defaultBinding,
       superConstructorBinding);
```

Using the described creation strategy, the inheritance structure shown in figure 4.5(a) will create the object structure shown in figure 4.5(b). An instance of



(a) Class diagram (b) Object diagram

Figure 4.5.: Class and object diagram of a professor

every ancestor is created. These ancestors are linked to the child by instances of *FieldInheritance* (in the diagram abbreviated to *FI*). The creation of all these objects is transparent to the programmer, because only a reference to the child *prof* is returned. The other objects are not directly accessible.

4.3.2. Field inheritance

The influence of inheritance on field lookup is similar to the influence on method lookup. The difference is that for field lookup an instance is required, whereas for method lookup knowing the type defining a method is sufficient. The implementation of field inheritance is shown in section B.4.

Virtual and super field lookup is very similar to virtual and super method lookup. The same bindings and constraints are present, but all bindings also rewrite the target of the message. Therefore, these bindings work for methods as well as fields, whereas the ones presented earlier work for methods only.

4.4. Other kinds of inheritance

In chapter 2 we showed that many types of inheritance exist. In the previous section, only the implementation of one kind of inheritance is shown: class-based, delegation-based, unordered asymmetric descendant-driven multiple inheritance. It is possible to implement many other types of inheritance too. The classification of Taivalsaari [28] lists 16 different properties of inheritance. From these 16 properties, we have implemented 12 in Co-op. That only leaves 4 properties which are not implemented:

sharing by concatenation, ordered horizontal combination and selective attribute and value inheritance. We will give a short explanation about these properties in the next few paragraphs.

Concatenation The sharing type identifies how relations are expressed in computer memory. However, Co-op does not provide access to the computer at a low level. Thus, it is not possible to influence the memory directly. Therefore, it is not possible to use concatenation as sharing type. We also saw that the behavioural differences between delegation and concatenation are small. It might be possible to simulate the exact behaviour of concatenation using delegation. However, we think that this would not provide any real benefits, therefore we did not try to implement sharing by concatenation.

Ordered horizontal combination To model ordered horizontal combination, it should be possible to express an order between different parents of the same class. Using constraints to express this order might seem a nice solution. However, constraints are not expressive enough. Constraints are designed to express relations among different composition operators, not within the same composition operator. We could improve the expressiveness of constraints to allow expressing relations within a composition operator too, but we have not encountered any other situation that would benefit from this addition.

Another possibility for ordering the parents could be assigning an ordinal to every parent. Then, we could give control to a specific parent explicitly. This could be done by using an annotation, for example *@Parent(ordinal)*. Then, when the default binding fails, the annotation *@Parent(0)* would be attached to the message. If parent number zero is unable to handle the message, it rewrites the annotation to *@Parent(1)*. This process continues until a parent handles the message or no parent is left and the message send fails. This algorithm should work with the current definition of Co-op. However, numbers are not implemented in the prototype, therefore it will not be possible to test the implementation.

Selective inheritance Both selective attribute and selective value inheritance should not be very difficult to implement. Due to lack of time, we did not try to implement any of these and leave them as an exercise for the reader.

Property	Co-op	Java	Beta	C++	Smalltalk
inheritance type					
– class-based	✓	✓	✓	✓	✓
– object-based (prototype-based)	✓				
sharing type					
– delegation	✓	✓		✓	✓
– concatenation			✓		
vertical combination					
– strict	✓				
– asymmetric descendant-driven	✓	✓		✓	✓
– asymmetric parent-driven	✓				
– composing descendant-driven	✓				
– composing parent-driven	✓		✓		
horizontal combination					
– ordered				✓	
– unordered	✓				
multiple inheritance	✓			✓	
dynamic inheritance	✓				
selective inheritance					
– selective value inheritance					
– selective attribute inheritance					
mixin inheritance	✓				

Table 4.1.: Supported inheritance properties by different languages

An overview of language support for the different properties of inheritance is shown in table 4.1. We see that Co-op already supports much more than other mainstream languages. Further, for Co-op only the properties that are currently implemented are listed. Because the properties are defined in the language itself, it will be possible to implement even more properties, as we saw in the previous paragraphs.

5 Co-op prototype

We have implemented a prototype of Co-op. This prototype is implemented in Haskell [18, 19]. All example classes presented in this paper have been tested using the prototype. Therefore, we can conclude that the presented language also works in practice.

The prototype provides a small set of built-in classes, namely: Null, Boolean, String, Array, Class, Constraint, Binding, System. A Class has a method getName() to retrieve the class name and a method new() to create a new instance of the class, a Constraint and Binding have a method activate() and a method deactivate(). The *lib* directory contains the standard classes. It also contains interfaces, which have a purely descriptive use only. Interfaces cannot be loaded by the interpreter. The classes for which an interface definition is present are defined in Haskell code.

Debugging is supported in the prototype by adding the annotation *@Debug* to a message send. For every message send containing this annotation, the prototype will generate a diagram of the dispatch as shown in figure 5.1. These diagrams are similar to the diagrams we presented in this paper. Often, the generated diagrams are less clear because they are polluted with too much information and the layout is sub optimal. However, even though the diagrams are still a bit primitive, they provide quite valuable debugging information already. In the diagrams, the evaluation order is shown by dashed blue arrows and colours show the applicability of bindings.

The given example uses the class diagram of figure 5.2. The presented call is the following:

```
prof.@Debug makeAppointment("next␣Friday");
```

The variable *prof* contains an instance of *Professor*. We see that the call can be dispatched both to *StaffMember* and *Person*. However, the dispatch to *Person* is cancelled by a skip constraint, resulting in the desired dispatch at *StaffMember*.

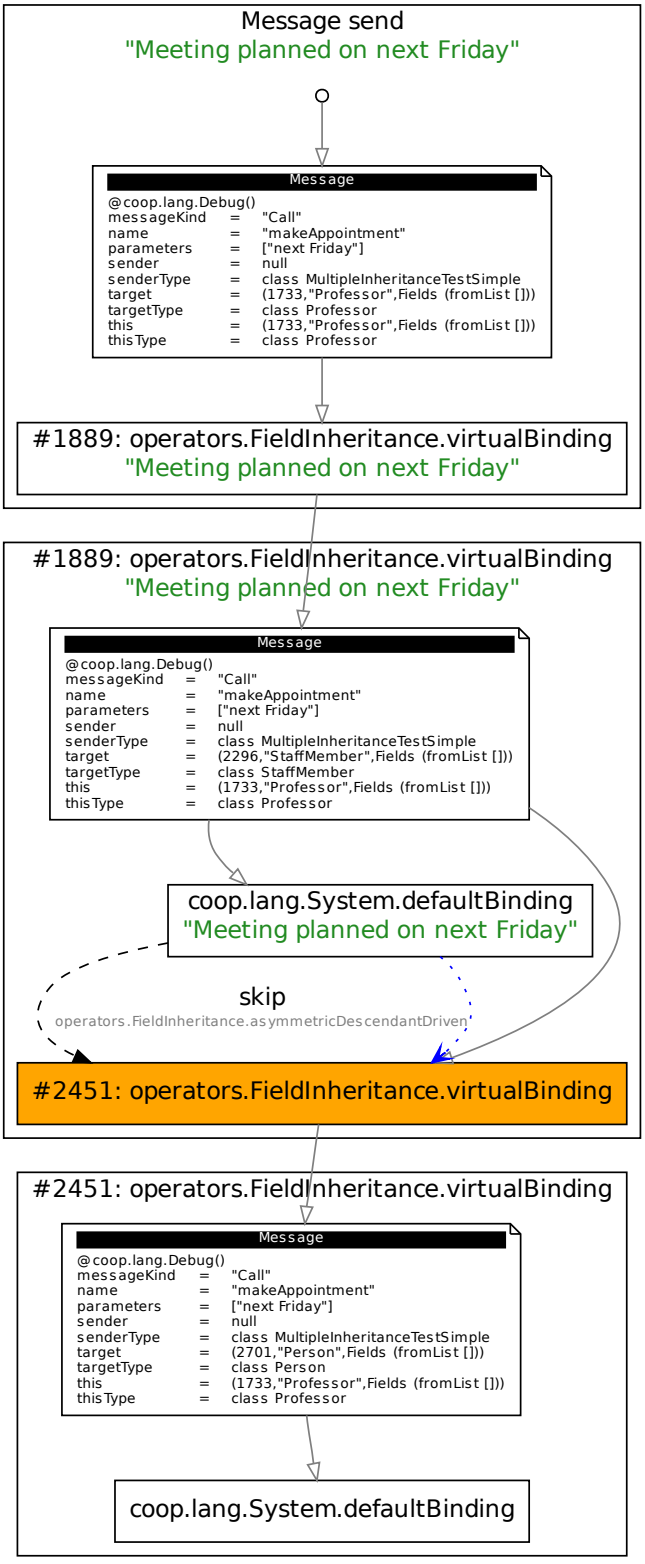


Figure 5.1.: Debugging of a function call using inheritance

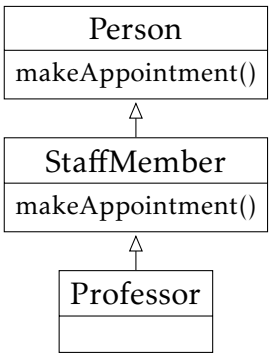


Figure 5.2.: Class diagram of dispatch shown in figure 5.1

5.1. Limitations

The implemented prototype has some limitations. All core concepts are implemented to show that the language works, but some other features are left out. We will describe the limitations of the prototype in this section.

Built-in classes

Only a limited number of built-in data types is provided. The supported data types are Boolean, String and Object. Arrays are available too, but their usage is limited. Array creation is supported, but accessing elements is not possible, because integers are not available. Arrays are used for passing explicit parameters to functions. When used in this way, the elements can be accessed because a name is assigned to each element.

Error reporting

Error reporting is limited. Some situations might give descriptive error messages, whereas others just return Haskell errors. Therefore, debugging can be difficult. The best way to handle errors is to check your code twice, especially for type errors and whether all variables are defined. If that is insufficient, you might have to use the Haskell debugger, which will not result in very user friendly debugging.

Single pass compilation

Our compiler runs a single pass only. Therefore, inner classes cannot be resolved from the outer class. This is not the desired behaviour, but the workaround is easy: create an import statement for every inner class using its full qualified name. This import statement ensures that the outer class is aware of the existence of the inner classes.

Message properties

It should be possible to remove message properties. However, the prototype cannot handle undefined message properties properly. Therefore, it will return *null* whenever an undefined property is read.

5.2. Performance

The prototype is designed to show the semantics of Co-op, not to be an efficient implementation of the Co-op runtime. Therefore, the prototype might use more memory and processing power than expected when executing any of the given examples. We have identified two main sources for high resource usage: memory management and constraint evaluation order. We will briefly explain these two deficiencies. Besides these deficiencies, we believe many other optimization possibilities exist. Whether Co-op can achieve the same runtime performance as current mainstream programming languages is open for further research.

Memory management The Co-op runtime does not have a garbage collector and does not provide the programmer with the ability to finalize objects which are used no longer. Therefore, every object which is created once, will remain in memory forever, causing excessive memory usage. The only exception is that the prototype finalizes messages when these are not used any more, which is important because bindings cause excessive message rewrites. However, this finalization does not apply to message properties. Thus, the application of bindings still causes an increase in memory usage.

Constraint evaluation order Binding applicability and constraint evaluation are defined as two separate concerns. Therefore, they are implemented separately: binding

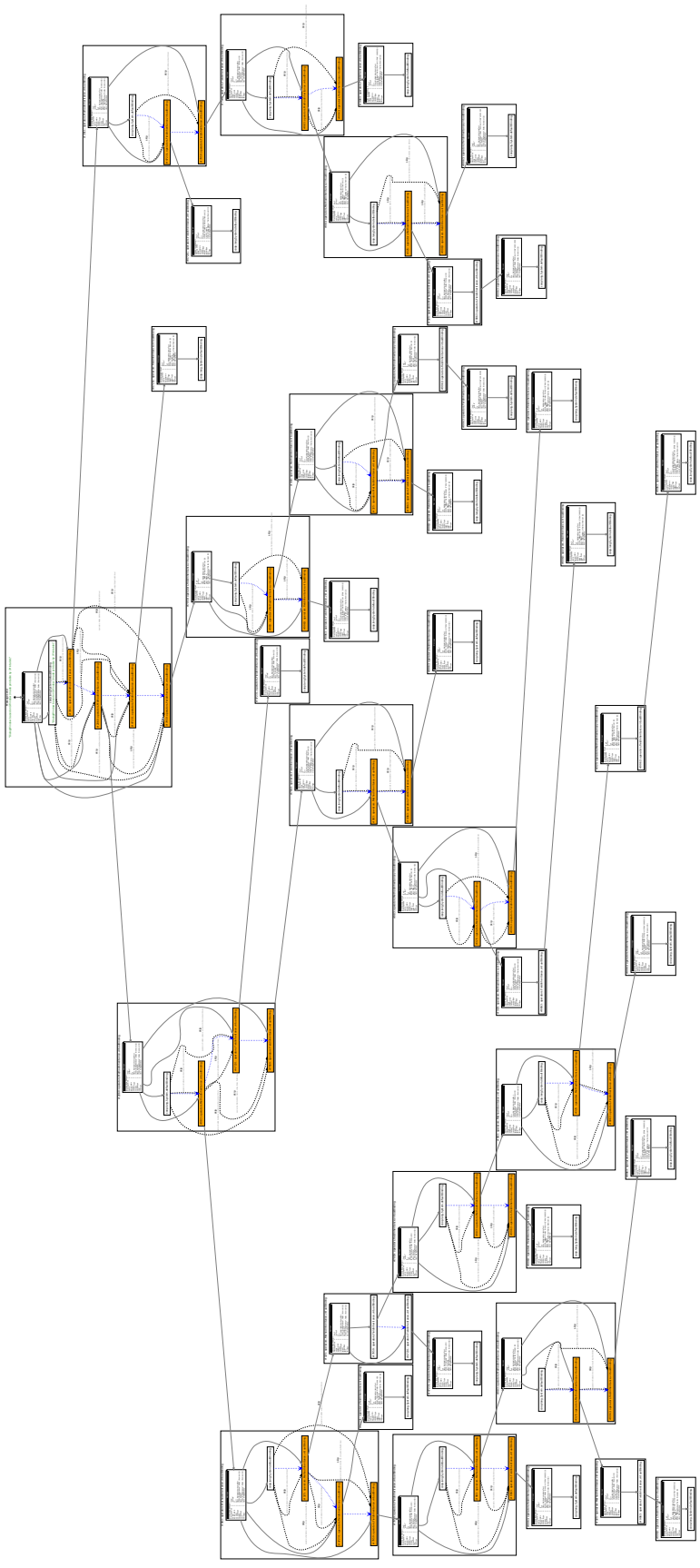


Figure 5.3.: Difficult function call using multiple inheritance

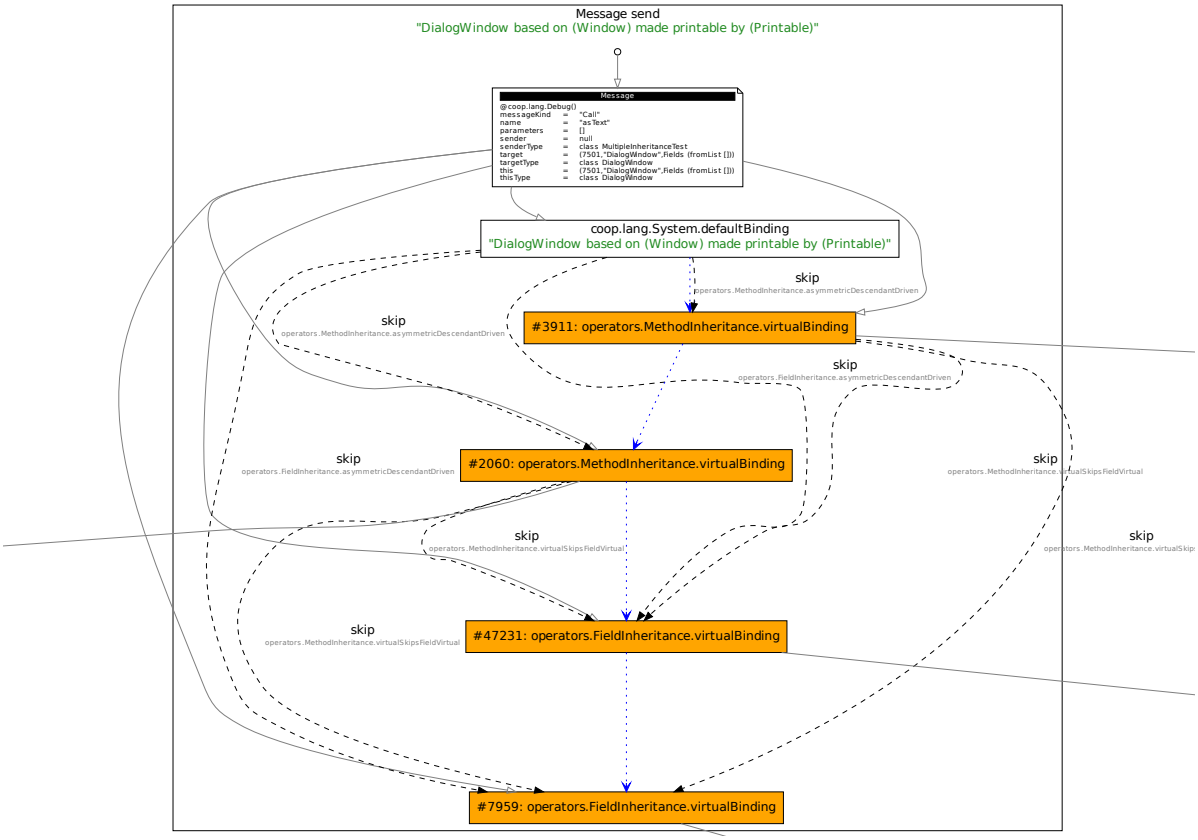
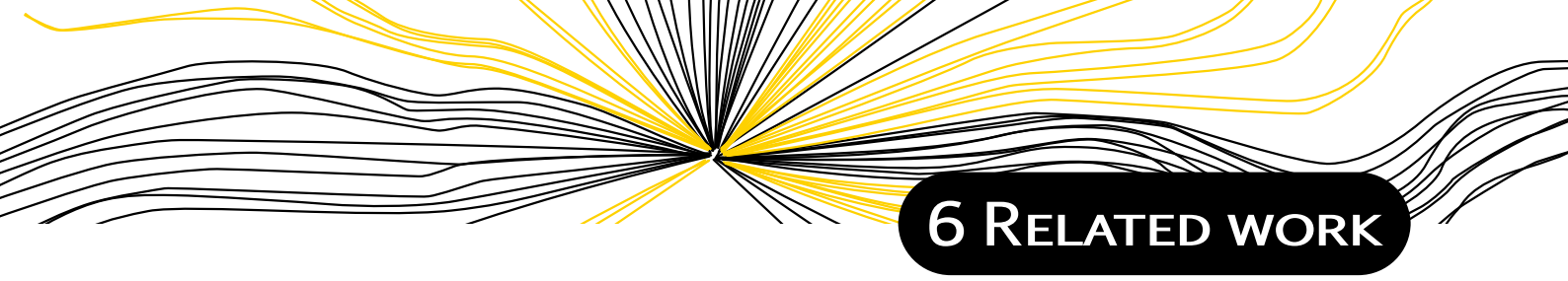


Figure 5.4.: Difficult function call using multiple inheritance, root of dispatch tree

applicability is calculated before constraints are evaluated. However, sometimes it is not necessary to know whether a certain binding is applicable or not, because the constraints prevent the execution of this binding anyway. Leaving the applicability of such a binding unevaluated might be a large performance gain, because the calculation of binding applicability can be lengthy due to its recursive definition.

An example of a function call where evaluating binding applicability is difficult, is a call using multiple inheritance. The dispatch of such a call is too difficult to show in detail here, but an overview is presented in figure 5.3. This diagram is extremely large because lookup is performed along every inheritance path until it reaches the topmost parent. However, when we zoom in on the root of the dispatch tree in figure 5.4, we see that the default binding succeeds. Therefore, the constraints cancel all other bindings which are present. Thus, it is unnecessary to evaluate the applicability of these other bindings. This means that evaluation of the root would be enough, which is considerably less than the evaluation of the full tree.



6 Related work

Havinga [16] published a first Co-op prototype, on which the design of our language is based. Therefore, our work is closely related to that of Havinga. Our main contributions to his work are:

More declarative selector language Selectors can be written in a more declarative way using specific selector expressions, which were not provided by Havinga. These specific expressions facilitate writing selectors easily and allows for more static reasoning, hence better robustness and more opportunities for optimizations. The selector expressions themselves are expressive enough for many selectors, but full expressiveness can only be achieved by invoking methods written in the turing-complete base code from selector expressions. Such an invocation is less declarative than a selector expression and is similar to the way Havinga handles all selectors.

Improved definition of constraints Havinga reused the constraints specified by Nagy [26]. We defined slightly modified versions of these constraints, increasing the expressiveness of the constraints. For example, we decouple ordering and control constraints and provide transitive versions of all constraints.

Unifying method invocation and field lookup The mechanisms for method invocation and field lookup are unified by modelling field lookup as message sends, which is how Havinga models method calls only. Allowing field lookup to reuse the functionality provided for method calls gives composition operators the ability to provide data abstraction.

Access to dynamic message properties Message rewrite can access the result returned by executed bindings, which is not possible in Havinga's work. Whenever a message send is dispatched to multiple methods, these methods can use the result returned by the previously called method.

Recursion avoidance in message selectors Because we allow using message sends in selectors, it is easy to cause infinite recursion accidentally. In the most common case, Co-op can detect and avoid this unwanted recursion. Havinga does avoid the use of message sends in selectors to avoid unwanted recursion.

Like the work of Havinga, our work is also related to a large body of research on defining new languages that support novel composition techniques, especially in the domain of object-oriented and aspect-oriented languages. Many papers also present a (small) set of composition techniques that aim at unifying existing composition techniques. However, most of such related research proposes a fixed set of composition operators, presented as part of a language, extension of a language, or an application framework. In contrast, our work focuses on a language that has no—or alternatively a single—built-in composition operators, but rather is a platform for constructing a wide range of user-defined composition operators.

To the best of our knowledge, there are no other languages that offer dedicated support for user-defined composition operators (that can be reused and combined), at least not within the domain of object-oriented and aspect-oriented languages. This excludes languages that offer generic extension mechanisms—such as macro's in Lisp—or allow for the extension and modification of the program through metaprogramming—as we will discuss below.

Of the research that aims at unifying composition techniques, we first discuss a few that relate particularly to the example composition operators we have shown in this thesis:

- In [7], mixin inheritance is presented as a generalization of both descendant- and parent-driven inheritance, as well as CLOS-style mixins. But the mixin mechanism itself is a fixed composition operator, and cannot be used to define new composition operators.

- We have demonstrated that we can define and use multiple composition operators, including the use of respectively Smalltalk/Java style *super* and Beta-style *inner*, in parallel. In [13], it has been shown how a specific method dispatching technique, implemented in the language MzScheme, allows the usage of both inheritance styles simultaneously.

- Compositional Modularity [2] is an inheritance model that supports a wide range of compositions on modules (which correspond to self-referential namespaces in this model). This is achieved by modelling the compositions as a set of operations on the modules. Compared to our approach, there are limitations to its expressiveness, due to the fixed set of primitive operations. The compositional modularity model has been applied to a variety of 'base' artefacts.

- Classpects [27] unify aspect- and object-oriented programming. The language Eos-U implements the classpect construct, which can be considered as a combination of aspect-behaviour and object behaviour in a single abstraction mechanism. Eos-U offers the concept of bindings, which have roughly the same structure as the bindings in Co-op: binding advice to join points. However, there is no mechanism for expressing ordering constraints beyond declaration order. Regardless of these similarities, Eos-U is distinct from Co-op by offering only a fixed set of composition operators and abstractions.
- Composition filters (or interface predicates) in the Sina language [1, 3] define a single language mechanism that can be used—among other things—to express various data abstraction mechanisms such as different forms of inheritance (single, multiple, conditional), delegation, and aspects. filter modules are abstractions of several filter expressions, but not an independent composition operator. The introduction of new filter types can be used to add additional composition behaviour to a system, but all within the same framework of composition filters, not as new, independent composition operators.

There are other approaches that allow for the construction of user-defined composition operators. In particular, our work relates to metaprogramming [9] and especially meta-object protocols [21]. Depending on the programming language/environment, metaprogramming offers the programmer the full power to modify the behaviour of programs. This includes the ability to write custom compositions. The power of metaprogramming comes with more complexity and responsibility [20]. In particular, it may be extremely hard to define multiple application-specific compositions in such a way that work together without interference (i.e. such that they are composable).

Meta-object protocols (MOPs) aim at addressing this by providing a framework—albeit at the metalevel—with more structure and constraints, so that e.g. composition operators can be defined within a well-defined structure. This means that the difficulty of language design—except for the concrete syntax—is now on the MOP designer. We are not aware of any MOPs (or languages, or frameworks) that offer similar generic abstractions and structure as we presented in this thesis. In particular, we do not know any MOPs that provide abstractions for defining new composition operators with similar variety, expressiveness and composability. For example, Co-op explicitly supports a variety of object-oriented as well as aspect-oriented composition mechanisms. In Co-op, composition mechanisms are constructed using first-class, composable elements, which can be reused to define or compose new composition mechanisms. In addition,

the resulting composition operators are also first-class entities, which means they can be reused and extended as well.

Of the research that aims at providing higher-level languages through frameworks or meta object protocols, we briefly discuss the following:

- AspectS [17] is framework for supporting aspect-programming in Smalltalk. It extends the Smalltalk MOP with features that enable aspect programming. As such, it does not extend the language itself. For instance, it uses Smalltalk itself as the pointcut language. Thus, it does not provide specific selector expressions like we do.
- MetaClassTalk [5] aims at 'unified aspect-oriented programming'. It exploits a combination of mixin-based inheritance and reflection to achieve this. Its aspects consist of (a) a set of mixins, (b) a pre-weaving script, (c) a post-weaving script. In this approach, every programmer is a meta-programmer, with a lot of control—and responsibility to write correct meta-programs. MetaClassTalk also involves 'weave-time' code; which is a disadvantage from the point of view of abstraction, but does have potential benefits with respect to performance optimization and static reasoning.

Finally, we mention several frameworks that aim at offering a generic platform for OO and AOP language implementations (e.g. [11], [15], and [29]). For such platforms, the designers have also made efforts to find a small set of generic constructs that typically serve as a target 'language' for a compiler/code transformation. An important distinction with our work is that these platforms do not aim at, and hence do not support, the ability of creating user-defined composition operators.

7 Possible improvements

7.1. Method signature

In Co-op, method signatures are defined by method name and number of arguments. The available implicit parameters are an implicit part of the method signature: a method can only be called when all implicit parameters are provided, but overloading based on implicit parameters is not possible. The message signature could be extended to include annotations explicitly. Having annotations as part of the method signature would give some additional possibilities. The next two sections explain some of the possibilities.

7.1.1. Annotations on the method

Currently, annotations can be specified on a method, but are not part of its signature. We could define (certain types of) annotations to be also part of the signature. Instead of encoding all information about a method in its name, this would allow the information to be specified in a more structured way using a set of annotations.

Of course, using annotations will complicate message selection, depending on the desired selection expressivity. Should it be possible to specify which annotations should be absent, besides the annotations that should be present? Should it be possible to specify annotation parameters and should these match exactly, or can logical operators be used for selection?

Further, the intention of annotations used at call side will become less clear. For example, think about the meaning of the annotation *@Debug* in the following call:

```
prof.@Debug makeAppointment("next␣Friday");
```

It could mean that we want to call an alternate implementation of *makeAppointment* which has *@Debug* as part of its method signature and shows debugging information when called. It could also mean that we use *@Debug* to add additional information to the call itself, which is handled by showing debug information during dispatch, instead of using it for selecting an implementation. Thus, because annotations can be

used in two different ways now, the intention of a call side annotation might not be clear immediately.

From the points mentioned above, we can conclude that some design decisions should be made before annotations can be integrated into method signatures. Because the notion of a method signature might become too complicated when annotations are integrated into the signature, we have decided not to do so. However, we think that the added expressivity might outweigh the added complexity. Therefore, further research would be interesting.

Side note
■ Defining a method by a set of annotations is a generalization of defining it by name. The name could easily be generalised to an annotation, for example *@Name("myMethod")*. At the source level, we could eliminate names by writing a message send using annotations only:

```
this.@Name("SomeMethod")();
```

We might even wonder why methods still have names at all. However, we think method names are very important to identify methods. A human can easily talk about the method called *SomeMethod*, but talking about a method having the annotation *@Name("SomeMethod")* would be less practical. Therefore, we think that methods should always have an explicit name at source level. However, at conceptual level we would prefer using annotations only, avoiding the additional concept of names. Defining a name as a synonym for a specific annotation might be a nice solution. □

7.1.2. Annotations on parameters

Allowing annotations on parameters to be part of the method signature could also be a nice addition. For example, specifying the type of parameters can be done by adding an annotation to it, like *@Type("String")*.

Another possibility is to use it for pattern matching, often used in functional languages. If multiple methods with the same name, but different annotations can be defined, one could write bindings allowing the following:

```
1 method factorial(@Equals(0) n) {
2     return 1;
3 }
4 method factorial(@GreaterThan(0) n) {
```

```
5     return n * factorial(n - 1);
6   }
```

Annotations on parameters are similar to annotations on the method itself: it would provide some nice abilities, but also creates additional complexities as mentioned in the previous section. Thus, introducing annotations on parameters is also open for further research.

7.2. Annotation processor

Processing annotations at compile time could improve the readability of the code. For example, in Co-op the annotation *@ImplicitParameters([])* is similar to the declaration of a static method. Therefore, it would be nice if the programmer could specify *@Static* as a synonym for this annotation.

Java offers the ability to process annotations at compile time, but does not allow modifications to the class being processed. Therefore, the functionality provided by Java is not expressive enough to achieve the behaviour described above. It would be nice if the Co-op compiler does allow the creation of new annotations based on annotations already present. This would not add any new features to the language, it will only improve the readability of the source code.

Another possibility of annotation processing could be generating some code for certain annotations. This might allow dispatch definition in a more declarative way. For example, we could assign to the annotation *@Extends(SomeClass)* to a class as follows:

```
1 class @Extends(SomeClass) ThisClass {
2   // Implementation of the class
3 }
```

If we could assign the code *Inheritance.extends(ThisClass, SomeClass)* to this annotation, inheritance could be specified using this single annotation. How code can be assigned to annotations and which expressiveness is desired is open for further research.

7.3. Binding scope

Currently, bindings and constraints are activated for the VM as a whole. Thus, bindings are a kind of static state. As a result, Co-op programs might not be re-entrant. Limiting the scope of bindings to the application only could avoid this problem. Therefore, it would be interesting to investigate the possibilities of creating different binding scopes, for example per application and per thread instead of per VM. Defining the scope could be done at definition or activation time. However, we do not know the possibilities and limitations of limiting binding scope. This is future work.

7.4. Java integration

It would be useful to have integration between Java and Co-op, allowing Co-op to reuse many libraries available to Java. In order to have proper integration, it might be desired to run Co-op inside the same virtual machine (VM) as Java. Using the Java Native Interface (JNI) [24], integration might also be possible, but it will be more difficult to do properly. In this case, proxy objects are necessary for every object passed from Co-op to Java and vice versa, in order to allow calls in both directions. Therefore, we think running Co-op in a—possibly modified—Java VM would be the most desired solution.

Co-op VM Compiling most Co-op code to JVM bytecode will be straightforward. The main challenge will be the dispatch logic. It should be possible to intercept all method calls and field accesses and handle them as message sends. Further, the evaluation of activated bindings and constraints should be done. This can be modelled in plain Java or using some kind of VM support. The former will be more portable, but the latter might allow better optimizations.

Frameworks which allow the interception of method calls and field accesses exist. AspectJ [22] is the most well known example, which is a Java language extension. We expect that AspectJ provides sufficient functionality to implement Co-op on top of it. However, it has a few limitations. For example, runtime weaving is not supported, so binding activation must be coded into the dispatch logic itself. ALIA [4] is another framework, providing support at a lower level than AspectJ; it does not provide a language by itself, but the features to compose a language from. In addition to

interception of method calls and field accesses, it provides the ability to deploy these interceptions at runtime. Thus, it features binding activation. It might also be possible to express our dispatch logic in terms of ALIA primitives, allowing the reuse of optimizations already part of ALIA.

Calling Java Calling methods on a Java object from Co-op only differs because in Java, argument types are part of the method signature. As long as the Java code does not provide methods overloaded on argument types, dispatch can work as in Co-op. In the presence of such overloads, a different default dispatch can be provided. This default dispatch could requires the programmer to annotate the message send—the method call—with the desired argument types. This way, it is possible to identify which Java method should be called. As soon as control is handled to a Java method, evaluation can proceed as normal Java, Co-op does not intervene in this process.

Passing arguments to Java Another difficulty is passing non-primitive arguments to Java. Method calls made on these objects should be handled as Co-op message sends. The difficulty might not be the interception of the call, but allowing Java to do these calls. Being strongly typed, Java requires objects passed as arguments to be subtypes of the declared argument types. Because Co-op is type dynamically, its objects do not conform to these types explicitly.

7.5. Constraint fields instead of methods

Currently, the definition of bindings and constraints causes the creation of a method which returns this binding or constraint. Instead of creating a method, it seems nicer to create a field containing the binding. Besides having a nicer way to retrieve bindings, a field would also allow reassigning its value. This might actually be a nice property to have, but can also cause unexpected results. For example, every binding instance has a reference to its surrounding object. However, when reassigning the binding to another object, this reference will remain unchanged, making it more difficult to identify the surrounding object of the binding.

Constraints specify a relation between multiple (types of) bindings. In order to identify these bindings, their fully qualified name is used. Normally, this is quite intuitive as the full qualified name equals the field name containing the binding.

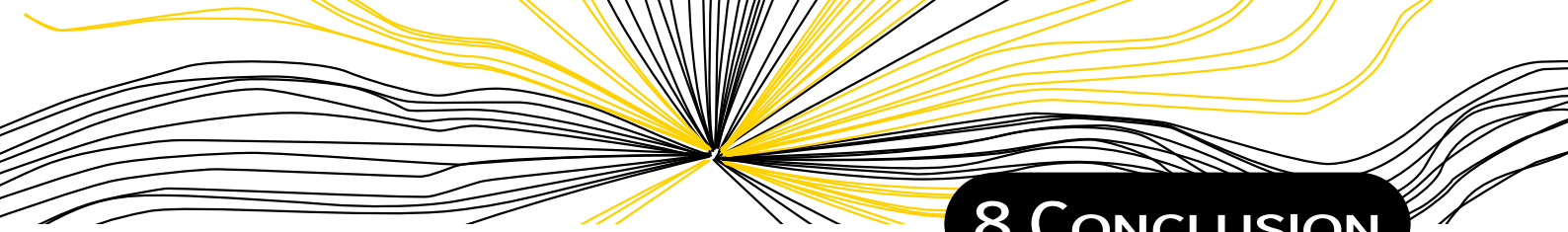
However, after reassigning a binding, the full qualified name might differ from the field name, making it difficult to see which constraint applies to which bindings.

7.6. Message send success

We have defined a message send to be successful if and only if at least one binding handles this message send successfully. This notion of message send success is an important part of the language. However, we think that the current definition of success might be too limited.

For example, introducing logging of all message sends might cause all these message sends to succeed, because the writing to the log succeeds. This behaviour is undesired, because logging should not influence the result of the application. Avoiding this influence can only be done by changing the definition of message send success. A possible solution would be providing an annotation that makes all annotated bindings invisible to the notion of success. This will solve the problem of the specific example, but we do not know whether it will be sufficient in general.

A more powerful approach would be allowing programmers to fully redefine message send success. This approach would be more complicated, because it requires a language in which message send success can be written. We do not know if such a solution is really desired or overly complicated. Therefore, we leave the notion of message send success open for further considerations.



8 Conclusion

Our goal was to provide a composition infrastructure for defining composition operators. We specified seven requirements of this composition infrastructure in chapter 1. We have defined the language Co-op that realizes this infrastructure. Below, we discuss to what extent Co-op meets the specified requirements.

Model composition operators as first-class entities Composition operators are defined using bindings and constraints. Both bindings and constraints are first-class entities, because they can be constructed at runtime, passed as parameters, returned from methods and assigned into a variables.

Provide a declarative way for defining composition operators Bindings, defined in section 3.6, specify message rewrites in a declarative way. Messages are selected using a message selector as defined in section 3.7. Such a message selector is declarative: evaluation order cannot be specified by the programmer. The runtime decides when and to what extent message selectors are evaluated. Relations among bindings are also expressed declaratively, by using constraints.

Support the definition of a variety of composition operators As shown in chapter 4, we have implemented several different composition operators. We have shown that we can implement at least 12 of the 16 properties of inheritance identified by Taivalsaari [28] in the Co-op prototype, and are confident that we can implement most of the other properties too. Thus, a variety of composition operators is supported.

Allow composition operators to be composed again We implemented different kinds of inheritance, as explained in chapter 4. Often, these different kinds of inheritance have certain properties in common. In such a case, the implementation of one kind of inheritance is composed from other kinds of inheritance. Thus, Co-op supports a variety of composition operators.

Allow multiple composition operators to be used within the same program Chapter 4 showed three composition operators: static method calls, event notification and multiple inheritance. It also explains that the first two of these composition operators are loaded by the Co-op runtime before any program is executed. Thus, the multiple inheritance example actually uses all three composition operators

together, showing that multiple composition operators are allowed in the same program.

Reuse the technology for defining programs in composition operator definition The message selectors presented in section 3.7 only have two special operators: the unordered *and* and *or*. Besides these operators, they reuse the operators that can be used for defining application programs, such as equality and message send. The message rewrite does also allow using any expression which can be used for defining application programs.

Allow composition operators to be distributed as libraries All composition operators presented in chapter 4 have been implemented as libraries. For every composition operator, we wrote a test that included this library, showing that all operators can be distributed as libraries.

From these requirements we can conclude that Co-op meets our goal. The contributions of Co-op can be summarized as follows:

More declarative selector language Selectors can be written in a more declarative way using specific selector expressions, which are not provided by Havinga [16].

Improved definition of constraints The constraints used by Co-op have increased expressiveness compared to the ones used by Havinga [16].

Unifying method invocation and field lookup The mechanism for field lookup is unified with the one for method invocation, allowing composition operators to provide data abstraction.

Access to dynamic message properties Message rewrite can access the result returned by executed bindings, which is not possible in Havinga's work [16].

Recursion avoidance in message selectors The most common case of infinite recursion, accidentally caused using message sends in selectors, can de detected and avoided by Co-op.

Implementation of most kinds of inheritance Of the 16 properties of inheritance, identified by Taivalsaari [28], we have implemented 12 properties, showing that all these kinds of inheritance can be used in a single language.

Visualisation of message dispatch We have created diagrams which visualise the dispatch performed by Co-op.

A Differences with Java

Some features of Java are not present in our language. These features are described in this section. Most features described here can be modelled in terms of dispatch.

Variable A variable definition is the same as in Java, except that we use the keyword *var* to define a variable instead of specifying its type.

Method A method definition begins with the keyword *method* instead of listing its return type. Because arguments are typeless, overloading based on argument types is not possible. Only overloading on number of arguments can be done and varargs are not available.

Inheritance Co-op is an object oriented language, which means all code is part of a class, like in Java. However, the language does not provide inheritance. This essentially means that the Java keywords *implements* and *extends* are not part of Co-op. Inheritance can still be used, because it can be defined in terms of dispatch.

Constructors Instead of constructors, a Co-op class object has a method *new()*, which instantiates a new instance of the class represented by the class object. This method replaces the Java keyword *new*.

Statics In Java, the keyword *static* has different semantics, depending on the context in which it is used. Co-op does not provide this keyword, but similar results can be achieved using the following concepts:

1. Static methods: a method without the implicit parameter *this* acts like a static method.
2. Static state: can be stored easily using singletons instead of static variables.
3. Static initializer: the method *initializeClass()* is called when a class is loaded.

4. Inner classes: all inner classes in Co-op are static inner classes. Non static inner classes can be defined in terms of dispatch by delegating to the surrounding instance.
5. Static imports: these imports are not possible in Co-op, because they do not provide any new functionality, only a shorter notation.

Access modifiers Which access modifiers make sense depends on which concepts are in use. For example, *protected* is only useful when inheritance is used. Therefore, the Java keywords *public*, *protected* and *private* are not part of the Co-op language. By default, all methods and fields are public. By using the right composition operator, access can be restricted. However, any code can disable this composition operator again, allowing access to private fields. This is similar to the method *AccessibleObject.setAccessible()* in Java, except that we have no security manager which can avoid changing the accessibility.

Pseudo variables The Java keywords *this* and *super* are used to access the current and the super class respectively. The *this* keyword acts like a variable, so in Co-op it is just an implicit parameter. However, the *super* keyword does not act like a variable, it controls the method and field lookup. That is, it changes the dispatch. Therefore, super can be modelled as a call annotation in Co-op.

Type checking Because dispatch can be changed at runtime, it is impossible to know at compile time which calls can be made and which fields can be accessed. Therefore, Co-op is dynamically typed. This is a major difference between Co-op and Java, but is not as far away from Java as it might seem at a first glance. The JVM will only check whether a method exists when it is executed, like Co-op does. The difference is that the Java compiler does not allow compilation when a called method does not exists. This can only be detected if the application is compiled as a whole, which is commonly the case, but is not required as every Java class has its own class file.

Imports Only single type imports are possible in Co-op. Imports on demand (like `import java.util.*`) are more difficult to resolve and more difficult to understand as some types imported on demand might be obscured by single type imports. Because

all modern IDEs provide features to define all necessary imports, the usefulness of imports on demand is not large. Therefore, these imports are not possible in Co-op.

Further, it is mandatory to import every used type. This is because we have reserved the dot solely for message sends, which means it is impossible to write full qualified names throughout the code. Thus, in general we use the simple name to refer to a class. However, simple names might not be unique. Therefore, importing classes under their simple name is only a default. Classes can also be imported under any other name in the following way:

```
import mylang.Object as MyObject;
```

Class In order to reference a certain class, it is enough to list its name. For example, where one would write *System.class* in Java, using *System* is enough in Co-op. In this example, *System* is an implicitly defined variable which will only be initialized when actually used. This means that classes are loaded lazily, like in Java.

B Composition operators

B.1. Static method calls

Listing B.1: Dispatch for static method calls

```
1  package coop.lang;
2
3  import coop.lang.Class;
4  import coop.lang.System;
5
6  class Static {
7    // Binding for static method lookup, which is applicable only if the default binding fails
8    // The check for a null target avoids infinite resends, Class Class will resend to itself only once
9    binding staticBinding = (targetType == Class & messageKind == "Call" & target != null)
         {
10     targetType = target;
11     target = null;
12     remove this;
13   }
14
15   constraint defaultBeforeStatic = skip(System.defaultBinding, staticBinding);
16
17   method initialize() {
18     this.defaultBeforeStatic().activate();
19     this.staticBinding().activate();
20   }
21
22   method @ImplicitParameters([]) initializeClass() {
23     Static.new().initialize();
24   }
25 }
```

B.2. Dispatch optional

Listing B.2: Dispatch optional

```
1  package coop.lang;
2
3  import coop.lang.System;
4
5  class DispatchOptional {
6    binding implementationOptional = (messageKind == "Call" & message @==
         @ImplementationOptional) {
7      targetType = DispatchOptional;
8      name = "noDispatch";
9      parameters = [];
10     message @-= @ImplementationOptional;
11   }
12
13   constraint whenDefaultFails = skip(System.defaultBinding, implementationOptional);
14
15   method @ImplicitParameters([]) noDispatch() {
16     // Nothing to be done when no dispatch is required
17   }
18
19   // Load this class before any other class.
20   // This class has a static initializer and after it has run, classes don't require one.
21   method @ImplicitParameters([]) initializeClass() {
22     var opt = DispatchOptional.new();
23     opt.whenDefaultFails().activate();
24     opt.implementationOptional().activate();
25   }
26 }
```

B.3. Method inheritance

Listing B.3: Method inheritance

```
1  package operators;
2
3  import coop.lang.System;
4  import operators.FieldInheritance;
5
6  class MethodInheritance {
7    class MultipleInheritance {
8      // Call this method for every parent type
9      method @ImplicitParameters([]) inherits(childType, parentType) {
10       MethodInheritance.new().createDispatch(childType, parentType);
11     }
12   }
13
14   var childType;
15   var parentType;
16
17   // Binding for virtual method lookup, which is applicable only if the default binding fails
18   binding virtualBinding = (messageKind == "Call" & targetType == this.childType) {
19     targetType = this.parentType;
20   }
21
22   // Binding for explicit super calls, causing the default binding to become inapplicable.
23   // Note that the targetType is not checked against the senderType, so besides this.@Super,
         something.@Super will also work.
24   binding superBinding = (messageKind == "Call" & message @== @Super & targetType
         == this.childType) {
25     targetType = this.parentType;
26     message @-= @Super;
27   }
28
29   // Call the super constructor to create a parent
30   binding superConstructorBinding = (messageKind == "Call" & name == "new" & target
         == this.childType) {
31     targetType = MethodInheritance;
```

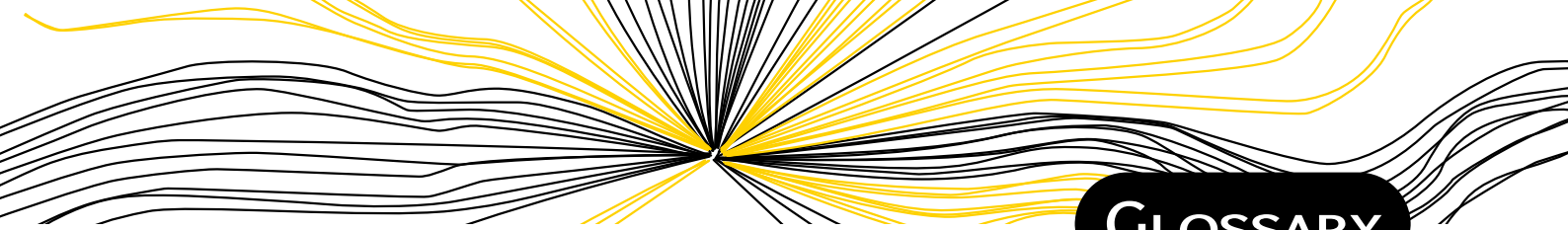
```
32      target = null;
33      name = "createParent";
34      parameters = [result, this.parentType]; // The newly created object is "result"
35    }
36
37    constraint superSkipsDefault = skip(superBinding, System.defaultBinding);
38
39    constraint superSkipsFieldSuper = skip(superBinding, FieldInheritance.superBinding);
40    constraint virtualSkipsFieldVirtual = skip(virtualBinding,
          FieldInheritance.virtualBinding);
41
42    constraint superConstructorConstraint = cond(System.defaultBinding,
          superConstructorBinding);
43    constraint superConstructorOrderConstraint = pre(System.defaultBinding,
          superConstructorBinding);
44
45    method @ImplicitParameters([]) createParent(child, parentType) {
46      FieldInheritance.new().createDispatch(child, parentType.new());
47      return child;
48    }
49
50    method createDispatch(childType, parentType) {
51      this.childType = childType;
52      this.parentType = parentType;
53      // Activate all constraints
54      this.defaultSkipsVirtual().activate();
55      this.superSkipsDefault().activate();
56      this.superSkipsFieldSuper().activate();
57      this.virtualSkipsFieldVirtual().activate();
58      this.superConstructorConstraint().activate();
59      this.superConstructorOrderConstraint().activate();
60      // Activate all bindings
61      this.virtualBinding().activate();
62      this.superBinding().activate();
63      this.superConstructorBinding().activate();
64    }
65  }
```

B.4. Field Inheritance

Listing B.4: Field inheritance

```
1  package operators;
2
3  import coop.lang.System;
4
5  class FieldInheritance {
6    var child;
7    var parent;
8
9    // Binding for virtual field lookup, which is applicable only if the default binding fails
10   binding virtualBinding = (target == this.child) {
11     target = this.parent;
12     targetType = System.classOf(this.parent);
13   }
14
15   // Binding for explicit super field access, causing the default binding to become inapplicable.
16   // Note that the target is not checked against the sender, so besides this.@Super,
         something.@Super will also work.
17   binding superBinding = (message @== @Super & target == this.child) {
18     target = this.parent;
19     targetType = System.classOf(this.parent);
20     message @−= @Super;
21   }
22
23   constraint defaultSkipsVirtual = skip(System.defaultBinding, virtualBinding);
24   constraint superSkipsDefault = skip(superBinding, System.defaultBinding);
25   constraint cloneChildFirst = skip(cloneChildBinding, cloneBinding);
26
27   method createDispatch(child, parent) {
28     this.child = child;
29     this.parent = parent;
30     // Activate all constraints
31     this.defaultSkipsVirtual().activate();
32     this.superSkipsDefault().activate();
33     // Activate all bindings
```

```
34      this.virtualBinding().activate();
35      this.superBinding().activate();
36      this.changeParent().activate();
37    }
38  }
```



Glossary

binding Selects and rewrites certain messages. 15, 16, 23, 24

composition operator Language mechanism allowing programmers to compose behaviour and/or data. 1, 2, 41

condition Boolean condition used to select certain messages. 15, 23–27, 31

constraint Defines constraints between bindings. 15, 16, 32, 33, 35–39

dispatch The processing of message sends. 15, 20, 22, 23

implicit parameter Parameter which is implicitly passed from a message to the called method. 17, 18, 21, 22

message rewrite Part of a binding which rewrites the incoming message into the one to be sent. 16, 23, 31

message selector Condition which is used in a binding for message selection. 23, 24, 27, 28

Bibliography

[1] M. Akşit and A. Tripathi. Data abstraction mechanisms in SINA/ST. In *3rd Conf. Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 267–275. ACM, 1988.

[2] G. Banavar and G. Lindstrom. An application framework for module composition tools. In *In ECOOP '96, number 1098 in Lecture Notes in Computer Science*, pages 91–113. Springer Verlag, 1996.

[3] L. Bergmans and M. Akşit. Composing crosscutting concerns using composition filters. *Comm. ACM*, 44(10):51–57, Oct. 2001.

[4] C. Bockisch. *An Efficient and Flexible Implementation of Aspect-Oriented Languages*. PhD thesis, Technische Universität Darmstadt, 2009.

[5] N. Bouraqadi, A. Seriai, and G. Leblanc. Towards unified aspect-oriented programming. In *Proceedings of ESUG 2005 (13th international smalltalk conference)*, 2005.

[6] G. Bracha. *The programming language jigsaw: mixins, modularity and multiple inheritance*. PhD thesis, Citeseer, 1992.

[7] G. Bracha and W. Cook. Mixin-based inheritance. In *Conf. Object-Oriented Programming: Systems, Languages, and Applications; European Conf. Object-Oriented Programming*, pages 303–311. ACM, 1990.

[8] A. Church and J. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, May 1936.

[9] P. Cointe. Reflective languages and metalevel architectures. *ACM Comput. Surv.*, page 151, 1996.

[10] E. Dijkstra. On the role of scientific thought. *Selected Writings on Computing: A Personal Perspective*, pages 60–66, 1982.

[11] R. Dyer and H. Rajan. Nu: a dynamic aspect-oriented intermediate language model and virtual machine for flexible runtime adaptation. In *AOSD '08: Proceedings of the 7th International Conference on Aspect-oriented Software Development*, New York, NY, USA, 2008. ACM.

[12] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1983.

[13] D. S. Goldberg, R. B. Findler, and M. Flatt. Super and inner: together at last! In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 116–129, New York, NY, USA, 2004. ACM.

[14] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java™ Language Specification, Third Edition*. Addison-Wesley, May 2005.

[15] M. Haupt and H. Schippers. A machine model for aspect-oriented programming. In *ECOOP 2007: Proceedings of the 21th European Conference on Object-Oriented Programming*, pages 501–524, 2007.

[16] W. K. Havinga. *On the Design of Software Composition Mechanisms and the Analysis of Composition Conflicts*. PhD thesis, University of Twente, Enschede, June 2009.

[17] R. Hirschfeld. Aspect-oriented programming with AspectS. In M. Akşit and M. Mezini, editors, *Net.Object Days 2002*, Oct. 2002.

[18] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. Report on the programming language haskell: a non-strict, purely functional language version 1.2. *SIGPLAN Not.*, 27:1–164, May 1992.

[19] S. Jones, C. Hall, K. Hammond, J. Cordy, H. Kevin, W. Partain, and P. Wadler. The Glasgow Haskell compiler: a technical overview. 1992.

[20] G. Kiczales. It's not metaprogramming. *Software Development Magazine*, (10), 2004.

[21] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Massachusetts, 1991.

[22] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. ECOOP 2001, LNCS 2072*, pages 327–353, Berlin, June 2001. Springer-Verlag.

[23] S. Kleene. *Introduction to metamathematics*. Wolters-Noordhoff, 1971.

[24] S. Liang. *The Java native interface: programmer's guide and specification*. Addison-

Wesley Professional, 1999.

[25] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. *SIGPLAN Not.*, 21(11):214–223, 1986.

[26] I. Nagy. *On the Design of Aspect-Oriented Composition Models for Software Evolution*. PhD thesis, University of Twente, 2006.

[27] H. Rajan and K. J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 59–68, New York, 2005. ACM Press.

[28] A. Taivalsaari. On the notion of inheritance. *ACM Comput. Surv.*, 28(3):438–479, 1996.

[29] É. Tanter and J. Noyé. A versatile kernel for multi-language AOP. In R. Glück and M. R. Lowry, editors, *Generative Programming and Component Engineering, 4th International Conference (GPCE)*, volume 3676 of *Lecture Notes in Computer Science*, pages 173–188. Springer, Sept. 2005.

[30] D. Thomas. Message oriented programming. *Journal of Object Technology*, 3(5):7–12, 2004.

[31] D. Watt and W. Findlay. *Programming language design concepts*. John Wiley & Sons, 2004.

[32] P. Wegner. Concepts and paradigms of object-oriented programming. *SIGPLAN OOPS Mess.*, 1:7–87, August 1990.

[33] W. Wilkes. Instance inheritance mechanisms for object oriented databases. *Advances in Object-Oriented Database Systems*, pages 274–279, 1988.