Vrije Universiteit Brussel - Belgium
Faculty of Sciences
In Collaboration with École des Mines de Nantes - France
2004

A Forward-Chaining Inference Rule Engine for a Prototype-Based Language

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange)

By: Óscar Andrés López Paruma

Promoter: Prof. Theo D'Hondt (Vrije Universiteit Brussel)
Co-promoters: Ellen Van Paesschen and Wolfgang DeMeuter
(Vrije Universiteit Brussel)

Abstract

One of the fundamental research areas in computer science is concerned with representing and inferring knowledge from information. Through the years, many techniques have been proposed to solve the related difficulties, either separately, or by combining knowledge representations with knowledge inference. In this dissertation, prototype-based languages, as an expressive knowledge representation, are combined with forward-chaining rule-based reasoning, as a mechanism to infer knowledge. More specifically, we propose that a forward-chaining inference engine on top of a prototype-based language, supporting a rule language, composed of expressions in the base language, can result in a powerful mechanism for meta-programming in an interactive manner. To support our claim, a forward-chaining hybrid system was integrated into the prototype-based programming language SELF. Experiments in the context of monitoring code, in search of "bad smells" and suggesting refactorings, show its utility as a meta-programming tool.

Acknowledgement

First of all, I would like to thank my promoter, **Prof. Dr. Theo D'Hondt** for giving me the chance of studying the *European Master in Object Orientation and Software Engineering* program.

Next I want to give a huge thank you to my advisor, **Ellen Van Paesschen**, for supporting me and helping me during the hardest moments of this thesis, thanks also to **Maja D'Hondt**, **Kris Gybels** and **Wolfgang De Meuter** for all the help you gave me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays, the field of study of computer science is strongly linked with the study of knowledge. How to infer knowledge from information and how to represent knowledge has been a research subject for years. There have been countless proposals to solve both questions.

On the one hand, modern object-oriented prototype-based programming languages (specifically, SELF [Sel]) have many advantages in the context of knowledge representation, especially in comparison with most class-based languages.

On the other hand, production rules can serve as a mechanism for inferring knowledge from information. This dissertation combines both approaches: a hybrid system that results from the blend between rule-based reasoning and an object-oriented prototype-based programming language.

The driving idea behind this dissertation is situated in the context of (declarative) meta-programming. Today there exist powerful, mature backward and forward-chaining tools such as SOUL [SOU] for performing rule-based reasoning at the meta-level. However, there is room for improvement, for instance in the context of interactive programming since these systems involve program execution outside the main programming environment. Moreover, the associated rule language often consists of expressions in a language different from the target language.

We propose that a forward-chaining system, capable of expressing rules in terms of expressions in a prototype-based programming language can be a means for meta-programming, possibly in an interactive programming style, as it can monitor a constantly changing fact base.

Therefore, a forward-chaining inference rule-based engine was built on top of SELF, with the ability to use SELF expressions in both the condition and action parts of the rules. New means were conceived for interpreting the (implicit) universal quantification in a rule, thus providing new ways for grouping together objects in SELF. Moreover, combining a forward-chaining production rules system with SELF adds a meta-programming layer, that

goes beyond SELF's own meta-object protocol.

As a case study in meta-programming, this dissertation will show how a forward-chaining rule-based engine can be used as a tool for guided programming in SELF, by providing rules that can suggest possible refactorings to the user while programming.

The rest of this dissertation is structured as follows: chapter 2 discusses the philosophy behind current knowledge representation paradigms, highlights the advantages of using the prototype-based paradigm, provides a classification of the related programming languages, and finally introduces the programming language SELF.

Next, chapter 3 introduces the formalisms used for expressing rules, the most common ways for chaining them and details the Rete algorithm. This chapter also introduces hybrid systems in the context of production systems, and presents NéOpus [NeO], which was used as a guide for building the forward-chaining system developed during this research.

Chapter 4 details the structure and implementation of this forward-chaining inference rule-based engine built on top of SELF. The topics discussed include the problems encountered during the transition from the class-based to the prototype-based paradigm and the challenges in finding ways to group objects in a prototype-based world. Also the (dis)advantages of prototypes as opposed to classes, in this specific context are discussed. The high-level structure and evaluation process of the rule-based system, together with the rule language are included.

In Chapter 5 we present the developed system and the associated rule language as a new meta-programming layer on top of SELF, which enables interactive programming, as illustrated with some experiments on refactoring code.

Finally, Chapter 6 enumerates the conclusions of this research, while chapter 7 proposes possible future research directions.

Part I

Preliminaries

Chapter 2

Prototypes and SELF

This chapter introduces some of the philosophical foundations that lie behind the way in which knowledge is represented inside object-oriented programming systems; in essence this means comparing the *classification* theory with the *prototype* theory of knowledge representation. Both approaches map into two different kinds of programming languages: class-based and prototype-based; more emphasis will be put on describing the latter, as the objective of this thesis is showing that a prototype-based language (PBL, for short) could be better suited for performing rule-based inference.

Subsequent sections will contain a brief history and some classification attempts of prototype-based languages, concluding with a detailed view of SELF [Sel], the language used for this research.

2.1 Knowledge Representation: Philosophical Foundations

There has been a long standing philosophical controversy questioning the best way to represent concepts: as abstract sets or classes, or as concrete prototypes. It will be shown that the roots of this discussion can be traced back to the works of ancient Greece's philosophers.

2.1.1 Early Developments on Classification Theory

The earliest mention of *classes* versus *instances* dates back to over two thousand years ago, and it was stated by Plato [Tai99]. He made the distinction between *forms* —"ideal" descriptions of things and the particular *instances* of this forms. For Plato, the world of ideas was more important than the world of instances. Aristotle, Plato's student, continued the research into classification and intended to provide a detailed and comprehensive taxonomy of all natural things, grouping them under the same category if they shared the same properties, and allowing the definition of new categories in

terms of other categories if the new ones have at least the same properties as the defining ("genus") ones. This rule can be summarized as:

$$\text{essence} = \text{genus} + \text{differentia}$$

In this way, a new category is defined in terms of its *defining* properties and *distinguishing* properties. However, Aristotle also realized that his model had problems dealing with those properties characteristic of an object that are atypical for that kind of objects. He called them "accidental" properties, and restated the substance of a concept in terms of two aspects: the *essence* and the *accidents*.

From a mathematical point of view, the classification theory is directly related with the concept of sets (or classes). A set is constructed either by describing the principles that identify membership in a set, or by enumerating all of its members. In the first case, the description of a set enumerates all the essential properties of its members, and an object is said to be a member or an *instance* of a set if it belongs to that set.

2.1.2 A Challenge on Classification

Nobody disputed Aristotle's ideas on classification during a long time. The first people that questioned the "classical" view were the British philosophers W. Whewell and W.S. Jevons [Tai99] in the 19th century. They realized that there is a great subjective component involved in the process of classifying things, and that process requires creative invention and evaluation. As a consequence, there are no universal rules to determine the properties that should be used as the basis for classifying objects, and there are no objectively "right" classifications.

In 1953, Ludwig Wittgenstein noticed that it is difficult to say in advance what are the essential characteristics of a concept, and he also provided several examples of "simple" concepts that are extremely difficult to define in terms of shared properties, like "game" or "work of art". As an alternative, Wittgenstein stated that the meaning of most concepts is determined not by definition, but by family resemblances. Such concepts can be defined only in terms of similarity and representative "prototypes".

2.1.3 Prototype Theory is Born

Wittgenstein's ideas inspired new research in the emerging prototype theory. J.L. Austin (1961), L. Zadeh (1965), F. Lounsbury and many others started to investigate the topic. But it was Eleanor Rosch who introduced prototype theory in the mid-1970s. She observed that categories, in general, have best examples ("prototypes"), and that all of human senses play a role in categorization. Rosch focused her research on two implications of the classical theory:

- If categories are defined only by properties that all members share, then no members should be better examples of the category than any other members.
- If categories are defined only by properties inherent in the members, then categories should be independent of the peculiarities of any beings doing the categorization.

The above implications are not typically true when people do classifications. In fact, some instances are "better" representatives of categories than others, and our background, mental capabilities and experience play a significant role in the classification process.

Observations such as those above showed the flaws in the classical classification model, and formed the basis for the prototype theory stated by Rosch and others. The essential results can be summarized as follows [Lak87]:

- Some categories are graded; that is, they have inherent degrees of membership, fuzzy boundaries, and central members whose degree of membership (on a scale from zero to one) is one.
- Other categories have clear boundaries; but within those boundaries there are graded prototype effects — some category members are better examples of the category than others.
- Categories are not organized just in terms of simple taxonomic hierarchies. Instead, categories "in the middle" of a hierarchy are the most basic. Most knowledge is organized at this level.
- The basic level depends upon perceived part-whole structure and corresponding knowledge about how the parts function relative to the whole.
- Categories are organized into systems with contrasting elements.
- Human categories are not objectively "in the world", external to human beings. Many categories are embodied, and defined jointly by the external physical world, human biology, the human mind, plus cultural considerations.

2.2 From Philosophy to Programming Languages

Section 2.1 summarized the two main philosophical currents concerned with the question of how to represent human knowledge. They obviously influence the way in which programming languages are built, exploiting either the class-based paradigm or the prototype-based paradigm for representing knowledge. In this section, the practical implications of the previous philosophical discussion shall be enunciated. As an aside note, it is surprising

to find that most software developers have no idea of the conceptual and philosophical background surrounding object-oriented programming, even though such concepts directly impact the way in which a program is built.

2.2.1 Classes vs. Prototypes

Most objected oriented languages are built around the idea that it is possible to abstract out the general, "essential" concepts that apply to all members of a class, and describe all of them at once. However, as Wittgenstein stated, it is difficult to say in advance what characteristics are essential for a concept. In the mathematical world the concept of set has proven useful for describing properties that apply to many objects, but in the every-day world the prototype approach in some ways corresponds more closely to the way people seems to acquire knowledge [Lie86]. In class-based systems, people are expected to find out from the beginning the abstract properties that apply to all the members of a class of objects; on the other hand, prototype-based systems encourage people to create concrete concepts first and then generalize them by saying what aspects of the concept are allowed to vary.

Class-Based Languages and their Limitations

Conceptually, one of the sources of complexity of having classes in an object-oriented system lies in the fact that they play multiple roles, with several different functions. For example, in Smalltalk [GR85] some of the roles played by a class include [Bor86]:

- generators of new objects,
- descriptions of the representation of their instances,
- descriptions of the message protocol of their instances,
- elements in the description of the object taxonomy,
- a means for implementing differential programming,
- repositories for methods for receiving messages,
- devices for dynamically updating many objects when a change is made to a method, and
- sets of all instance of those classes

From a more pragmatic point of view, several limitations arise when using an object-oriented programming language for representing the knowledge of a domain; these limitations are not exclusive of but are more evident in class-based systems [Tai99]:

Limited modeling capabilities The programming model used by most object-oriented programming languages closely resembles the Aristotelian classical model of the world: new classes are defined in terms of shared properties, and all the instances of a class have the same set of properties; even the inheritance model commonly used in object-oriented languages is very similar to the Aristotelian way of defining new categories in terms of existing genealogical parents. As was discussed in sections 2.1.2 and 2.1.3, Aristotle's model has its own shortcomings and given that class-based programming languages follow so closely this model, it is easy to see that the same shortcomings also apply to such languages. In particular, there are many concepts and domains that cannot naturally be modeled in terms of shared properties (things like a traffic jam, water, the greenhouse effect).

No optimum class hierarchies Aristotle believed in the existence of a "single correct taxonomy of all natural things". Rosch's research has shown that the categorization process is a subjective activity, and a "single and correct taxonomy" cannot be achieved. As a consequence, it is possible to assert that an *optimum* class hierarchy does not really exist for a given domain. In practice, a good class hierarchy involves trade-offs in various regards, for example a reusable library is not necessarily an efficient one, or a very efficient library may lack extensibility.

Basic classes and the need for iteration A very important contribution of Rosch's prototype theory is the observation that not all concepts and categories are equal. There are categories that are more "basic" than others and objects that are "better" representatives of their respective categories. When categories are organized into taxonomic hierarchies the basic classes end up in the middle of the class hierarchy, as the classes at the top tend to be overly generic and the ones at the bottom overly specific. However, the basic categories are usually found first whereas the general ones can only be deduced later when more experience from the problem domain has been gathered. A conceptual conflict arises here: On one hand, the implementation of a class-based hierarchy starts from top to bottom, as superclasses must exist before their subclasses. On the other hand, the generic, more abstract classes are usually found after some experience working with a knowledge domain reveals generalizations and new abstractions that should be placed at the top of the hierarchy. As a consequence of this conflict, the construction of object-oriented class libraries becomes an iterative process where the useful abstractions are discovered rather than invented after a number of iterations.

Prototype-Based Languages

The classification theory in philosophy laid the foundations for class-based languages (CBLs for short), in a similar way the prototype theory inspired a new paradigm of object-oriented languages: the prototype-based programming model. In this model, all programming is done in terms of concrete, directly manipulable objects often referred to as prototypes [Tai99].

There are many different languages that adhere to the prototype-based paradigm, each of them with its unique set of characteristics (see 2.3.3 and 2.3.4). However, some features are common to most of them, such as an increased object flexibility (for instance, it is often possible to add or remove variables or methods in individual objects), creation of new objects by coping or cloning existing ones, and inheritance is replaced by other mechanisms such as *delegation* in SELF [Sel] or *concatenation* in Kevo.

The existing PBLs have been developed for different application domains, with different goals in mind. However, they share the same goals and same advantages. For instance, their goals include [CDB99]:

- Provide simpler descriptions of objects. People naturally grasp new concepts by creating concrete examples rather than abstract descriptions.
- Offer a simpler programming model, with fewer concepts and primitives.
- Offer new capabilities to represent knowledge, because CBLs constraint objects too tightly.

Some advantages of using PBLs include [Tai99], [Bor86]:

- A reduced need for *a priori* classification.
- The "infinite regression" problem with classes is completely absent.
- Encouragement of a more iterative and exploratory programming and design style.
- In general, the designer does not deal with abstract descriptions or concepts, instead is faced with concrete realizations of those concepts.
- Design is driven by evaluation in the context of examples.
- Learning a PBL can be simpler than learning a CBL, there are fewer concepts to cope with.
- Any object can be given individualized behavior.
- A PBL would provide better support for concrete, visual programming systems.

- Prototypes have advantages for incremental learning of concepts.
- Provide better support for experimental programming

Not everything is perfect, of course. Most prototype-based languages are more motivated by technical matters than the philosophical basis of knowledge representation, and they do not usually take into account the conceptual modeling side [Tai99]; there is an organizational gap evident in some PBLs which lack a mechanism for grouping together objects that share similar characteristics, as sometimes a more rigid, structured way of organizing an object system is necessary (this point is particularly interesting for the current research, and will be treated in more detail in the following chapters). Also, PBLs have not yet found a place in the industry; besides the huge success of JavaScript and a rather modest usage of NewtonScript, no other PBL has been embraced by the industry.

2.3 Prototype-Based Languages: Origins and Classification Attempts

For any given field of knowledge, it is important to be able to recognize its origins and be aware of its diversity. To accomplish this objective, the present section provides an overview of the genesis of prototype-based programming, alongside three different classification attempts that show several examples of such languages and how do they compare with each other in terms of their features, primitive mechanisms and program organization.

2.3.1 Origins of Prototype-Based Programming

Prototypes and Frames Systems

The predecessors of prototype-based languages were invented in the late seventies by the AI community for knowledge representation purposes. The idea of using the prototype theory developed in cognitive science (see 2.1.3) and differential description to represent knowledge was pioneered by the frame theory (Minsky 1975) and the frame-based languages KRL (Bobrow and Winograd 1977) and FRL (Roberts and Goldstein 1977) [CDB99]. Frames were designed to represent knowledge such as typical values, default values, or exceptions, which are difficult to describe in other formalisms. It is clear that frame-based languages have influenced prototype-based languages.

Structure A frame is a set of attributes, each one representing one characteristic of the frame as an "attribute name – set of faces" pair.

Differential description and parents It is possible to create a new frame by only expressing the differences from an existing one used as a prototype. This produces an *"is-a"* kind of relationship.

```
Frame
  name: ‘‘whale’’          Frame
  category: mammal           name: ‘‘Moby-Dick’’
  environment: sea           is-a: whale
  enemy: man                 color: white
  weight: 10000              enemy: Captain-Ahab
  color: blue
```

Figure 2.1: Frame and differential description

Frame hierarchies and inheritance The "is-a" relationship is an order relationship that defines frames hierarchies. A frame can inherit a set of attributes from its parent.

Figure 2.1 shows an example of a frame and a differential description.

Actor Languages

The actor language Act1 (Lieberman 1981) also represents entities with classless objects. Although Act1 provides objects and mechanisms conceptually similar to those of frame systems, it should be pointed that the difference between frame-based languages and Act1 is that Act1 is a programming language, not a knowledge representation language [CDB99].

Actors have attributes ("properties") and behavior ("methods"). Methods are invoked by sending messages to actors. Actors are created by cloning and extending existing ones with the primitives `create`, `extend` and `c-extend`.

Cloning Act1 introduced cloning (shallow copying) as a standard way to create objects. The basic idea is that, given an existing concrete object, it is easier to get a new similar object by copying it. The `create` primitive combines cloning with the addition of new properties.

Extension The `extend` primitive allows the creation of objects by using the delegation mechanism to achieve differential description. An object created this way it is called an *extension*, and its parent the *proxy*.

Delegation and inheritance The relationship between an actor and its proxy is similar to the "is-a" relationship between frames. It is implemented by a link named `proxy`, which is used by the delegation mechanism. An actor can inherit properties from its proxy, and whenever it does not know how to handle a message, the proxy is asked to answer for it by way of the delegation mechanism.

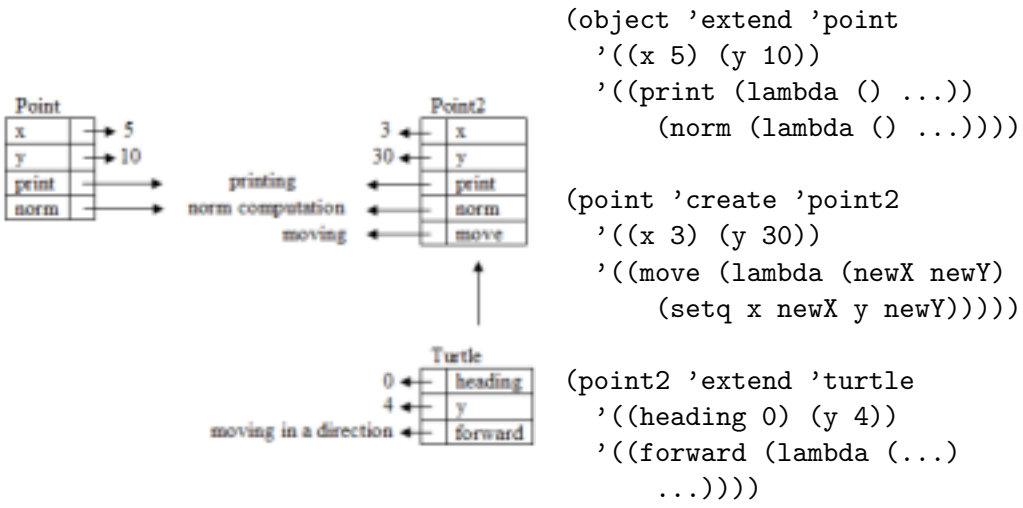

Figure 2.2: Cloning and extension in Act1

Figure 2.2 shows an example of cloning and extension in Act1.

As a final remark for this section, there is one very important reading, essential for understanding the origins of prototype-based languages: "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems" by Henry Lieberman [Lie86]. It is the first reference to a prototype-based *programming* language, and documents the decisions behind the implementation of a PBL, such as using delegation instead of inheritance, makes a comparison between delegation and inheritance (concluding that the former is more powerful than the latter), talks about efficiency concerns and provides several examples of prototype-based applications. It is indeed a classical document in its subject, and it should be studied together with this chapter.

2.3.2 The Treaty of Orlando

The treaty of Orlando [LHU87] [DUH88] is a small document written by H. Lieberman, L. Stein and D. Ungar during OOPSLA '87 in Orlando (Florida), which constitutes one of the earliest attempts to find the general principles that apply to all object-oriented systems, class-based, prototype-based or otherwise. Its importance lies in the fact that it also illustrates a rudimentary classification attempt of the PBLs available at the time, based on their features. Sections 2.3.3 and 2.3.4 provide a more up-to-date classification.

The treaty of Orlando embodies a consensus between different approaches for building an object oriented system. The most important contribution of this treaty to the field of computer science is the identification of two funda-

mental concepts which in one way or another are present in any object oriented language; in fact, they are the foundations of modern object-oriented systems: templates and empathy. They are called "fundamental" because they cannot be defined in terms of one another, and most object-oriented languages can be described largely in terms of the ways in which they combine these mechanisms. Also, it becomes clear that there is not a "best way" for designing an object oriented programming language around the concepts of templates and empathy, because there are independent degrees of freedom or dimensions for each of them, which make sense in different contexts.

The templates mechanism provides the way for creating new objects, offering guarantees about the similarity of group members; the instantiation of classes and the copying (cloning) of prototypes are particular cases of templates. The empathy mechanism allows an object to act as if it were another object, thus allowing the sharing of state and behavior. Inheritance and delegation are two different implementations of empathy.

The different dimensions of templates include: whether prototypes, classes, both or neither mechanism is used for constructing new objects; whether an object, once it is created, can or can not gain or lose attributes -in other words, whether the template is or is not strict-, or if the concept of strictness simply does not apply.

The degrees of freedom for the empathy include: whether the empathy it is static (the sharing patterns are fixed at compile time or at object creation time) or dynamic (the sharing patterns are established during runtime); whether it is implicit (the system directs automatically the patterns of sharing between objects), explicit (the patterns of sharing are directed by the programmer) or both; and whether the behavior of an object can be specified for just one object (idiosyncratic behavior), for a group of objects, or for both.

There are two kinds of sharing in object-oriented systems: anticipated and unanticipated. If the designer of a system can anticipate the general structure and relationship between the parts of a system, and if this structure is relatively static, then the best approach would be to write down the anticipated structure, encoding the anticipated sharing of behavior in a mechanism suitable for the task, e.g., classes. However, if the system that needs to be modeled has a high probability of following unpredictable paths, a programming language that is easily adapted to changes in structure and behavior would be a better choice.

Table 2.1 [LSU88] summarizes the findings of the authors of the treaty, applied to some object-oriented systems of the time of the document, both class-based and prototype-based

Language	Determination of	Empathy		Template	Mechanisms
	When	**How**	**For**	**What**	**How**
Actors	runtime	explicit	per object	none	none
Delegation	runtime	both	per object	none	none
SELF	runtime	implicit	per object	templates	non-strict
Simula	compile time	implicit	per group	classes	strict
Smalltalk	object creation	implicit	per group	classes	strict
Hybrid	runtime	both	both	any	non-strict

Table 2.1: Languages and their attributes according to the treaty of Orlando

2.3.3 Classification According to Primitive Mechanisms

Dony, Malenfant and Bardou [CDB99] present a comprehensive classification of prototype-based languages under two different criteria: according to the primitive mechanisms that constitute their virtual machine, and according to the way in which programs built with the languages are organized. This and the next section summarize the authors' findings, which serve as a good starting point to understand the commonalities and differences found in modern PBLs.

The classification according to primitive mechanisms relies on the answers to several questions about the low-level implementation details of the PBLs under study.

Regarding object representation, are objects represented with attributes and methods or with slots?. Slots allow an uniform way of accessing attributes and methods, enforcing an implicit encapsulation mechanism.

Regarding object creation and evolution, is it possible to create new objects *ex-nihilo*? Can an object's structure be modified dynamically? Is a cloning primitive available in the language?.

With respect to the way inheritance is implemented and life-time sharing between objects, does the language allow creation by extension? Is it possible to have multiple parents? Is it possible for an object to change its parent?. If the language provides no delegation mechanism, is there a propagation mechanism supporting life-time sharing?.

Regarding extensions, delegation and sharing, when the language provides delegation, does it achieve "property sharing"? That is, an extension object and its parent can be seen as different parts of the representation of the same domain entity? Or "value sharing"? Meaning that the notion of extension is used to express the sharing of attribute values and methods between two objects representing different entities of the application domain.

The results of this type of classification are summarized in table 2.2

	SELF	Object Lisp	Garnet	Amulet	Agora	Moostrap
Distinction between variables and methods	no	yes	no	no	yes	no (slots)
Creation ex-nihilo	yes	no	yes	no	no	yes
Dynamic modification of object structure	yes	yes	yes	yes	no (possible at meta level)	yes
Cloning	yes	yes	no	no	yes	yes
Extension mechanism	yes	yes	yes	yes	yes	optional
Propagation mechanism	no	no	no	no	no	no
Single / multiple parents	multiple	single	multiple	simple	simple (mixins)	simple or multiple
Dynamic parent modification	yes		yes		no	yes
Interpretation of extension mechanism	property sharing	property sharing	value sharing	4 kinds sharing	encapsulated inheritance	property sharing

	NewtonScript	Kevo	Omega	Obliq	Yafool
Distinction between variables and methods	no	yes	yes	no	no
Creation ex-nihilo	yes	no	no	yes	yes
Dynamic modification of object structure	yes	yes	yes only for prototypes	no	yes
Cloning	yes	yes	yes	yes (multiple)	yes
Extension mechanism	yes	yes	no	no	yes
Propagation mechanism	no	yes	yes	no	no
Single / multiple parents	double	—	—	—	multiple
Dynamic parent modification	yes	—	—	—	yes
Interpretation of extension mechanism	property and value sharing	—	—	—	value sharing

Table 2.2: PBL comparison according to primitive mechanisms

2.3.4 Classification According to Program Organization

Another way of classifying PBLs, also according to [CDB99], derives from studying the way in which programs are organized. Even though it seems to contradict the basic principles of prototype theory, it is necessary to organize programs by grouping similar objects. Several possible solutions have been found through the years, which give rise to the classification attempt explained in this section; they describe how an individual object's state is linked to its behavior or to the part(s) that share(s) in common with other similar objects.

One kind of object and one kind of link Applies to languages where all the objects are homogeneous (e.g., there are no distinctions between them), and delegation is used for sharing. All objects are mutable, can be used as parents and can be cloned. The only link is the *parent-of* link.

Two kinds of objects, one kind of link A language with one kind of link and one kind of object tends to evolve towards one with two kinds of objects. Besides prototypical objects, some special, distinguished objects appear, probably supported at the language level. They are immutable or abstract, and may not be cloned or used as parents. The traits-based programming model is an example of this class of languages.

Two kinds of objects, two kinds of links This refers to languages where one of the links is for delegation and implementing sharing, and the other introduces a structural description, similar to an *instance-of* link. The two kinds of objects are the prototypical ones and a new object that allows sharing of structural information between structurally identical objects, for instance, the *maps* in SELF.

One kind of object, two kinds of links This type of languages seem to derive from the "two kinds of objects, two kinds of links" type, where a map object is replaced by a slot in the standard objects, containing a *descriptor*, a kind of slot dictionary. This is very similar to the implementation of a class-based language, except that it lacks a sharing mechanism between descriptors that would have a semantics similar to inheritance.

Figure 2.3 shows some examples of languages adhering to the classification explained in this section.

2.4 SELF: The Power of Simplicity

The current section details the most important features of SELF, presenting a technical overview of the language, the organizational structure of its

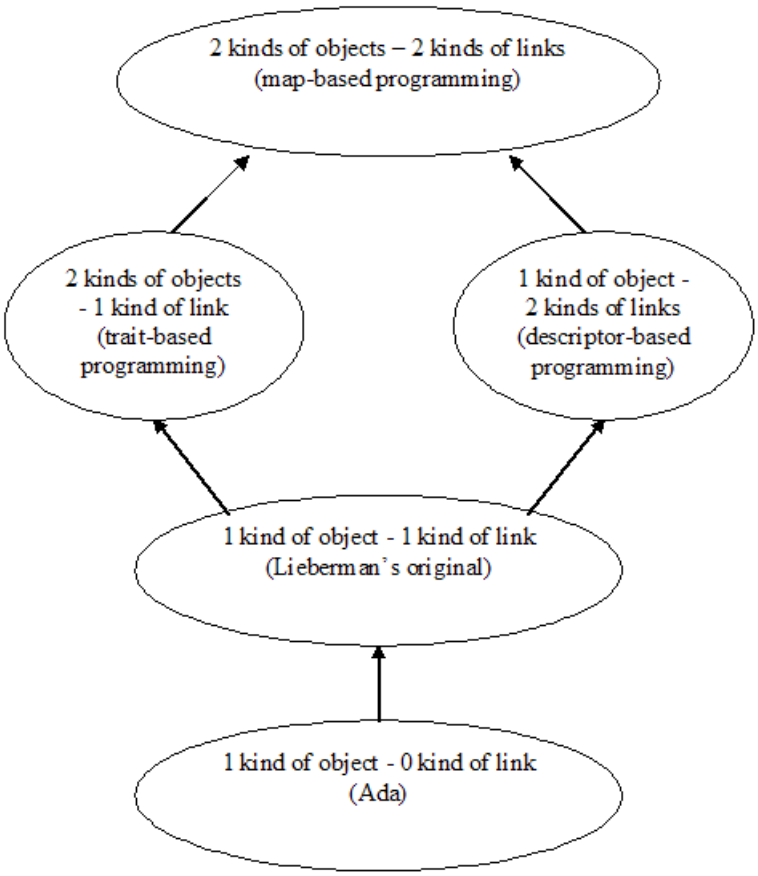

Figure 2.3: Examples of classification according to program organization

programs, an introduction to the "transporter" (the mechanism for saving SELF programs as source code) and an overview of Mango, a parser generator for SELF which was used during this research.

2.4.1 Technical Overview

Self [US87] is a "pure" object-oriented prototype-based programming language and an associated visual programming environment. Designed by David Ungar and Randall Smith in 1986, Self was initially implemented for Solaris at Stanford University in 1987. The project moved to Sun Microsystems Labs [Sel] in 1991 and further developments continued on the language, until the dissolution of the SELF research group at Sun. However, the project is still active, and in recent years there have been releases of the language ported to Mac OS X; the latest version at the time of this writing is 4.2.1. It is worth mentioning that the SELF interest group [Int] is a great source of information and help for anyone interested in learning about SELF.

This academic language has been used as an instrument for language, environment, user interface and implementation research, encouraging an exploratory programming style. Some of its most interesting features include creation *ex-nihilo*, cloning, usage of slots for unifying variables and methods into a single construct, message passing as the fundamental operation, delegation with late binding of self, dynamic typing, dynamic and multiple parent modification and powerful reflective capabilities.

Blending State and Behavior

In SELF encapsulation of state is enforced, thanks to the fact that there is no direct way to access a variable, instead objects send messages for accessing and modifying a data slot. This makes inheritance more powerful: In an extension object, it is possible to redefine the access and modification messages of a data slot. For example, by replacing the contents of the assignment method of a data slot with an empty operation the slot becomes read-only. If some data needs to be shared between a group of objects, it can be placed in their parent object: this is similar to having class variables in a CBL.

Closures and Methods

SELF provides its users with the ability to use blocks (closures, λ-expressions) like in Smalltalk, allowing the definition of new control structures; blocks are represented by an object containing an environment link and methods named `value`, `value:`, `value:With:` etc., depending on the number of arguments.

Local variables. The local variables in SELF's methods are stored using slots. Interestingly, SELF's methods are *prototypes* of activation records, and as such they are copied and invoked when executed. The local variables are allocated by reserving slots for them in the prototype activation record.

Environment link. Generally, a method contains a link to its enclosing closure or scope. In SELF, the method's *parent* link performs this function, and if a slot is not found the lookup continues through the outer scopes following the parent link. When a method is invoked its prototype it is copied and the parent link is set to the message's receiver. Also, when accessing local variables the message lookup for the implicit "self" receiver starts at the current activation record, but the receiver of the message is set set to be the same as the current receiver. Thanks to this technique, the procedure for accessing local and instance variables and method lookup is unified in SELF.

Delegation

Delegation means that when an object receives a message, it first attempts to respond to the message using its own behavior. At failing, the object automatically forwards the message to its parents. The delegation link allows state and behavior of the parent to be shared between parent and child.

As between inheriting classes in a CBL, delegation supports late binding of the self pseudo variable. Messages send to self in the parent will "come back" to the object that originally received the message.

A parent is just an object that resides in a slot named with a trailing asterisk and makes the child inherit all the slots of the parent slot. When two objects choose the same object to be their parent, the parent gets shared between the two children: every change made in the parent by one child will also be visible to the other child. This is commonly called Parent Sharing and is something most PBLs feature. Another kind of sharing that is specific to Self might be called Child Sharing (or multiple inheritance). Child sharing occurs when two or more parent objects share the same child object, i.e. when an object decides to name two or more parent slots (i.e. it has two or more slots with an asterisk).

A drawback of cloning objects is that all the slots are copied, even those (like methods) that could be shared between the object and all of its clones. To avoid copying behavior (next to state) every time an object is cloned, the Self group introduced traits objects for storing the shared behavior in an object and let the cloned objects inherit from it. Using traits objects can be seen as class-based programming in a prototype-based language.

Traits encourage the creation of delegation hierarchies that look like inheritance hierarchies: their highest levels are made of traits and the concrete

objects are found at the leaves. Traits-based programming leaves more flexibility in the creation of objects, and traits are not classes [CDB99]. A very interesting property of traits is that testing whether two object support the same protocol is reduced to testing if the two objects' traits are the same or have a common parent; this fact was used heavily during the course of this research, as it shall be seen in the following chapters. In a sense, traits-based programming emphasizes the notion of similar behavior among objects [CDB99].

Syntax

The syntax for a textual representation of SELF objects closely resembles Smalltalk's, with a few exceptions. For instance, the receiver is omitted when it is "self", the return value of a method is always the result of the last expression, keyword messages associate from right to left and case is used to make keyword message-sends easier to read: the first keyword must be a lowercase, and subsequent keywords in the same selector must be uppercase.

A few new elements have been added to the syntax of the language. For example, there is a slot list syntax for creating objects inline that, if present, must be nestled in a pair of vertical bars; SELF objects are written between parenthesis and include a dot-separated list of slots and in the case of methods, also include code. There are several forms for slots:

- A selector by itself denotes *two* slots: an accessor slot initialized to *nil* and an assignment slot with the same name of the accessor slot plus a trailing colon, initialized to the assignment primitive. (denoted by ←) For example, the object `(|name|)` contains two slots: one called **name** containing *nil*, and another one called **name:** containing ←.

- A selector followed by a left arrow (`<-`) and an expression also denotes two slots: one slot initialized to the value of the expression and the corresponding assignment slot. In other words, it is like an initialized variable.

- A selector followed by an equals sign "`=`" and an expression denotes only the accessor slot with the constant value resulting from the evaluation of the expression.

- A unary selector (identifier) preceded by a colon defines an argument slot of a message. For example, `[ | :x. :y | x + y ]` defines a block with two arguments, `x` and `y`.

As an example, the code in figure 2.4 implements a very simple `point` prototypical object. The point inherits from the traits clonable that stores the common behavior and state of clonable objects, and has two data slots containing the x and y coordinates. The remaining method slot contains a

```
(|
    parent* = traits clonable.
    x <- 0. y <- 0.
    addPoint: aPoint =
      ((copy x: x + aPoint x) y: y + aPoint y)
|)
```

Figure 2.4: Example of SELF syntax

method for adding two points, by copying point and initializing it with the added x and y coordinates.

Advanced Concepts

There are some features of the SELF language that are not commonly found in PBLs and should be considered "advanced". They can be quite powerful and a good SELF programmer must be aware of their existence. A brief description of such features follows, and a more detailed explanation can be found in [Ung02] and [OA00].

Multiple Inheritance An object can have more than one parent slot; when a message is received the delegation mechanism finds the parent object that can respond to the message, if more than one object implements the same message an error occurs.

Resends The resend mechanism allows the code inside a method to explicitly delegate a message send to the parent slot of the object, or to specify to which of its parents the message should be delegated to.

Dynamic Parents The parent slots of an object can be defined as assignable, and as such the parent object that they are pointing to can be changed simply by changing the slot to point to another parent.

Copy-Down When an object is extended, it is necessary to copy-down its data slots to its children, this is due to a limitation of SELF's compiler that cannot optimize for dynamic inheritance. This is similar to "subclassing" an object.

Mirrors Mirrors are SELF's reflective mechanism, and they allow the programmer to examine and manipulate objects in almost any conceivable way: any kind of slot can be added, removed, or modified; the parent hierarchy can be traversed and modified, etc. A mirror on object `x` can be obtained by sending the message `reflect: x` to any object that inherits `defaultBehavior`. All the changes applied to a mirror object also get applied to the original, "reflectee" object.

2.4.2 Organizing Programs Without Classes

The advantages of using prototypes in an object-oriented language have been enumerated before in this document. However, one of the fundamental questions that must be asked is: How to organize a program without classes?. In section 2.3.4 several alternatives were shown, and this section shows the particular way in which programs can be organized in SELF. Notice that this subject creates a very important distinction between different PBLs, as there is no "standard" or universal solution to the problem of organizing programs. Therefore, several of the possible solutions will be specific to SELF.

The authors in [CDB99] suggest that SELF should be considered a "two kinds of objects, two kinds of link" type of PBL, because it has maps for sharing structural information between structurally identical objects. The author of this document disagrees with that classification; it is true that SELF uses maps on its underlying implementation, but this is made for optimization purposes and it is never evident from the programmer's perspective. What is more, maps are not first-class objects and cannot be manipulated directly; maybe they *can* be manipulated by direct use of the primitives of the language, but even so that is discouraged and would be considered bad style. SELF is more a "two kinds of object, one kind of link" type of PBL, and the traits-based programming model is encouraged. The SELF group came up with several ideas for sharing behavior between objects [DUH91]:

Intra-Type Sharing In its simplest form, a data type is implemented as a prototype object holding the state with a parent pointer to a traits object, holding its behavior.

Inter-Type Sharing New data types can be defined as differences from existing data types. One possibility would be by creating a new prototype object for holding the state with a parent pointer to a new traits object that is a child of an existing traits object. Here, the new data type is defined by *refining* a traits object.

Representation Sharing As an augmentation of the inter-type sharing, it is also possible to refine a prototypical object holding state by using the copy-down mechanism discussed in section 2.4.1. Here, the representation extension takes into account data parents.

Using dynamic, multiple parents also counts as a way for sharing behavior. It was discussed in section 2.4.1.

The SELF group [DUH91] also defined a way for naming and categorizing objects:

Name Spaces Programs need to refer to well-known objects from different places in the system. To provide this functionality, SELF defines name

space objects whose sole function is to provide names for well-known objects. The name of an object in a name space is simply the name of the slot that refers to the object. The name spaces in SELF are rooted at the `lobby` object since most objects inherit from it, and after an object has been "installed" in the lobby it is said to be well-known and can be accessed from any object that inherits from the lobby.

The lobby is further sub-divided in the `globals`, `traits` and `mixins` inheritance hierarchies that group respectively the prototypes, behavior and "mixable" behavior objects of the system. There is no real difference between the three types of objects, they are implemented using the same mechanisms, although they will be used for different purposes. For example, in general it does not make sense to clone an object in the `traits` or a `mixins` hierarchy, whilst the objects in `globals` are intended to be cloned.

Categories As a further way to organize the contents of an object, it is possible to define categories (much like the protocols in Smalltalk) and put slots "inside" such categories. The categorization mechanism is implemented with annotations (see the section 2.4.3).

2.4.3 The SELF Transporter

The prototype-based paradigm of programming encourages people to construct programs by directly manipulating concrete objects. SELF's visual development environment helps to achieve this objective, providing the programmer with a graphical representation of the objects in the system, which can be easily changed and manipulated. This representation of an object system is not a textual one, and usually a programming session is stored using binary image files. However, in general the image files are not portable between instances of SELF installed in different machines, and it is necessary to use a textual persistence mechanism that allows a program to be re-created in other worlds: the transporter [Ung95].

The key to transporting objects lies in the ability to annotate an object, that is, adding extra pieces of information to whole objects or to individual slots. This information is used mostly by the programming environment; although any object can be an annotation, the SELF syntax only supports the textual definition of string annotations [OA00]. In particular, the annotations are used for specifying a slot's module, and also for annotating a copied-down object with the source of the copy (copy-down parent).

A couple of explanations should be made at this point: first, not all the objects currently present in SELF's image are saved, only those that are accessible from the lobby. Second, the designers of the transporter chose to implement persistence on a per-slot basis, i.e., for each slot it is necessary to specify its module, as slots are the minimum unit of functionality in SELF.

A module is a unit of persistence in SELF: all the slots annotated to belong to the same module are stored together in the same text file, even if they belong to different objects. Also it is possible that one object belongs to several modules, as each of its slots could be placed in different modules.

The process of supplying all the necessary annotations for saving a module can be a bit complex, as is the algorithm that uses the extra information to write out a slot, but it is explained in detail in [Ung95]. What is really important is understanding the five missing pieces of information that must be provided to determine how to write a slot: which module to use, whether to transport an actual value or a counterfactual initial value, whether to create a new object in the world or to refer to an existing one, whether an object is immutable with respect to transportation, and whether an object should be created by a low-level, concrete expression or an abstract, type-specific expression.

The flowchart in figure 2.5 illustrates the order in which the transporter queries the annotations in order to make its decisions.

As a final remark for this section, it should be noticed that the transporter is the biggest source of frustration when working with SELF. It is all too easy to put incorrect values in the annotations necessary for saving an object, and there seem to be some bugs in the transporting mechanism (at least in the Mac OS X version). The consequences? At the moment of filing-in a saved file, it is possible that some objects are corrupt, have incorrect values in some of their slots, or the filing-in process fails altogether. Also, in theory the order in which new slots are loaded into the image should not be important, but in practice some serious errors occur if the transporter does not find an object because it has not been defined yet. Extreme caution should be taken when using the transporter, and it is highly recommended to use an external version control system such as SubVersion or CVS to store the file-outs.

2.4.4 Mango: A Parser Generator for SELF

Mango [Age94] is SELF's parser generator. It is fully integrated with SELF's world, just like the parsers it produces. Both Mango and the generated parsers are SELF objects themselves. It has several interesting features that go beyond what is offered by other more conventional parser compilers such as YACC. For instance, Mango grammars are structured and map directly onto parse trees, and the parse trees are built automatically and can be decorated with attributes after the parsing is completed, instead of just providing hooks for calling reduce actions during parsing.

A structured context-free grammar is a grammar for which each non-terminal has exactly one production and each production is structured. A structured production has one of the following five forms:

Construction: `A ::= X1 X2 ... Xn`

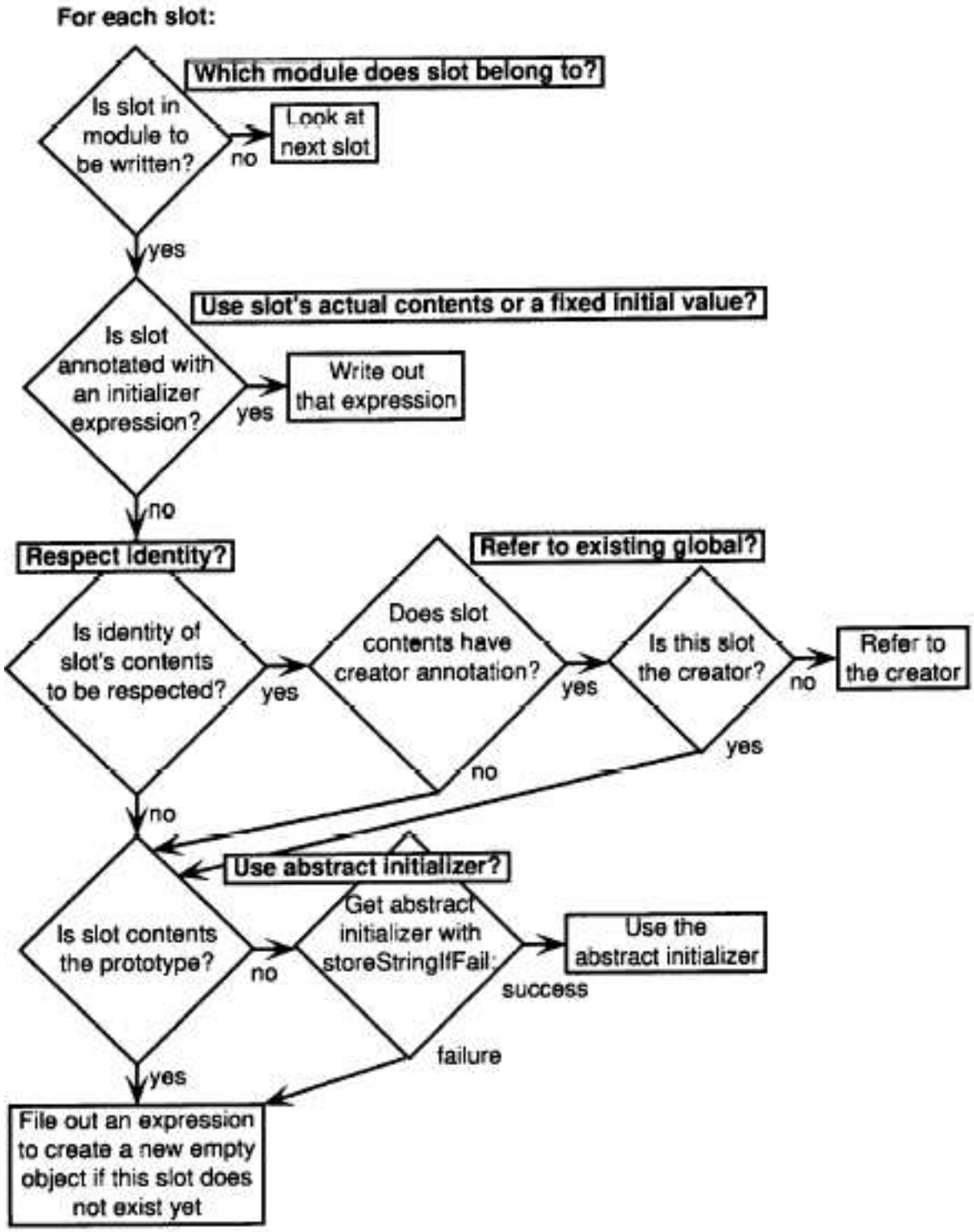

Figure 2.5: Transporter Flowchart

"The nonterminal `A` derives the sequence `X1 X2 ... Xn`."

Alternation: `A ::| X1 X2 ... Xn`

"The nonterminal `A` derives one of `X1, X2, ..., Xn`."

Non-empty list: `A ::+ E S`

"The nonterminal `A` derives a non-empty sequence of the form `ESES ... ESE`." `E` is the element and `S` the separator of the list derived by `A`. The separator is optional; if it is left out, `A` derives `EE ... E`.

Possibly-empty list: `A ::* E S`

"The nonterminal `A` derives a possibly empty sequence of the form `ESES ... ESE`." In other words, `A ::* E S` either derives ε or something that is also derived by `A ::+ E S`. The separator is once again optional.

Optional: `A ::? E`

"The nonterminal `A` derives `E` or ε."

The process of building a parser is as follows: Mango takes grammars (specifying both syntax and lexical parts) as input and produces parsers. The parsers, in turn, take strings as input and produce parse trees.

A new parser is obtained by sending the message

```
mango parsers stGrammarParser copy
```

Then, the message `parseFile:` can be sent to the returned object, passing the name of a file with a grammar as the argument; finally, the generated parser can be obtained by sending `output makeParser` to the same object.

Once a new parser has been built as explained above, it can be used by sending it the message `parseString:` or `parseFile:` with the appropriate argument. The result is a parse tree node, corresponding to the start symbol in the grammar. The node can be evaluated by sending it the appropriate message, which depends on the custom behavior that was added to that node.

The result of parsing a string or a file is a parse tree with a structure corresponding to the grammar used to generate the parser. Each node in the tree corresponds to a nonterminal symbol in the grammar, with an interface (the set of messages it can respond to) determined by the form of the nonterminal symbol and any custom behavior that may have been added to it. The specific details can be found in [Age94].

Perhaps the most interesting part, is the fact that custom behavior can be added to mango parse trees by means of a "behavior file". Mango will process this file during parser generation and ensure that the prototype and/or traits parse tree nodes possess the behavior.

The behavior file is just the textual description of a single SELF object, the *behavior object*. The behavior object has a number of slots whose names are also the names of nonterminal symbols in the grammar; the contents of these slots get added to the traits object of their respective nodes. It is also possible to add behavior to the prototype object of a parse tree node.

Chapter 3

Rule-Based Inference and NéOpus

This chapter introduces the fundamental concepts behind the theory of rules and rule-based languages as a means of representing and reasoning about human knowledge.

First, a comparison is made between the two basic formalisms (more can exist and variations thereof) for expressing rules: production rules and predicate logic (3.1). The next section describes the two main methods for chaining rules: forward-chaining and backward-chaining (3.2). Later on, section 3.3 describes hybrid systems that result from the blend between rule-based reasoning engines and object-oriented programming languages. Section 3.4 details the Rete algorithm, an efficient way to implement a forward-chaining production rule system. Finally, section 3.5 describes Opus and NéOpus, two particular examples of hybrid systems.

3.1 Formalisms for Expressing Rules

Rule-based reasoning is a specific topic of study in the field of Artificial Intelligence, its main objective is representing human knowledge as a set of *rules*. A rule is a statement of the form

```
If <condition> Then <action>
```

or alternatively

```
<conclusion> If <action>
```

(both are equivalent). This might seem like an `If-Then` statement in a procedural language, however the main difference here is the fact that rules are not activated ("fired") in a predetermined order, instead, the inference engine supporting the rules determines which rules can be fired and in which order.

There are two main formalisms for expressing rules: **production rules** – where the rules are called *productions*– which give rise to *production systems*, and **first-order predicate logic** –where the rules are *logic clauses*– which originate *logic-based systems*.

The set of all productions in a system is called *rule base* or *production memory*. The data representing the knowledge to which the rules apply is called *fact base* (a.k.a. *working memory* or *knowledge base*). The `If` part of a production is called LHS (left-hand side) and its `Then` part is called its RHS (right-hand side).

There are rule languages specifically created for dealing with either type of formalism. There are also hybrid languages (see section 3.3) that combine a "normal" language (typically object-oriented) with a rule language.

3.1.1 Production Rules

Production rules present empirical associations between patterns of data and actions that should be performed as a consequence. As such they also have the format conclusion if condition, although the conclusion can sometimes be interpreted as recommending an action rather than concluding that a certain proposition is true. The patterns and concrete data are usually represented as object-attribute-value triplets. Bear in mind that the *objects* mentioned in the definition are constant symbols which denote a conceptual object, and not full-fledged objects in the sense of entities that encapsulate data and behavior in object-oriented languages.

An interpreter executes a production system performing the following operations [For82]:

1. *Match.* Evaluate the LHSs of the productions and find out which are satisfied, given the current contents of the knowledge base.
2. *Conflict Resolution.* Select one production that satisfied the LHS, or halt the interpreter if there is not any available.
3. *Act.* Perform the actions in the RHS of the selected production.
4. Goto 1.

OPS5 [BFKM85] is a typical example of a rule language belonging to the Production Rules formalism.

3.1.2 First-Order Predicate Logic

First-order predicate based logic is the basis of logic-based systems. *Propositional* logic is concerned with establishing the truth value of combinations of atomic propositions, which each have a definite truth value. *Predicate logic*, on the other hand, exposes the internal structure of propositions and

analyzes them into predicates that are applied to individuals, which are objects the world consists of and have individual properties. Universal and existential quantification is permitted over the individuals [DH04].

Logic-based systems use logic as their programming language, with Prolog [Fla94] as the typical example of a rule language belonging to the First-Order Predicate Logic formalism.

3.2 Rule Chaining

The way in which an inference engine activates a rule is known as rule chaining. This process is fired when new data arrives that matches either the condition or conclusion activation pattern of a rule.

3.2.1 Forward-Chaining

Forward chaining starts with the data available and uses the inference rules to conclude more data until all possible facts are inferred and asserted, a halting statement is encountered, a number of cycles have passed or a specific goal has been reached (the stopping strategy depends on the engine). An inference engine using forward chaining searches the inference rules until it finds one in which the LHS is known to be true. It then concludes the RHS and adds this information to its data. Because the data available determines which inference rules are used, this method is also called "data driven."

Typically, synthetic tasks such as design, scheduling and assignment are usually solved by forward chaining [SAA$^+$00] . The Rete algorithm (explained in section 3.4) was designed to be a fast implementation for forward-chaining engines. Among others, OPS5, Opus, and NéOpus carry out this type of rule activation.

3.2.2 Backward-Chaining

Backward chaining starts with a list of goals (conclusions) and works backwards to see if there is data (conditions) which will allow it to conclude any of these goals. An inference engine using backward chaining would search the inference rules until it finds one which has a RHS that matches a desired goal. If the LHS of that inference rule is not known to be true, then it is added to the list of goals.

Note that chaining is not the same as reasoning: backward reasoning attempts to prove a particular goal whereas forward reasoning infers new knowledge starting from the existing knowledge. In principle, either chaining modes are able to express both kinds of reasoning strategies for solving problems [Jac86], but using the corresponding chaining mode for a particular reasoning strategy is more expressive.

Typically, analytic tasks such as classification, diagnosis and assessment are goal-oriented and solved by backward reasoning [SAA+00]. Prolog uses this type of rule activation.

3.3 Hybrid Systems: Objects and Rules

In the current discussion, a hybrid system is one that integrates both a rule-based language and an object oriented language. Many of such hybrid languages have been implemented through the years, and the author in [DGJ04] and [DH04] provides an exhaustive survey of more than 15 languages, focusing on the investigation of the level of integration between the two languages.

Frame-based languages were excluded from the survey because "frames are usually prototype-based" and "nowadays, the state-of-the-art object-oriented programming languages for software engineering are still class-based. Therefore, we only consider these kind of languages." [DH04]. However, as was mentioned in section 2.3.1 frames are only regarded as *predecessors* of prototype-based languages, and not fully mature members of the prototype-based programming language paradigm.

3.4 The Rete Algorithm

There is one recurring problem when implementing a production system: How to efficiently compare a large collection of patterns (premises in the context of production systems) to a large collection of objects?. One of the most well-known solutions comes in the form of the Rete algorithm [For82], developed by Charles L. Forgy in 1974; it was adapted and used in Opus and NéOpus (see 3.5), therefore it is important to understand the original algorithm to fully comprehend the design and implementation of either one of those inference engines.

A detailed explanation and implementation of Rete (which means "network" in Latin) is available in the white paper "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem" [For82], this section summarizes parts of that paper.

3.4.1 Overview

During the inference and evaluation cycle of a production system, the interpreter has a collection of objects and a collection of patterns (conditions in a rule that must be satisfied), and it is necessary to find every object that matches the pattern — in other words, which productions have satisfied condition parts. The naïve approach of matching all the patterns against all the objects during the inference process is unacceptable in terms of efficiency,

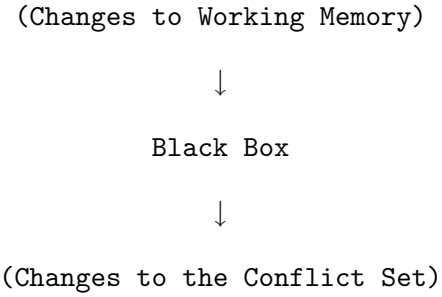

Figure 3.1: Pattern Matcher as a Black Box

as this procedure can be slow when large numbers of patterns or objects are involved.

The Rete algorithm was designed to make many pattern/many object pattern matching less expensive in terms of number of matches performed, and it is useful for production system interpreters.

In a production system interpreter, the output of the match process and the input to conflict resolution is called the *conflict set*, which is a collection of ordered pairs of the form

```
<Production, List of elements matched by its LHS>
```

also called *instantiations*. The Rete algorithm computes the conflict set by comparing a set of LHSs to a set of elements in order to discover all the instantiations.

3.4.2 How to Avoid Iterating over Working Memory

The key to avoid performing constant iteration over every object in the working memory when trying to match a pattern, lies in storing information between cycles. To determine whether a given pattern matches one element in working memory, it might seem necessary to iterate over all the elements, however the iteration can be avoided altogether by storing with each pattern a list of the elements that it matches. The lists are updated when the working memory changes: if a new element enters the working memory or if an existing element is modified or is removed, the interpreter finds all the patterns that match it and updates their respective lists by adding, modifying or removing the object. In this way, the pattern matcher component of a production system never has to examine working memory, and can be viewed as a black box with one input and one output, as shown in figure 3.1.

The descriptions of working memory changes that are passed into the black box are called *tokens*. A token is an ordered pair of a *tag* indicating the operation to be performed (adding, removing) and a list of data elements.

Notice that the modifications are implemented by removing an element and then adding it with its new values.

3.4.3 How to Avoid Iterating over Production Memory

The previous section showed how to avoid iterating over the fact base in a production system, effectively avoiding the need to traverse all the objects in a system when something changes in the fact base. Then, the next question would be how to avoid iterating over the rule base?, so it would not be necessary to traverse all the productions in the system when a change occurs. The answer is, by using a tree-structured sorting network or index for the productions. The network is compiled from the patterns and is the principal component of the black box.

Compiling the Patterns

There are two kinds of tests that a pattern matcher performs when processing a working memory element: testing for *intra-element* features which only involve one working memory element, and testing for *inter-element* features, for the cases when a variable occurs in more than one pattern.

The pattern compiler builds a network by linking together nodes which test elements for these features.

> When the compiler processes an LHS, it begins with the intra-element features. It determines the intra-element features that each pattern requires and builds a linear sequence of nodes for the pattern. Each nodes tests for the presence of one feature. After the compiler finishes with the intra-element features, it builds nodes to test for the inter-element features. Each of the nodes has two inputs so that it can join two paths in the network into one. The first of the two-input nodes joins the linear sequences for the first two patterns, the second two-input nodes joins the output of the first with the sequence for the third pattern, and so on. The two-input nodes test every inter-element feature that applies to the elements that they process. Finally, after the two input nodes, the compiler builds a special terminal node to represent the production. This node is attached to the last of the two-input nodes. Note that when two LHSs require identical nodes, the compiler shares parts of the network rather than building duplicate nodes. [For82]

As an example, figure 3.2 shows two productions (expressed in the OPS5 language) together with the network compiled from them.

```
(P Plus0x
        (Goal ^TypeSimplify ^Object<N>)
        (Expression ^Name<N> ^Arg1 0 ^Op+ ^Arg2<X>)
-->      ...)

(P Time0x
        (Goal ^TypeSimplify ^Object<N>)
        (Expression ^Name<N> ^Arg1 0 ^Op* ^Arg2<X>)
-->      ...)
```

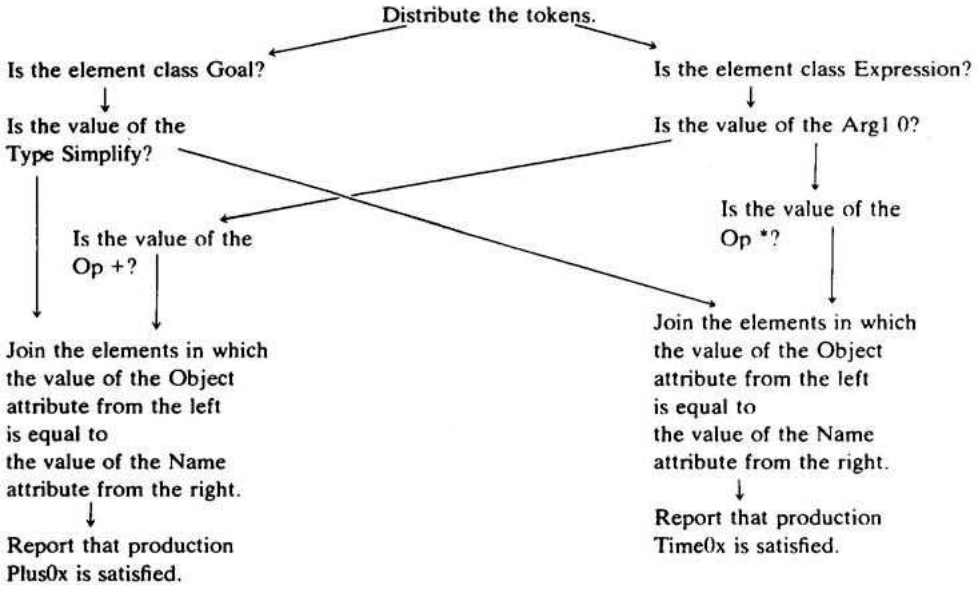

Figure 3.2: Rete Network for Plus0x and Time0x [For82]

Processing in the Network

The root node at the top of figure 3.2 represents the input to the black box. It receives tokens that are intended to be sent to the black box and passes copies of the tokens to all its successors.

The first nodes after the root node are the one-input nodes that perform intra-element tests, and can have more than one output. Each node tests one feature and sends the tokens that pass the test to its successors.

Two-input nodes compare tokens from different paths and join them if if they satisfy the inter-element conditions of the LHS.

Finally, a terminal node will only receive tokens that instantiate the LHS, because the nodes before it have already performed all the necessary tests.

Saving Information in the Network

The key point in Rete Network's algorithm is that it must maintain state information. Generally, such state is stored by the two-input nodes, each one of them contains two lists, called the left and right memories.

The left memory stores copies of the tokens that arrived at its left input, and the right memory holds copies of the tokens that arrived at its right input.

3.4.4 Completing the Set of Node Types

The network in figure 3.2 contained nodes of four types: root, terminal, one-input and two-input. As a minimum, two more types of nodes are necessary for building a useful Rete network.

Another type of two-input node is needed for negated patterns (negative premises). This node stores a count with each token in its left memory, indicating the number of tokens in the right memory that "allow consistent variable bindings" [For82]. The right memory's tokens contain the elements that match the negated pattern. The node only allows tokens with a count of zero to pass.

The final type of node is a variant of the one-input node, and tests working memory elements for constant features. This nodes compare two values from a working memory element, and are used to process patterns that contain two or more occurrences of the same variable.

3.4.5 Final Remarks on Rete

Rete was designed to be a fast algorithm, and as such the implementation details in [For82] are very low-level: the information is represented as binary words and manipulated at the bit level via an assembler-like programming language. It does not translate well into the more "modern" object-oriented

programming languages, and it is necessary to sacrifice some of the efficiency of the original algorithm by implementing it at a higher level of abstraction. However, the essential aspects of the algorithm can be preserved by respecting the requirements of avoiding unnecessary iteration over working and production memory.

3.5 Opus and NéOpus

Opus –A production system written in Smalltalk– and a later, expanded version of it called NéOpus were studied as a reference for implementing a hybrid system. This section introduces both systems.

3.5.1 Opus Overview

Opus is a tool for rule-based programming which integrates the production system paradigm with the Smalltalk-80 environment, and is modeled after the OPS5 expert system. It is described in detail in [AL87]. The biggest contribution of Opus was demonstrating how first-order, forward-chaining rules could be accommodated in Smalltalk in a seamless way, via an object-oriented realization of Forgy's Rete network algorithm.

Opus was designed with one main objective in mind: bringing together in one system two different programming and knowledge representation paradigms –objects and production rules–. The authors also intended to provide a full integration between the system, the environment and the language with complete freedom to use any Smalltalk expression in either the premise or action part of the rules. The system's architecture is composed of four main parts:

The Work Memory The set of categories defined by the user (Smalltalk classes) together with all the objects belonging to them (Smalltalk instances).

The Rule Base A special kind of classes whose methods are Opus' rules.

The Rules A textual representation of rules, with the following structure: one identifier, a variable declaration, a list of premises and an action part.

The Interpreter The component in charge of managing the execution of the rule base. In particular, it selects the rule(s) to be executed during each cycle of the inference process.

Of course, there are some important differences between Opus and OPS5. For instance, the messages for accessing attributes and some comparison messages (`=`, `<`, `>`, ...) will have a different syntax. But the real richness of Opus, compared with OPS5, derives from two fundamental differences:

- The premises can be expressed in terms of any Smalltalk message send.
- Utilization of functional links between objects (e.g., an object can be accessed by passing messages to another object, and needs not to be declared).

3.5.2 NéOpus Overview

NéOpus was built upon the ideas of Opus, as the Ph.D. research of François Pachet [Pac92]. The author reimplemented Opus with several improvements which will be detailed in the next section, creating a more powerful –but also more complex– inference system. Two very important principles were established and implemented throughout the system:

Principle "Any Object" Any object in Smalltalk's environment can be used in a rule

Principle "Any Expression" Any Smalltalk boolean expression can be used as a premise in the rules

Here lies the power of the inference engine: the rules can be stated over any object, expressing any condition.

New Features

NéOpus adds several new features to the basic Opus system:

- It is possible to establish single inheritance relationships between rule bases.
- A declarative control architecture is defined with the introduction of Metarules [PP94].
- Assertions and Goals as part of the rule language.
- 0-Order reasoning via global variables.
- Programming environment fully integrated with Smalltalk (Browsers, Inspectors, Views, ...).
- Activation Contexts for defining which instances of a class should be considered while evaluating a rule base.
- Local variables in rules, which also can be fired.
- Two modes of reasoning when considering the class inheritance: "simple" only takes into account the instances of a class, and "natural" which takes into account the instances of a class *and* all the instances of its (potential) subclasses.

Part II

Contributions

Chapter 4

Prototypes Meet Rules

The present chapter introduces a **F**orward-chaining **I**nference **R**ule **E**ngine (FIRE), the tool developed for exploring a hybrid system built on the combination between a production system and a prototype-based language. Section 4.1 motivates why we built a rule-based system in combination with a prototype-based object-oriented programming language and justifies some implementation decisions taken. Section 4.2 goes deeper into the implementation details of FIRE, in particular, explaining how the transition was handled for going from classes to prototypes in the context of a hybrid system. Section 4.3 enumerates the advantages and disadvantages of using a PBL as an implementation language. A brief set of instructions for using FIRE is provided in section 4.4, and finally Section 4.5 presents an outline of the inference and evaluation process in FIRE.

4.1 Motivation

Chapters 2 and 3 introduced two different paradigms for representing knowledge and inferring information about it, namely prototype-based languages and rule-based systems. The survey in [DGJ04] shows several hybrid systems where object-oriented programming languages and rule-based systems have been integrated. All of the object-oriented languages in that survey are class-based, and those hybrid systems based on frames —prototype-based languages for knowledge representation— were excluded from the survey. However, frames are regarded as predecessors, but not *true* members of the prototype-based language paradigm [CDB99].

It is also true that there have been numerous (but old) approaches for combining frames and rules, but in recent years hybrid systems have been built around the class-based paradigm of knowledge representation, leaving apart the prototype-based paradigm, even though it offers many potential advantages over classes for representing knowledge (like the ones discussed in section 2.2.1).

The background in the two previous paragraphs was necessary to make this statement: To the author's knowledge, in present times there are no hybrid systems created from the combination of a production system and a modern prototype-based programming language. This is a gap in the field of knowledge representation and inferencing that is worth exploring and filling. For this purpose, **FIRE** was created: A **F**orward-Chaining **I**nference **R**ule **E**ngine for a Prototype-Based Language. This tool will allow exploring the challenges of building such a hybrid system, and will allow investigating the expressive power of a rule language built on top of a prototype-based language in the context of meta-programming, as described in chapter 5

Why SELF and NéOpus ?

The first implementation issue for building FIRE was which PBL should be chosen for the job, and what hybrid system to use as an implementation guide.

Even though there are many PBLs, SELF is one of the most widely known prototype-based languages, and has been developed and improved during more than fifteen years. As was shown in the classification sections (see 2.3.2, 2.3.3 and 2.3.4) it has features, primitives and organizational mechanisms that are commonly found in most PBLs and that makes it a good example of the prototype-based paradigm; this and the fact that it has reached a high level of maturity and stability, make it ideal for using it in a research about PBLs, such as the subject of this thesis.

Similarly, there are lots of class-based hybrid production systems. This is an important starting point, because many of the (possible) implementation problems encountered when building a forward-chainer on top of another language have already been solved; besides it is always possible to learn from other people's research and findings *before* attempting to provide a new, original contribution to any field of knowledge.

In the context of the current research, the results obtained in the NéOpus system are very relevant when it comes to implementing a rule-based system on top of a prototype-based language such as SELF, and make it ideal as an implementation guide for FIRE. From a purely practical point of view, it is very important that the source code of NéOpus is available free of charge [NeO] and can be studied and used with complete freedom.

At a deeper level, NéOpus embodies the results and research of three generations of rule-based systems: the experience accumulated from the development of OPS5, Opus and NéOpus itself can be profited in several levels. For instance, Opus and then NéOpus showed how to realize the Rete algorithm in an object-oriented system, overcoming the original, very efficient but very outdated implementation of Forgy [For82]. Also, at the heart of NéOpus lies the simple but fully functional rule-based engine of Opus that can serve as a starting point to later build a more powerful

one. As a proof of concept, FIRE includes all the functionality described in the original Opus, plus some of the more advanced characteristics found in NéOpus (to be precise, the last three items enumerated in 3.5.2).

Finally, since Opus and NéOpus were implemented in Smalltalk, a typical and well-established example of a class-based language, an excellent opportunity arises to experiment and discover how to interpret the ideas and concepts of a program that relies heavily in the structure and organization of classes, into the more flexible —but less structured— world of prototype-based languages.

4.2 FIRE for SELF Implementation

This section outlines FIRE's structure in 4.2.1; the important implementation issues of building an inference engine over a prototype-based language are described in sections 4.2.2 and 4.2.3 alongside the solutions found to them. Also, an explanation of the language created for writing rules in FIRE is given in section 4.2.4.

4.2.1 Overview of FIRE's Structure

FIRE's inference engine is composed of several objects, with complex interactions between them. It is no use trying to summarize all the implementation details in just a few paragraphs, so only an outline of its structure will be given in this section.

At a high level, FIRE's structure is identical to that of NéOpus. It consists of a **kernel** with the most important objects of the system, a **network** with several type of nodes implementing the Rete algorithm and a **parser** for the rule language.

The key areas that set apart FIRE from NéOpus lie in the fact that they are realized under completely different paradigms: class-based versus prototype based. Section 4.2.3 presents a detailed discussion of this topic, emphasizing the aspect with the deepest level of differentiation, namely, the way of grouping together objects with similar characteristics.

Kernel

The objects in the kernel (shown in figure 4.1) implement the object representation of the elements in a production rules system, and the rules themselves. Figure 4.2 depicts the relationship between different objects in the system. The main responsibilities of each object are:

rule Provides an object representation of a rule and contains all the elements present in the textual representation of it, namely: name, variable declaration, premises and actions.

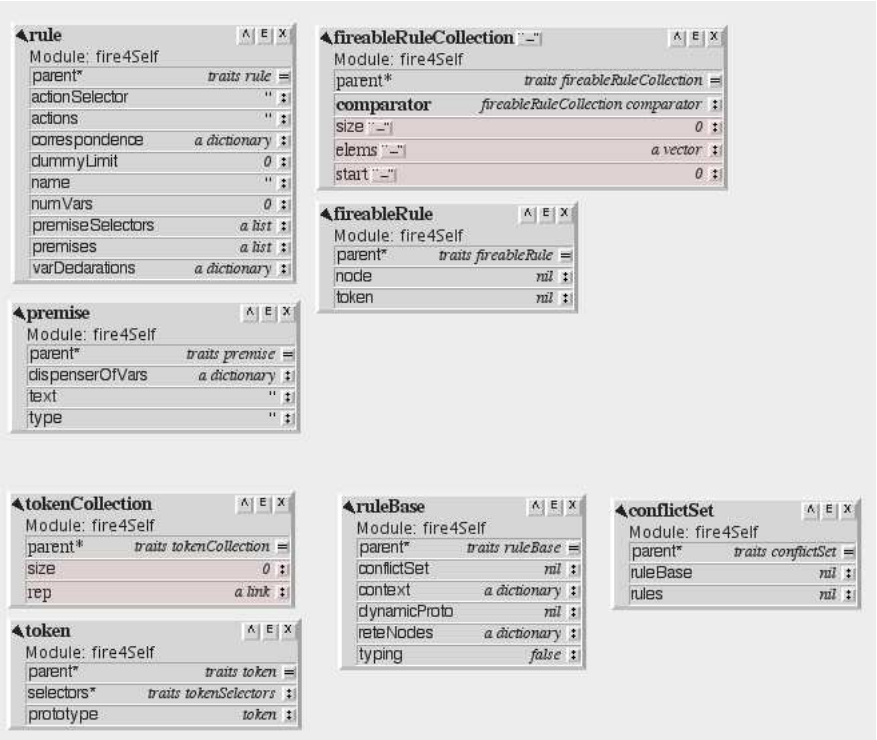

Figure 4.1: Kernel Objects in FIRE

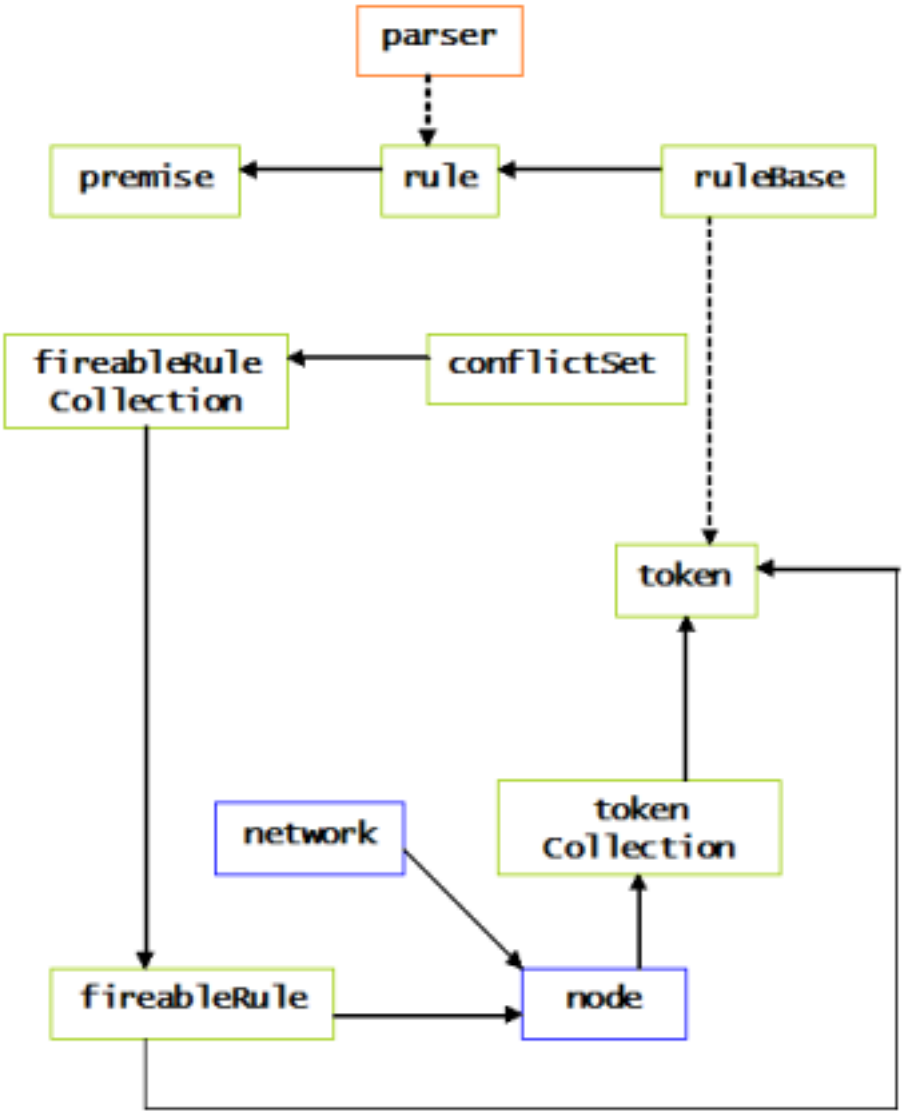

Figure 4.2: Relationship Between Objects in FIRE

premise Provides an object representation of a premise, it is aggregated in the rule object. It has information about the type of the premise and the dispensers it refers to. The possible types include general positive, general negative, positive with one free variable, negative with one free variable, positive with no free variables, assignment to a local variable and assignment to a fireable local variable.

fireableRule Encapsulates a pair of node-token objects; it contains a token that has succeeded all the tests in the Rete network and the final node in the network, the latter contains the action part of the rule that will be executed with the values of the token.

fireableRuleCollection Aggregates fireable rules inside a sorted sequence.

token Represents a change to the state of the network, it signals the addition, deletion or modification of an object in the system. It gets propagated in the network, and holds both the set of values bound to the declared variables and the tests that should be performed on them.

tokenCollection Aggregates tokens inside a list.

ruleBase The main object of the system; it is responsible of adding rules to the base, starting the evaluation process and managing an evaluation context.

conflictSet Contains a fireableRuleCollection and provides mechanisms for managing the execution of rules.

Network

The objects in the network (shown in figure 4.3) implement a Rete network, with a hierarchy of nodes specialized for different types of premises; Figure 4.4 shows the traits inheritance hierarchy between nodes, and figure 4.7 details an interesting feature of that hierarchy, namely, the use of delegation and multiple inheritance to avoid code duplication between nodes. The main responsibilities of each object are:

network Provides an entry point to the sequence of nodes associated with each rule; creates and interconnects the nodes that implement the Rete network. Each type of premise maps to a different kind of node, the kind of node created by the network depends on the type of the premise.

node Represents the most general type of node, all the others inherit from its behavior. It has mechanisms for testing, memorizing and propagating tokens through the network. It maps to the general positive premises.

Figure 4.3: Network Objects in FIRE

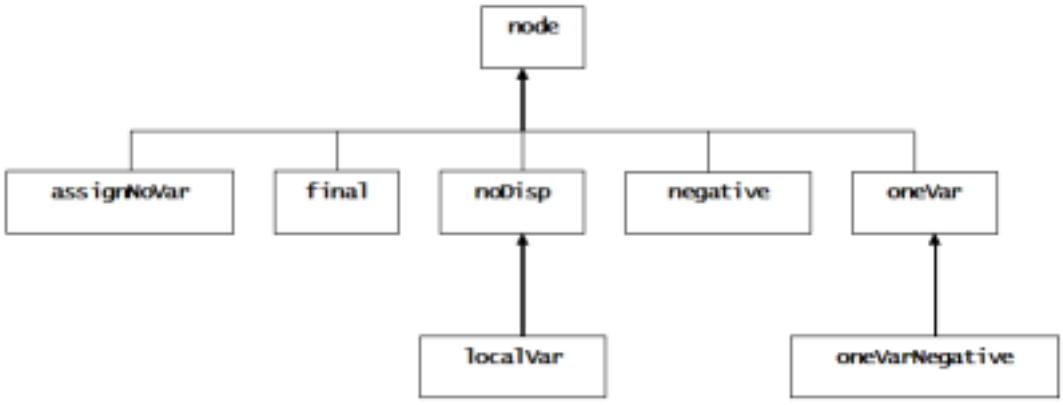

Figure 4.4: Node Hierarchy in FIRE

nodeNoDisp Maps to the premises of type positive with no free variables.

nodeOneVar Maps to the premises of type positive with only one free variable.

nodeLocalVar Maps to the premises that perform an assignment to a local variable.

nodeAssignNoVar Maps to the premises that perform an assignment to a fireable local variable.

nodeNegative Maps to the premises of type general negative.

nodeOneVarNegative Maps to the premises of type negative with only one free variable.

Parser

The final element of FIRE is its `parser` object. The parser was built with the help of Mango [MAN], using the grammar defined in appendix A. The parser generates a parse tree after a successful parsing, the tree has all the behavior methods defined in the file `ruleParser.behavior.self` of the FIRE distribution.

The parser is responsible of reading the textual representation of a rule and performing several transformations on the resulting parse tree:

- Pre-processing the text.
- Determining the internal name of the variables in the declaration part.
- Renaming the variables in the premises and action part accordingly.
- Modifying assignments so they return the assigned value.
- Expanding macros.

The reasons behind this transformations are implementation-specific, necessary to accomplish the final responsibility of the parser: creating rule and premise objects from textual representations of rules, in such a way that they can be evaluated by the system.

4.2.2 Adapting the Rete Algorithm

The core of FIRE's forward-chainer implementation lies in its adaptation of the Rete Algorithm (see section 3.4) for performing an efficient matching of the objects in the fact base against the premises in the rules. The SELF realization of Forgy's algorithm [For82] is based on the NéOpus implementation, in turn based in the Opus implementation, in turn based in OPS5's implementation. It is important to remember that Rete was originally designed

for evaluating rules in the OPS5 system, which represented knowledge and expressed premises and actions via a "pre-object" language, with objects more similar to records in Pascal than objects in the sense of the current object-oriented languages (i.e., encapsulated abstractions containing data and behavior).

By comparison, the fact base in FIRE is composed of SELF objects; the premises in the rule language can be written as any valid expression in SELF that returns a boolean value and the actions as any valid expression: hence the rules become very expressive, but implementing the underlying engine to support them is harder. Also, it is possible to reference more than one object in a single premise, and the role of the variables is reversed because they are no longer used to reference the value of attributes in objects, instead they reference the objects themselves.

The preceding differences have a great impact [Pac92] in the way Rete should be adapted for working in a *true* object-oriented language. In particular:

- There is no need for a discrimination network (one-input nodes). In FIRE, this is implicitly replaced by the link between objects and their dispensers.
- There are no n-ary relations; premises are expressed in terms of SELF message passing.
- The nodes must be capable of filtering more than one new object per premise (nodes with more than one dispenser).
- It becomes too difficult to factor out nodes for several rules sharing a common premise. For instance, how to discover that messages such as `x isNil` and `x == nil` or `a > b` and `b < a` are equivalent and should be represented by the same node?.
- It is impractical to represent the network and the tokens in exactly the same way suggested in [For82]: the data structures are too low-level and require the manipulation of information at the bit level. A higher level of abstraction is used, in the form of SELF prototypes and traits.
- For the same reason as above, it is impractical to linearize the network as a series of assembler-like instructions; instead, an object-oriented representation of the network is provided.

However, the two key ideas of the original algorithm are preserved, namely, avoiding iteration over working memory by storing with each premise the list of elements that it matches and avoiding iteration over production memory by compiling the premises in a network of nodes with memory.

4.2.3 Adapting NéOpus: From Classes to Prototypes

The biggest challenge of building FIRE was departing from the traditional class-based paradigm of knowledge representation used in hybrid systems. There were other, secondary implementation challenges like the ones mentioned in the previous section, but the most fundamental question to be solved was how to interpret the universal quantification of variables in a prototype-based world.

FIRE is a first-order reasoning engine, and all the variables declared in a rule are (implicitly) universally quantified. For instance, this means that a variable said to be of kind `person` refers to "all" the `person` objects in the system. In a class-based world the previous example can be easily translated to implementation terms: the kind of a variable would be its class, and "all" the objects of the same kind in the system would be the instances of that class. Classes, subclasses, instances ... all these concepts applied to the quantification of variables in a rule needed an equivalent in a prototype-based world.

Grouping Objects in SELF

In NéOpus, a *dispenser* refers to a class that sends new instances that bind to one of the variables in the declaration part of a rule. FIRE adopts the name of "dispenser", but adapts the concept to a prototype-based world. The variable declaration part of a rule in FIRE has the form:

```
| "prototype" p1 p2. "traits object" t1 t2 |
```

The symbol between double quotes it is said to be the *dispenser* of the variables it precedes. If a variable is declared to belong to a `"prototype"` (or alternatively, to a `"globals prototype"`) then *prototype-based grouping* applies to it, therefore the prototype and all of its copies are considered when binding values to the variable. On the other hand, if it is declared to belong to a `"traits object"`, then *traits-based grouping* is enforced, and all of the children of the traits object are considered when binding values to the variable, including children from different inheritance hierarchies that happen to share the same parent, but excluding children that are traits themselves.

Traits-based grouping is similar to the grouping of objects that results from considering a class and its instances (although it is possible to have cross-cutting grouping, because objects from completely unrelated hierarchies can share the same parent traits object), whilst prototype-based grouping is exclusive to a prototype-based world. The possibility of combining traits-based grouping and prototype-based grouping simultaneously in a rule is unique to FIRE, and holds the potential to be a very powerful feature.

Given that grouping objects is a fundamental part in FIRE, a special set of algorithms had to be developed for dealing with the subject.

```
rule name
  | variable declaration |
  premises
ACTIONS
  action part
```

Figure 4.5: Rule Structure in FIRE

Simple vs. Natural Typing and Inheritance

NéOpus introduced the concept of *typing*: a variable in a rule is said to be of "simple type" if it refers to all the instances of a class, and is of "natural type" if it refers to all the instances of a class *and* all the instances of its (potential) subclasses.

As one further generalization, the inheritance concepts of superclasses and subclasses are re-interpreted in the light of the two kinds of grouping of objects defined above: An *ancestor* is defined to be either the immediate parent trait(s) of a trait object or the immediate copy-down parent(s) of a prototype object, depending on the kind of dispenser. In a similar way, a descendant is the immediate child (children) traits object of a traits dispenser or the immediate copy-down child (children) prototype object of a prototype dispenser. Both ancestors and descendants are considered when "natural typing" is enabled in FIRE.

4.2.4 The Rule Language

A very simple language was developed for expressing rules in FIRE; the full grammar can be found in appendix A. This section describes the characteristics and general syntax of the language, alongside with some caveats. Every rule has the structure depicted in figure 4.5

The following subsections explain in detail each of the parts of a rule. Section 4.2.4.1 explains the naming convention for rules; section 4.2.4.2 shows the different types of variables that can be declared for using in a rule alongside with the naming convention for variables; section 4.2.4.3 specifies the SELF expressions that can be used as valid premises and finally section 4.2.4.4 details the types of premises that can be used in the action part of a rule together with the macros available in the language.

4.2.4.1 Rule Name

The rule name is simply any valid identifier in SELF: a string starting with a lower-case letter, followed by any number of letters (upper or lower-case),

numbers or the `"_"` character. This name is used as an identifier in FIRE, and allows accessing the object representation of the rule inside FIRE.

4.2.4.2 Variable Declaration

The variables that will be used in the rule are declared between vertical bars, in a list of dot-separated declarations. A declaration has two parts: the name of the dispenser between double quote characters (`""`) followed by one or more variables. A trailing dot after the last declaration is not allowed. As was detailed in 4.2.3, the dispensers can have one of two forms: `"prototype"` (equivalent to: `"globals prototype"`) or `"traits object"`, assuming that the referenced objects do exist in the system and can be accessed from the lobby. Additionally, non-firing local variables (i.e., variables that cannot be added, removed or modified in the actions part) can be declared if their dispenser is set to `"local"`, these are useful for holding temporary values when writing the rule.

Conceptually, three kinds of variables can be declared in the variable declaration part:

First-Order variables Those declared to be bound to a certain dispenser.

Local variables Those declared with the `"local"` dispenser. This kind of variable cannot be fired in the action part of the rule, and are intended to be assigned to the result of a message sent in the premises part

Fireable local variables Declared in the same way as the first-order variables, but intended to be assigned to the result of a message sent in the premises part (therefore it must have the same kind of dispenser as the object returned from the message). This kind of variable can be fired in the action part of the rule.

The example in figure 4.6 will make things clearer. There is one big caveat regarding the names of variables. Due to limitations in the implementation of the parser, the variable names inside a rule must not be substrings of one another, and they must not be substrings of any string in the premises or actions part. For example, the variable names `stud` and `student` will cause problems as the former is a substring of the latter. The simplest solution to this problem is following some naming conventions for variables, like using names with more than four characters and ending with a number. The parser does not check for the problem just described, so care should be exercised when writing the rules.

4.2.4.3 Premises

Premises are SELF expressions enclosed inside back quotes (the ‘ character) and separated by dots. A trailing dot after the last premise is not allowed. There are three types of premises:

```
  varTypeExample
|
    "string" stri1 stri2.          <- first-order variables
    "traits students" studentList. <- first-order variable
    "local" loca1.                 <- local variable
    "student" stud1                <- fireable local variable
|
    `loca1: stri1, stri2`.
    `stud1: studentList first`
ACTION
    `stud1` GO
```

Figure 4.6: Types of Variables in a Rule

Assignment A SELF assignment expression where the left side is either a local or a fireable local variable present in the variable declaration part, and the right side any valid SELF expression.

Positive premise Any valid SELF expression that returns a boolean value.

Negative premise A premise with the following special syntax:

```
NOT | "proto" prot1 | `prot1 < 10`.
```

"NOT" is a keyword marking the beginning of a negative premise; then comes a variable declaration that obeys the same rules outlined in the previous section but whose scope is limited to the negative premise; finally, a SELF expression enclosed inside back quotes that must evaluate to a boolean value. Notice that the negation tested in this kind of premise is the negation by absence, not to be confused with the `not` message, and the absence is only checked at the moment of the test.

4.2.4.4 Action Part

The action part is the sequence of dot-separated actions following the "ACTIONS" keyword. Each action is any valid SELF expression between back quotes. A trailing dot after the last action is not allowed.

Optionally, a macro can be applied to an action, as was the case in the last action in figure 4.6. There are three types of macros, and all of them get automatically expanded by the parser:

GO This macro signals that the object in the action must be added to the evaluation context of the rule.

REMOVE The object in the action has to be removed from the evaluation context of the rule.

MODIFIED The object in the action has been modified, so it must be removed and added again to the evaluation context.

It is worth emphasizing that macros can only be applied to the first-order and fireable local variables present in the variable declaration part. The need to signal an object that it has been modified is known as "the problem of the modified" and is discussed in [Pac92]. It all boils down to the inability of an object of deciding if the modifications performed on it during the execution of a rule should or should not be permanent, therefore the responsibility of taking such a decision is delegated to the programmer of rules.

The syntax of the language has room for a couple of improvements. Both the problem with the naming of variables outlined above, and the necessity of surrounding SELF expressions with back quotes could be avoided if a syntactic Scanner (like Smalltalk's) and a structured grammar (see [Int] messages number 49, 813, 814) were available for the SELF language. For the moment, the implementation of those artifacts is proposed as a possible future work.

4.3 Consequences of Using Prototypes

Advantages

Of course, there were several advantages of using the prototype-based paradigm, most of them thanks to the inherent flexibility of PBLs. It was possible to define a very compact way of sharing behavior between prototypes, with a minimum of code duplication. For example, the network nodes hierarchy benefited from multiple inheritance and delegation, as shown in figure 4.7. As can be seen, the behavior of `nodeOneVarNegative` is defined in `traits node`, `traits nodeOneVar`, `traits nodeOneVarNegative` and `traits nodeNegativeCommon`; the "common" part is shared with `nodeNegative`, even though that prototype is in a separate branch of the inheritance hierarchy.

Several other parts benefited from a heavy refactoring and simplification of the code with respect to NéOpus (for instance the context management category in `traits ruleBase`), although these improvements are due to good software engineering practices and not to the fact that prototypes were used in place of classes.

One of the most interesting parts of FIRE in terms of profiting from SELF's flexibility and highly dynamic nature, is the implementation of the tokens. Neither the number of data slots in a `token` object nor the method slots are known in advance when a rule base is instantiated, that information depends on the rules subsequently added to the base. However, once a

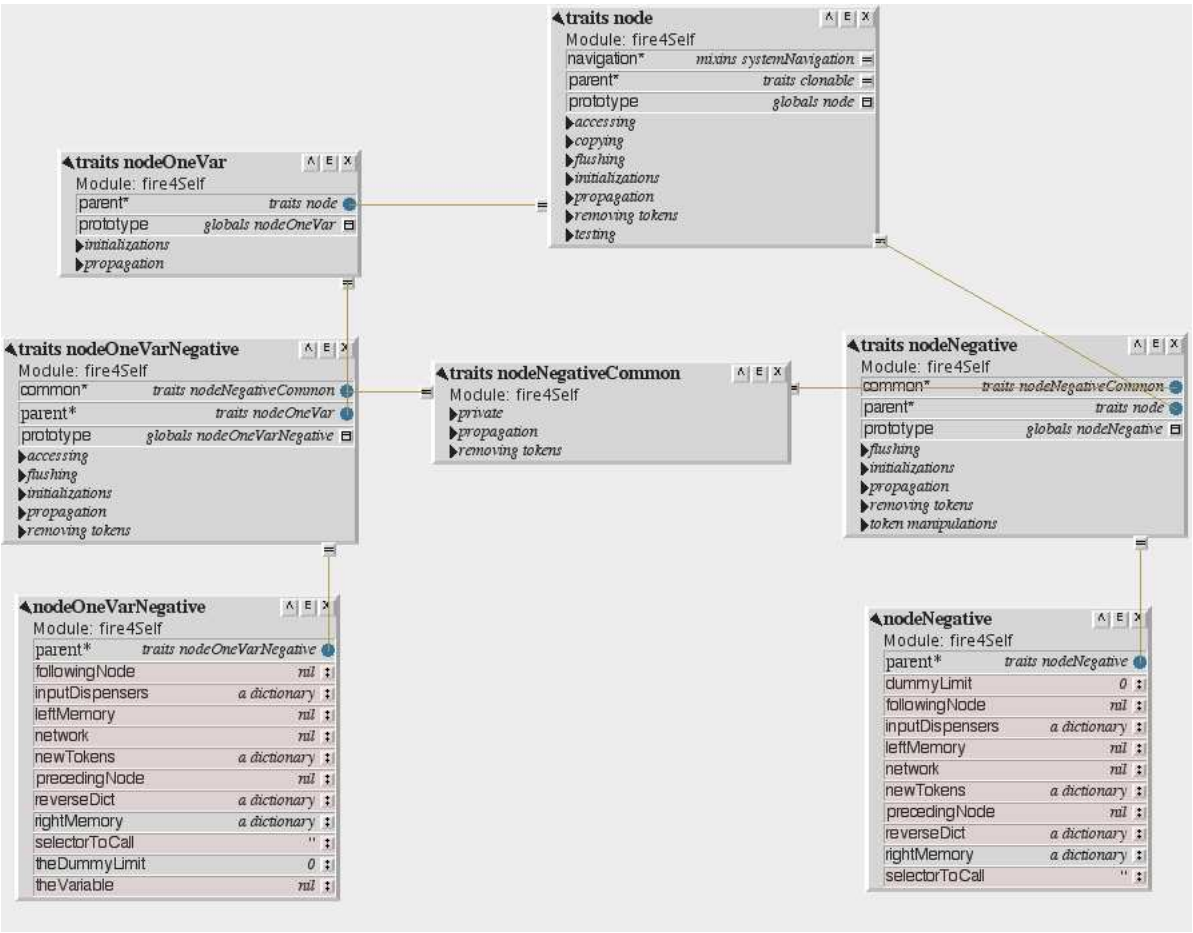

Figure 4.7: Delegation in Nodes Hierarchy

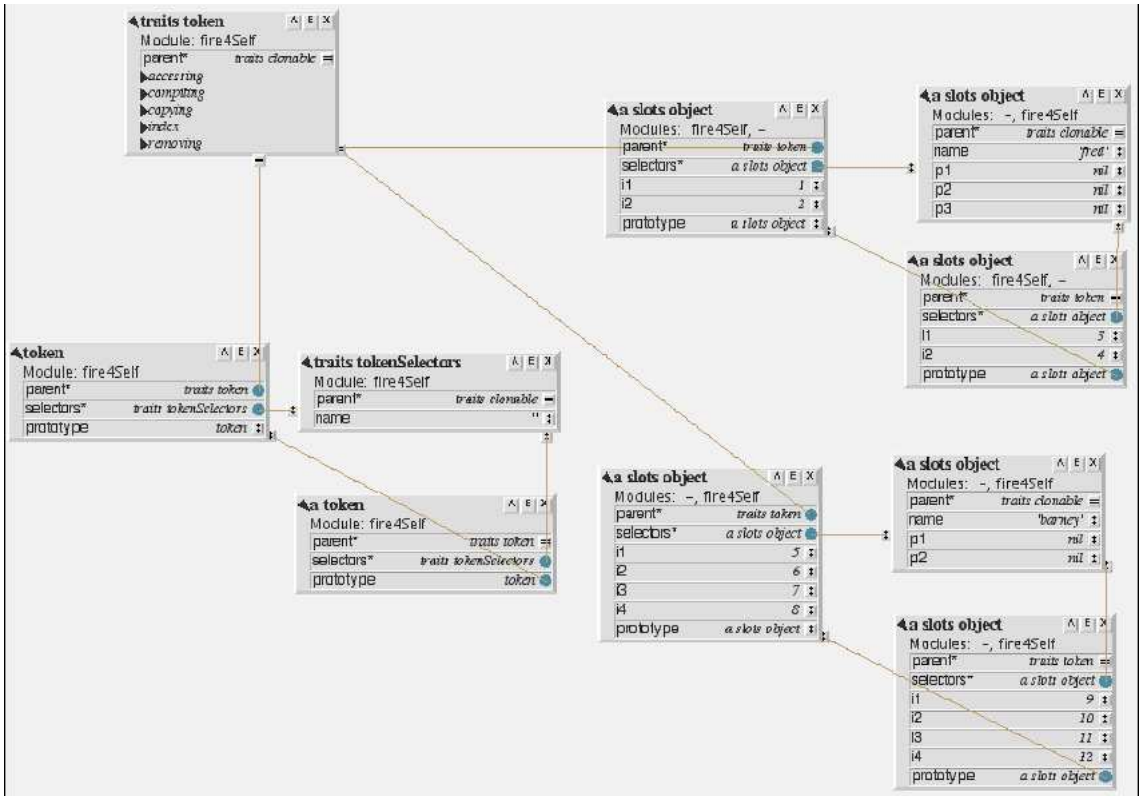

Figure 4.8: Meta-Prototypes in Tokens

token is created, all its successive copies should have the same method and data slots (albeit with different values in latter case). The solution for this problem was creating a Meta-Prototypical `token` object: a prototype whose copies are different prototypes with a varying number of data and method slots, but the copies of these copies are clones of their prototype. Figure 4.8 illustrates the situation; the left-hand side of the image depicts the Meta-Prototype, with zero data and method slots, alongside with one copy of it. The right-hand side shows two different prototypes, created by sending the message `newDynamicProto: aName` to the Meta-Prototype, each one with a different number of data and method slots, and each one with a copy of itself. Notice that the dynamic prototypes are modified when new rules are added, by adding new slots as required. Given that tokens are frequently propagated —and in great numbers— in the Rete network, it was mandatory that their memory footprint was as small as possible. The previous design accomplishes just that, the behavior is centralized in the `traits token` and `selectors` objects, from whence it is shared by all the tokens in the network, which are clones (i.e., shallow copies) of each other, only the idiosyncratic data in their $i_1, \ldots, i_n$ slots is different between tokens.

From a more practical point of view, the ability of SELF to add data or method slots dynamically greatly simplifies the generation of code. For example, to dynamically add a method in Smalltalk it is necessary to perform some string manipulations on the source code, compile the source and then add the new method to the method dictionary of a class, in SELF these operations are as simple as:

```
(reflect: selectors)
  at: selector
  PutContents: (body parseObjectBody)
```

Where `selectors` is the traits object where the method will be added, `selector` the name of the method and `body` its source code. Also, Mango does a great job simplifying the process of building a parser, thanks to its programming model where the user adds behavior to the nodes of a parse tree, which simplifies the implementation of transformations and manipulations of the parsed code.

Disadvantages

There were also disadvantages of using the prototype-based paradigm. Most of the prototypes were defined in a class-like way: the shallow `copy` message had to be overridden with messages to create copies with their data slots initialized to new values (as opposed to, slots pointing to the same objects as their prototype). The only exception was the `token` prototype, where making clones and not instance-like objects made sense. But the most challenging part was dealing with the lack of structural information in prototypes: in a class-based environment, it is generally possible to send the `class` message to an object and get the class object that originated it. In a prototype-based world, it is not always possible to get the dispenser of an object (in particular, not in SELF), and there are situations where it is necessary to propagate this information as an argument to message sends:

```
aMessageWith: anObject WithDispenser: objectDispenser
```

The converse of this situation also had to be dealt with: In a class-based world, a message can be sent to a class to find out its instances. In SELF, it was mandatory to develop efficient algorithms for retrieving either the copies of a prototype or the children of a traits object (depending of the kind of dispenser used); also to support the natural typing of variables, it was necessary to develop algorithms for traversing the ancestors and the descendants of a dispenser, and finding out their copies or children. These algorithms are outlined in the appendix B.

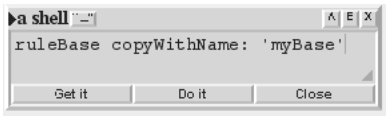

Figure 4.9: Creating a New Rule Base

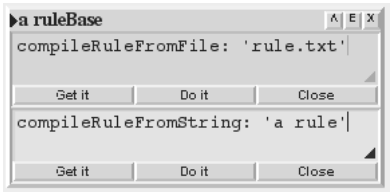

Figure 4.10: Adding Rules to a Rule Base

4.4 Using FIRE for SELF

This section explains the basic steps for adding and evaluating rules in FIRE, and introduces the concept of activation contexts and their management.

Adding Rules to the Rule Base

The first step for working with FIRE is creating a new rule base. This is accomplished by sending the message `copyWithName:` to the `ruleBase` prototype, as shown in figure 4.9.

After a new `ruleBase` object has been created, there are two possibilities (see figure 4.10) for adding new rules to it:

- Reading a text file with a single rule, by sending the message `compileRuleFromFile:` to the `ruleBase`. The path to the file is either an absolute path or a path relative to the `objects/` directory in SELF's working dir.
- Directly typing a rule into an evaluation box, by sending the message `compileRuleFromString:` to the `ruleBase`. In this case, all the single-quote characters that may be part of the text of the rule must be escaped by preceding them with a backslash, like this: `\'`

Context Management

The notion of an evaluation context was introduced by NéOpus and is also adopted in FIRE. Normally, the methods for grouping object in FIRE (see appendix B) would return all the objects in the system for a given type of dispenser. This is necessary for rules that apply globally to objects in all of

SELF's image, but has some inconveniences. First, for certain dispensers, the computation of the results will take some time to calculate due to the large number of objects belonging to the dispenser (for instance, `copiesOf: vector` returns around 40000 objects and takes close to one minute to calculate in a Mac G3). This can cause a big performance hit in the evaluation cycle. Second, not all of the objects obtained will be valid children or copies of a dispenser, some could be objects that are no longer referenced by any other object and are waiting to be garbage collected.

The evaluation contexts are presented as a solution to the above problems. The idea is simple: given that rules generally apply to certain objects member of a particular dispenser, then put them explicitly inside a container and tell FIRE that the rules should only be evaluated over the objects in the context. The context is just a `dictionary` object: each key is a string with the name of a dispenser, and each value is the list of objects belonging to that dispenser.

The user of FIRE can create its own context and pass it as an argument to the rule base (method `executeWithContext:`) or can use the rule base's own context. Several methods are provided in the "context management" category of the rule base (see figure 4.11).

The most commonly used methods include `addInContext:ToDispenser:` for adding a new object to the evaluation context;
`emptyContextForDispenser:` empties the list of objects for a given dispenser and `putAllInContextForDispenser:`, which causes the system to evaluate *all* the objects in the system belonging to the specified dispenser (this has the same effect as not using a context for that particular dispenser).

Firing the Rules

Several parameters must be considered when choosing an strategy for firing rules. FIRE provides a set of default values for all the parameters, which should suffice for most situations; however, if necessary it is possible to change them. Table 4.1 enumerates the parameters, their default values, the object to which messages should be sent, and the actual messages for modifying the values. Notice that the third column refers to the objects accessible from the rule base whose parameters need to be modified. For instance, for changing the insertion order of fireable rules in the conflict set, a user would type and execute the following command into an evaluation box of the `ruleBase` object:

```
conflictSet rules comparator: aNewComparatorObj
```

The parameters determine which actions get triggered by the forward-chainer, and it is possible that the results obtained depend on the order of firing of the rules (for instance, in the case of rules dealing with the ordering of objects).

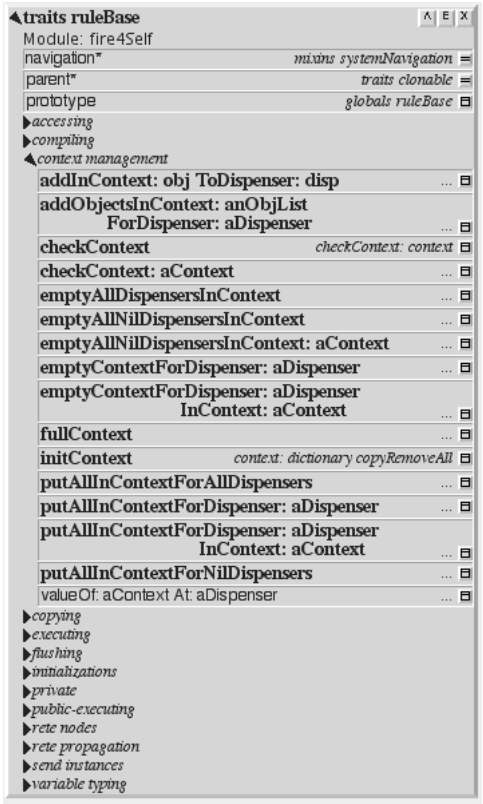

Figure 4.11: Context Management in FIRE

parameter	**default value**	**configurable from**	**messages to send**
activation context used	rule base's own activation context	`ruleBase`	`go` `executeWithContext:` `executeWithAllObjects` `executeWithSingleObj:` `WithDispenser:`
stop condition of evaluation process	conflict set is empty	`ruleBase`	`executeUntil:` `publicStopCondition:`
insertion order of fireable rules in conflict set	rules added at end of conflict set	`conflictSet rules`	`comparator:`
fireable rule triggered each evaluation cycle	first rule in conflict set	`conflictSet`	`trigger:` `triggerFirst` `triggerLast` `triggerDefault`
typing (simple or natural)	simple typing	`ruleBase`	`setNaturalTyping` `setSimpleTyping`

Table 4.1: Parameters for Firing Rules

A typical evaluation of a rule base starts by sending it the message `go` and ends when either the conflict set is empty or the argument passed as a parameter to the message `publicStopCondition:` evaluates to `false`. If the evaluation cycle needs to be halted, simply send `publicStopCondition: false` to the rule base.

Some remarks on the messages for configuring parameters: The evaluation process stops when either the public stop condition evaluates to `false` *or* the conflict set is empty, whichever happens first. Also, the typing of variables applies to `all` first-order variables in all the rules in the rule base.

4.5 Inference and Evaluation Cycle

This section summarizes how the different parts of FIRE interact to finally pick up and execute a rule, without delving deeply in implementation details.

In the `network` object it is stored a chain of `node` objects for each rule in the rule base, each one connected to the following and preceding node. The nodes are connected in the same order as the premises in each rule. Also, when one of the variables of the declaration part appears for the first time in a premise (such a variable it is said to be *free* up until this point), a dispenser is connected to that node: an object in charge of sending all the possible values that bound to that variable.

The evaluation cycle of the network is performed by sending new `token` objects on it, each one representing one possible set of instantiation of the variables declared in the rule. More specifically, the network takes care of putting in the tokens all the possible combinations of the values stored in the activation context of the rule base, that correspond to the type of dispensers declared for the rule. New tokens also get sent into the network when the execution of a rule causes a macro to be executed, therefore new objects get added, or existing ones get modified or removed.

The tokens are propagated down the network, going through the nodes corresponding to each one of the premises in the rule, until they reach the final node.

Each type of premise in the rule maps to one type of node in the network, although some types of nodes are just optimizations for representing more efficiently a type of premise (for instance, the `noVar` node is a simplification of a common `node`, for the cases where a premise does not need a dispenser attached to it because there are no free variables in the premise). FIRE is in charge of creating the right kind of node for each premise in a rule, and the specific implementation details are not important for understanding the propagation process. What really matters, is that nodes essentially fall down in two categories: they are either positive or negative (with assignment nodes being a special case of positive nodes).

The positive nodes perform the test corresponding to the premise they

represent, and then propagate the token if the *evaluation* of the premise succeeded. The negative nodes implement a more elaborate algorithm (explained in detail in [Cha90]) but with the same result: only if a token succeeds the premise, is propagated to the following node, where it will be stored in its left memory.

At the end of the sequence of nodes there is a `finalNode` object; if a token reaches this node means that it has succeeded all the tests of previous nodes, and is packed inside a fireable rule and added to the conflict set.

From that point, the rule that gets picked and fired depends entirely on the configuration parameters for firing rules, detailed in section 4.4.

The inference and evaluation cycle can proceed in two different ways, depending on how it was parameterized (see section 4.4). In the standard mode, the cycle continues as long as the conflict set is not empty, or the public stop condition is met. If acting as a monitor (see next chapter), the forward-chainer keeps testing objects in the system with respect to the rules. This is a polling approach towards the goal of constantly monitoring the state of the system.

Chapter 5

Applying a Forward-Chaining Rule-Based Engine for Meta-Programming

In a broad sense, *meta-programming* can be defined as the act of writing programs that write or manipulate other programs as their data. Certain programming languages allow writing programs that can examine and possibly modify their own high-level structure at runtime; such languages are said to be *reflective*. Furthermore, "*Declarative* Meta-Programming (DMP) is the use of a Declarative Programming language at Meta-level to reason about and manipulate programs built in some underlying base language" [DMP].

On the one hand, given that SELF is a reflective language, it is possible to perform meta-programming with it, but not in a declarative way. On the other hand, FIRE's rule language contains a declarative element: the ability to express patterns for grouping objects. This chapter explores the benefits of combining a reflective programming language with a rule language with declarative elements. Why FIRE is appropriate for performing meta-programming is motivated in section 5.1, next, a case study is presented in section 5.2 where FIRE takes on the role of an interactive tool for suggesting refactorings.

5.1 Why FIRE is Appropriate for Meta-Programming?

The SELF language has built-in meta-programming capabilities in the form of the very powerful reflective mechanism known as *mirrors* (see 2.4.1). FIRE adds another layer of meta-programming facilities thanks to its partially declarative capabilities and its rule language that is a subset of the

base language.

The language for expressing rules in FIRE acts as a partially declarative language over SELF's environment. It allows the expression of patterns, in the form of conditions, for grouping objects that conform to a certain criteria. Moreover, it is possible to constantly monitor the objects in the system, allowing *verification* of objects with respect to a description, although we still need to specify how to perform such a verification (this is the reason for stating that the rule language in FIRE is *partially* declarative). When an object is positively verified against a description in a rule, then a sequence of actions gets triggered by the forward-chainer.

Since the rule language is expressed in terms of SELF code (i.e., conditions and actions are "normal" SELF expressions) it is possible to use all of SELF's expressive power —in particular, its reflective capabilities— as part of the rule language itself. As a first consequence, the rules allow the expression of partially declarative meta-programming conditions for monitoring objects and performing actions over them. As a second consequence, it is possible to express rules that deal simultaneously with two kinds of grouping (traits-based and prototype-based), and either simple or natural typing can be used for interpreting inheritance relationships (see section 4.2.3). Both consequences make FIRE a good meta-programming candidate.

5.2 FIRE as a Tool for Guided Programming Based on Refactorings

This section demonstrates a case study in meta-programming with FIRE, explaining how it can be the starting point for a tool for guided programming that detects bad smells in code and suggests refactorings to correct them.

5.2.1 Refactorings as a Case Study

One of the most remarkable characteristics about FIRE is that it has unique grouping capabilities, and is capable of detecting commonalities in such groups.

Also FIRE can be used for performing forward-chaining inference in an interactive environment such as SELF's because it can take on the role of the interactive control loop, by monitoring the objects in the system.

Both characteristics are useful for detecting bad smells and suggesting refactorings, since certain types of bad smell detections involve looking for commonalities across groups of objects and meta-level operations for finding out their structure/contents, and FIRE's forward-chaining type of inference make it ideal for triggering a refactoring suggestion when an object(s) structure/contents changes in such a way that it corresponds to the description of a bad smell.

An introduction to refactoring and some related experiments are shown in the following sections.

5.2.2 Introduction to Refactoring

Refactoring, as defined in [Fow99], is "the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its external structure". It is a means for improving the design of the code after it has been written. The purpose of refactoring is taking a bad design and reworking it into well-designed code. To ensure that the code is still working after performing a refactoring, unit tests are executed. A unit test is a method for testing the correctness of a particular module of source code.

Bad Smells

A bad smell is a hint that indicates that something has gone wrong somewhere in the source code of a program. A smell can be used to track down the problem, and can indicate "when" a refactoring would be convenient and "what" refactoring should be applied. Martin Fowler [Fow99] defines 22 different bad smells that can be detected in code written in a class-based programming language. Each bad smell has an associated list of refactorings, which constitute possible solutions for getting rid of the smell. Fowler [Fow99] provides a comprehensive catalogue with more than 70 refactorings, each of them representing a concrete solution for a specific problem in the source code. The next sections mention the types of refactorings considered in the experiments. Two of the bad smells are relevant in the context of the current discussion, and they are explained in the next paragraphs. For more details about the refactorings mentioned, refer to [Fow99].

Duplicated Code Having the same code in different places is the most common smell found in applications. We can use the *Extract Method* refactoring to unify the code in one place. However, if the duplicated code is in two sibling subclasses, we can use *Extract Method* and *Pull Up Method*. In cases when the code in the extracted method has nothing to do with the class, the *Extract Class* refactoring would be appropriate.

Speculative Generality Sometimes programs get full of all sorts of hooks for special cases to handle things that aren't required. The result are methods which are only called by their own tests and thus only add to the complexity of maintenance and understanding. Replace unnecessary delegation with the code of the delegate with the *Inline Class* refactoring, *Collapse the Hierarchy* if there are abstract classes that are not doing much. Purge unused parameters with *Remove Parameter* or apply *Rename Method*.

5.2.3 Experiments

This section describes the experiments performed with FIRE to demonstrate its utility as both a meta-programming layer on top of SELF and a tool for performing guided programming in an interactive environment. Specifically, the experiments aim at the detection of bad smells in SELF code at the moment they appear, and at suggesting an appropriate refactoring. The following paragraphs gather some general considerations that are applicable to all experiments.

The kind of bad smells detected by the experiments are commonly found during the early stages of the development of a SELF program, when the prototypes are just being defined and the behavior of a prototype has not yet been completely factored out into a traits object.

The rules use a naïve approach towards detecting bad smells (like checking for methods with the same name ignoring their contents). The intention of the experiments is showing FIRE's meta-programming facilities as a way to do guided programming, and therefore the smell detection is simplistic, but could be enhanced in future works.

Given that the original refactorings suggested in Fowler's Book [Fow99] were conceived for class-based languages, they had to be adapted to a prototype-based paradigm. In particular, the extract to traits refactoring suggested in the third experiment does not have a direct equivalence in class-based languages, although it can be compared with the Extract Class refactoring.

The refactorings are suggested via a message printed on SELF's console. Again, this is a simplistic approach that could be enhanced by displaying a pop-up in SELF's graphical user interface.

Taking advantage of SELF's flexibility, a group of helper methods specific to the experiments was added to the rule base, so they can be easily invoked from inside the rules. These methods are shown next to the rule that uses them. All of them make use of SELF's reflective capabilities.

The experiments were performed over an object system like the one shown in figure 5.1. It includes a prototypical student object, three copies of it and a traits object for holding the common behavior.

As a final remark, the experiments do not take advantage of FIRE's evaluation context, but instead are evaluated over all the objects in the system that match the declaration part of the rule. This evaluation mode was selected, as it resembles more closely the control loop of an interactive tool for guided programming.

The following sections detail the experiments performed. Each one shows the rule that implements the smell detection, the helper method called from the rule and the results obtained.

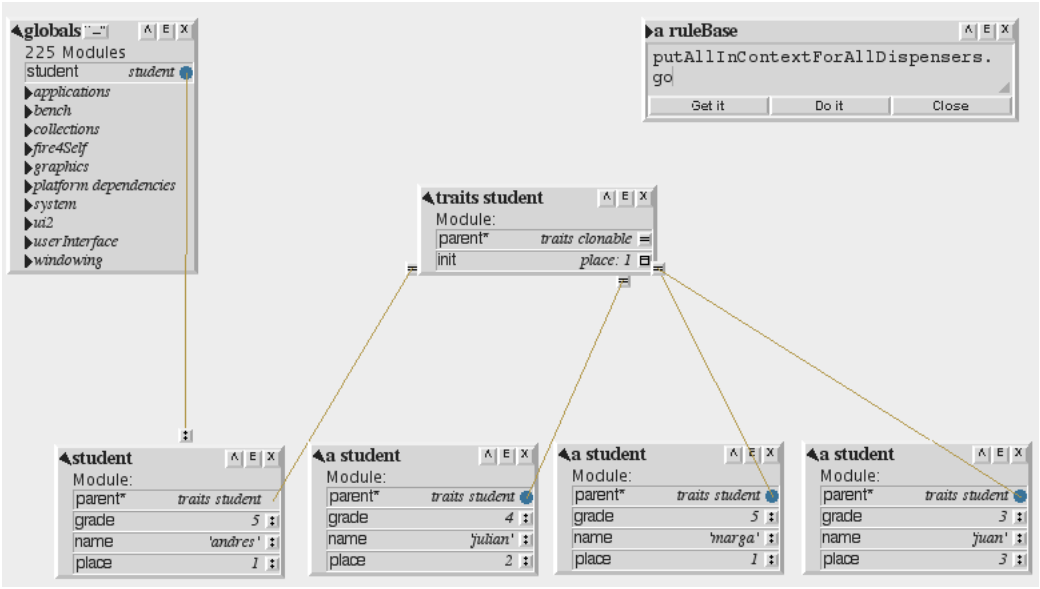

Figure 5.1: Experiments Setting

Pull-Up Method Refactoring

> *If two objects share a method of the same name, then suggest that the method should be moved to a common traits object*

The bad smell detected by the rule in this experiment is one type of duplicated code: if two objects share the same traits object, and a new method is added to one of them with the same name of a method already existing in the other, then a "duplicated methods" condition is detected and the suggested refactoring triggered by this situation is pulling-up the method to the traits object. Figure 5.2 shows the rule and figure 5.3 illustrates the helper method.

Results. The rule gets triggered whenever two student objects have methods with the same name. It starts printing messages in the console until one of the methods is removed or the method gets refactored into the traits object.

Remove Parameter Refactoring

> *If the arguments of a method are not used inside its body, then suggest that they should be removed.*

The bad smell detected by the rule in this experiment is one kind of speculative generality. FIRE checks the methods in an object, if a new method gets added and there are no references to one or more of its parameters

```
duplicatedMethods
  |
    "traits student" studentX studentY.
    "local" methodNames1 methodNames2 common
  |

  ‘studentX != studentY‘.
  ‘methodNames1: methodNamesIn: studentX‘.
  ‘methodNames2: methodNamesIn: studentY‘.
  ‘common: (methodNames1 intersect: methodNames2)‘.
  ‘common size > 0‘

ACTIONS

  ‘’duplicated method(s) found: ’ print‘.
  ‘common do: [|:e| (e, ’, ’) print]‘.
  ‘’consider pull up to traits object’ printLine‘
```

Figure 5.2: Rule for detecting duplicated methods

```
methodNamesIn: obj = (
  ((reflect: obj) asList
    select: [|:e| e isMethod])
      collect: [|:e| e name])
```

Figure 5.3: Helper method for detecting duplicated methods

inside the body of the method, then an "unused parameter" condition is detected and the suggested refactoring triggered by this situation is removing the parameter(s) that are not used in the method. Figure 5.4 shows the rule and figure 5.5 illustrates the helper method.

```
unusedParameters
  |
    "traits student" studentX.
    "local" numParameters1
  |

  ‘numParameters1: unusedArgsIn: studentX‘.
  ‘numParameters1 size > 0‘

ACTIONS

  ‘numParameters1 do: [|:e| e printLine]‘
```

Figure 5.4: Rule for detecting unused parameters

```
unusedArgsIn: obj = (
  | args. notUsed |
  notUsed: list copyRemoveAll.
  (reflect: obj) do: [ |:s|
    (s isMethod)
      ifTrue:
        [ args: s value arguments.
          args do: [ |:a|
            s contents source findSubstring: a
              StartingAt: 0
              IfPresent: []
              IfAbsent:  [ notUsed add:
                ('argument ', a, ' not used in method ', s key)]]]].
  notUsed)
```

Figure 5.5: Helper method for detecting unused parameters

Results. The rule gets triggered whenever one or more arguments in a method are not referenced in the method's body. It starts printing messages in the console until all the arguments in a method are used inside its body.

Notice that the lookup for a "reference" to an argument is just a substring search, so it is possible to find such a substring in the body

of the method, even though the argument was not referenced in the method's body.

Extract to Traits Refactoring

If a non-traits object has "too many" methods, then suggest that they should be moved into a traits object.

The bad smell and subsequent refactoring defined by the rule in this experiment are specific to traits-based PBLs. During the development of a prototype-based system, objects can start receiving some additional behavior in the form of new methods getting added to them. At a certain point, the methods are "too many" and should be extracted to a separate traits object, so they can be shared more efficiently between copies of the object. The rule written for detecting this bad smell checks to see if any object has more methods than a certain threshold for the number of methods (also defined in the rule). When the number of methods in an object is greater or equal than the threshold, then a "too many methods" condition is detected and the suggested refactoring triggered by this event is extracting them to a traits object. Figure 5.6 shows the rule and figure 5.7 illustrates the helper method.

Results. The rule gets triggered whenever the number of method slots in an object goes beyond a threshold value, which was defined with value "5" for this experiment. It starts printing messages in the console until the number of methods in the objects drops below the threshold, or the methods get refactored into the traits object

5.2.4 Discussion of Experiments

The experiments showed the straightforward nature of writing rules that apply to subsets of objects in the system and perform advanced reflective operations on them. In particular, FIRE showed its utility as a tool for guided programming that can detect bad smells and can suggest possible refactorings (pull-up method, remove parameter, extract to traits) in real time.

The experiments serve as proof of concept that FIRE effectively adds a layer of partially declarative meta-programming capabilities to SELF. Even more, FIRE has also proven its utility as a monitor in an interactive environment. More experiments are needed for exploring the expressive power of the rule language.

Certain extensions could be made on FIRE for improving its performance and/or adding more expressive power to the rule language; these are discussed in chapter 7: Future Work.

```
tooManyMethods
  |
    "traits student" studentX.
    "local" numMethods1 thresholdValue1
  |

  `numMethods1: numMethodsIn: studentX`.
  `thresholdValue1: 5`.
  `numMethods1 >= thresholdValue1`

ACTIONS

  `('object named ', studentX name, ' has more than ',
    thresholdValue1 asString, ' methods. Consider
    refactoring them to a traits object') printLine`
```

Figure 5.6: Rule for detecting an object with many methods

```
numMethodsIn: obj = (
  | num <- 0 |
  (reflect: obj) do: [ |:s|
    (s isMethod)
      ifTrue:
        [ num: num succ ]].
  num)
```

Figure 5.7: Helper method for detecting an object with many methods

Part III

Conclusions and Future Work

Chapter 6

Conclusions

This dissertation started by introducing the philosophical foundations of knowledge representation. Next, we discussed prototype-based languages, that are very expressive and are preferred to their class-based counterparts for representing knowledge. The language SELF was introduced as an example of a PBL.

Then, we introduced rule-based engines as a means for performing inference and extracting knowledge out of information. Hybrid systems were also introduced, which are the combination of object-oriented programming languages with rule-based systems. NéOpus was shown as a representative of hybrid systems. One particular type of inference engine was of special interest for this dissertation: the type that performs forward-chaining of rules, as it is better suited for monitoring a constantly changing fact base.

The main part of this research started with the introduction of FIRE, a tool built as a combination of a forward-chaining rule-based inference engine with the modern object-oriented, prototype-based language SELF. This combination provides a unique set of advantages, like the ability of representing the knowledge base via prototypes, the possibility of using SELF expressions as part of the rule language (in both condition and action parts), and the re-interpretation of the meaning of universal quantification of variables in terms of grouping and inheritance in a prototype-based world. Perhaps one of the most prominent advantages of building FIRE on top of SELF is that even reflective expressions can be used in the conditions and/or actions parts of rules. This, in combination with the declarative nature of grouping of objects in FIRE produces a meta-layer that might supersede SELF's mirrors mechanism.

As a proof of concept, we have applied FIRE as an interactive meta-programming tool for guided programming, performing experiments on detection of "bad code smells" and on suggesting possible refactorings to eliminate them. This is a very relevant research topic, as FIRE can detect bad smells at the moment they occur and suggest a course of action; the same

principle could be applied to detect other types of "anomalies" in the source code (e.g., non-conformance with coding standards) or even going beyond the source code and verifying conformance with documentation, models, etc. The next chapter points to other possible directions of future work.

Chapter 7

Future Work

This section analyzes some possible areas of future work in the context of this dissertation. First, possible implementation extensions for FIRE are presented, aiming at improving its performance, adding more expressive power to it and finding more uses to it. Second, future research directions for the combination of SELF and rules are discussed.

7.1 Improving FIRE

The algorithms for grouping objects are not considered very efficient. Currently, they are implemented at a higher level, but we believe that a more efficient way to iterate over groups of objects exists. Further study is required in the low-level constructs of SELF, such as the primitives mechanism, the maps of objects and the virtual machine.

Both the language and the rule parser have well-known shortcomings, like the need to back-quote SELF expressions and the variable naming problems. Both might be solved if a parser for SELF source code was available. Such an artifact could also be very useful for enhancing the ability of FIRE to express rules dealing with the structure of source code.

The rule language can be enriched with more advanced characteristics. Possible candidates are assertions, adding temporal reasoning, rule base inheritance and meta-rules for controlling the firing of other rules.

The monitoring aspect of FIRE can be greatly improved by using a mechanism to restrict monitoring to specific attributes of specific objects, reducing the number of times the forward-chainer is triggered. Such a mechanism is described in [DGJ04].

7.2 Future Research Directions

More important than the implementation improvements, it is necessary to perform more experiments with FIRE in the area of interactive program-

ming, going beyond the suggestion of refactorings. If FIRE is to be used for reasoning about code, then it is necessary to have some more advanced mechanism for manipulation of code at the object level.

It is also necessary to keep exploring the unique expressive power of the rule language, perhaps looking for ways to exploit the double hierarchy of traits and prototypes, and the cross-cutting nature of traits-based inheritance. Interestingly, the double hierarchy somewhat resembles the hierarchy of classes and interfaces in Java; it may be worth exploring this relationship, in the context of language-independent rules that take advantage of the structural similarities in the hierarchies of both languages.

Bibliography

[Age94] Ole Agesen. Mango: A parser generator for SELF. Technical report, Sun Microsystems Laboratories, 1994.

[AL87] R. Atkinson and J. Laursen. Opus: A Smalltalk production system. In *OOPSLA '87*, pages 377–387, 1987.

[BFKM85] L.. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*. Addison-Wesley, 1985.

[Bor86] Alan Borning. Classes versus prototypes in object-oriented languages. In *Proceedings of the ACM/IEEE Fall Joint Computer Conference*, pages 36–40, Dallas, Texas, November 1986.

[CDB99] Jacques Malenfant Christophe Dony and Daniel Bardou. *Classifying Prototype-Based Programming Languages*. Springer Verlag, 1999.

[Cha90] S. Charbonnel. Etude et réalisations de l'environnement du générateur de systèmes experts NéOpus. Master's thesis, Université de Nantes, Nantes, 1990. Rapport de stage de DESS au CEMAGREF.

[DGJ04] M. D'Hondt, K. Gybels, and V. Jonckers. Seamless integration of rule-based knowledge and object-oriented functionality with linguistic symbiosis. In *Proceedings of the 19th Annual ACM Symposium on Applied Computing, Special Track on Object- Oriented Programming, Languages and Systems*. ACM Press, 2004.

[DH04] Maja DHondt. *Hybrid Aspects for Integrating Rule-Based Knowledge and Object-Oriented Functionality*. PhD thesis, Vrije Universiteit Brussel, 2004.

[DMP] Declarative meta-programming. http://prog.vub.ac.be/research/DMP/.

[DUH88] Lynn Andrea David Ungar, Henry Lieberman and Stein Daniel Halbert. Panel: Treaty of Orlando revisited. In *OOPSLA*, pages 357–362, 1988.

[DUH91] Craig Chambers David Ungar and Urs Hözle. Organizing programs without classes. *LISP and Symbolic Computation: An International Journal*, 4(3):223 – 242, 1991.

[Fla94] P. Flach. *Simply Logical: Intelligent Reasoning by Example*. John Wiley, 1994.

[For82] Charles Forgy. Rete: A fast algorithm for the many Pattern/Many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[Fow99] Martin Fowler. *Refactoring - Improving the Design of Existing Code*. Addison Wesley, 1999.

[GNU] Free Software Foundation homepage. http://www.gnu.org/.

[GR85] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1985.

[Int] SELF interest group. http://groups.yahoo.com/group/self-interest/.

[Jac86] P. Jackson. *Introduction to Expert Systems*. Addison-Wesley, 1986.

[Lak87] George Lakoff. *Women, Fire and Dangerous Things*. University of Chicago Press, 1987.

[LHU87] L. A. Lieberman H., Stein and D Ungar. Of types and prototypes: The treaty of Orlando. In N Meyrowitz, editor, *Second ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, volume 23, pages 43–44, October 1987.

[Lie86] H Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proceedings of OOPSLA '86, Object-Oriented Programming Systems, Languages, and Applications*, pages 214–223. SIGPLAN Notices, 21(11)., November 1986.

[LSU88] H. Lieberman L.A. Stein and D. Ungar. A shared view of sharing: The treaty of Orlando. In W. Kim and F. Lockowsky, editors, *Object-Oriented Concepts, Applications, and Databases*, pages 31–48, 1988.

[MAN] Mango source code. http://www.cs.ucsb.edu/labs/oocsb/self/release/Self-4.0/.optional/SelfSources.tar.Z.

[NeO] NéOpus homepage. http://www-poleia.lip6.fr/ fdp/NeOpus.html.

[OA00] David Ungar et Al. Ole Agesen, Randall B. Smith. *The SELF 4.1 Programmer's Reference Manual*. Sun Microsystems, Inc. and Stanford University, 2000.

[Pac92] François Pachet. *Représentation de Connaissances Par Objets et Règles : Le Système NéOpus*. PhD thesis, Université Paris 6, September 1992.

[PP94] F. Pachet and J. Perrot. Rule firing with metarules. *Software Engineering and Knowledge Engineering . SEKE '94*, pages 322–329, June 1994.

[SAA+00] G. Schreiber, H. Akkermans, A. Anjewierden, R. de Hoog, N. Shadbolt, W. Van de Velde, and B. Wielinga. *Knowledge Engineering and Management: The CommonKADS Methodology*. The MIT Press, Cambridge, Massachusetts, 2000.

[Sel] SELF homepage. http://research.sun.com/research/self/.

[SOU] The SOUL logic meta programming tool. http://prog.vub.ac.be/research/DMP/soul/soul2.html.

[Tai99] Antero Taivalsaari. *Classes Vs. Prototypes Some Philosophical and Historical Observations*. Springer Verlag, 1999.

[Ung95] David Ungar. Annotating objects for transport to other worlds. In *Proceedings of the 1995 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '95)*, pages 73–87, October 1995.

[Ung02] David Ungar. *How to Program in SELF 4.1*. Sun Microsystems, Inc. and Stanford University, 2002.

[US87] David Ungar and Randall B. Smith. SELF: The power of simplicity. In *In Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages and Applications, October 1987*, pages 227–242. ACM SIGPLAN Notices, 22(12), December 1987.

Part IV

Appendix

Appendix A

Rule Language Grammar

```
Name:     'ruleParser'
Behavior: 'ruleParser.behavior.self'

Syntax:          SLR(1)
Transformations: 'elimEpsilons', 'elimSingletons' ;

<rule>            ::=  <name> <varDeclaration>
                       <premises> 'ACTIONS' <actions> ;

<name>            ::=  {identifier} ;

<varDeclaration>  ::=  '|' <declarations> '|' ;
<declarations>    ::+  <declaration> '.' ;
<declaration>     ::=  {dispenser}  <variables> ;
<variables>       ::+  {identifier} ;

<premises>        ::+  <premise> '.' ;
<premise>         ::|  <positivePremise> <negativePremise>  ;
<positivePremise> ::=  {selfexp} ;
<negativePremise> ::=  'NOT' <varDeclaration> <positivePremise> ;

<actions>         ::+  <action> '.' ;
<action>          ::|  {selfexp} {selfexp_go}
                       {selfexp_mod} {selfexp_rem} ;

Lex:             SLR(1)
Transformations: 'elimEpsilons', 'elimSingletons', 'useCharClasses' ;

{whitespace}    ->  [ \t\n\v\f\r\b]+ ;
{identifier}    ->  {lowercase}  (  {letter} | {digit} | '_' )* ;
```

```
{dispenser}     ->  '"' {whitespace}? {identifier}
                    ( {whitespace} {identifier} )* {whitespace}? '"' ;
{selfexp}       ->  '`' {whitespace}? {anychar}
                    ( {whitespace} {anychar} )* {whitespace}? '`' ;
{selfexp_go}    ->  {selfexp} {whitespace} 'GO';
{selfexp_mod}   ->  {selfexp} {whitespace} 'MODIFIED';
{selfexp_rem}   ->  {selfexp} {whitespace} 'REMOVE';
{lowercase}     =   [a-z] ;
{uppercase}     =   [A-Z] ;
{letter}        =   {lowercase} | {uppercase} ;
{digit}         =   [0-9] ;
{validchar}     =   '(' | ')' | '#' | '$' | '%' | '&' | '_' | '|'  |
                    '~' | ',' | '-' | '!' | '*' | '<' | '>' | '\'' |
                    '{' | '}' | '.' | '?' | '+' | '^' | '/' | '\\' |
                    '[' | ']' | '@' | '"' | '=' | ';' | ':' ;
{anychar}       =   ({letter} | {digit} | {validchar})+ ;
```

Appendix B

Mixin for System Navigation

FIRE defines different, novel ways of grouping objects in SELF, giving rise to new interpretations of the universal quantification of the variables in a rule. Since this is such an important topic in the current research, it is appropriate to explain in detail the mechanisms invented for grouping objects. In concrete, the `mixins systemNavigation` object was developed, which makes heavy use of SELF's mirrors facilities and provides a rich interface for traversing and grouping the objects in the SELF image. It was implemented as a mixin because its applicability goes beyond the scope of FIRE, and it should be easily integrated in other objects that may use it. Notice that it was implemented in a procedural style, with the receiver of a message being passed as an argument to it. Figure B.1 shows the methods defined in `mixins systemNavigation`, and subsequent sections enumerate the most important methods and outline the algorithms behind them.

Traits-Based Grouping

traitsOf: anObject Returns a list with all the parent slots of the object argument which are also traits objects.

childrenOfTraits: traits Returns a vector with all the children slots of the traits argument which are not traits objects themselves.

Prototype-Based Grouping

prototypeOf: anObject Returns the prototype of the given object. If the object is a traits or mixins object then the object itself is returned; if the object can respond to the `prototype` message then the method returns the result of invoking that message; finally, the method `prototypeIfPresent:IfAbsent:` is tried on the mirror of the object returning either the prototype or `nil` if none was found.

Figure B.1: Methods for Grouping and Traversing Objects in SELF

copiesOf: aProto Returns a vector with all the copies of the given object. Since in SELF there is not a direct way of finding out the copies of an object, this algorithm was developed: First, the list of all the parent slots of the object is obtained. Then, for each of the parents, a list is returned containing all their children which are *not* well-known and have the same prototype as the given object. This strategy excludes traits that might be children of one of the parents of the object and children copied from other prototypes that happen to share a parent with the object. Each successive list of children is intersected with the result of intersecting the previous lists; in the end, only the children common to all of the parents are returned.

Traits-Based Inheritance

allTraitsAncestorsOf: traits Returns a list with all the parent slots of the traits argument which are also traits objects. This operation is performed recursively up the traits inheritance hierarchy, until all the parent traits are found. The first element of the list is the traits object passed as argument.

allTraitsDescendantsOf: traits Returns a list with all the well-known children objects of the traits argument that are also traits objects. This operation is performed recursively down the traits inheritance hierarchy, until all the traits children are found. The first element of the list is the traits object passed as argument.

childrenOfTraitsIncludingAncestors: traits
Calls `allTraitsAncestorsOf:` and then calls `childrenOfTraits:` on every element of the returned list. All the resulting elements are returned in a new list.

childrenOfTraitsIncludingDescendants: traits
Calls `allTraitsDescendantsOf:` and then calls `childrenOfTraits:` on every element of the returned list. All the resulting elements are returned in a new list.

Prototype-Based Inheritance

allCopyDownAncestorsOf: aProto Returns a list with all the copy-down parents of the prototype argument. This operation is performed recursively up the copy-down inheritance hierarchy, until all the copy-down parents are found. The first element of the list is the prototype object passed as argument.

allCopyDownDescendantsOf: aProto Returns a list with all the well-known copy-down children of the prototype argument. This operation

is performed recursively down the copy-down inheritance hierarchy, until all the copied-down children are found. The first element of the list is the prototype object passed as argument.

copiesIncludingCopyDownAncestorsOf: aProto
Calls `allCopyDownAncestorsOf:` and then calls `copiesOf:` on every element of the returned list. All the resulting elements are returned in a new list.

copiesIncludingCopyDownDescendantsOf: aProto
Calls `allCopyDownDescendantsOf:` and then calls `copiesOf:` on every element of the returned list. All the resulting elements are returned in a new list.

Putting it all Together

The methods in this section are the more general ones, as they can receive either a prototype or a traits object, and they return the correct answer depending on the type of the argument.

str2Disp: str Receives as an argument a `string` with a path relative to the lobby and returns the object in that path. It is useful for passing a string instead of an object to all the methods in the mixin. This is very helpful in FIRE, as the dispensers are represented internally as strings, because traits object could not be used as keys in dictionaries.

allAncestorsOf: protoORtrait
Calls either `allCopyDownAncestorsOf:` or `allTraitsAncestorsOf:` depending on the type of the argument.

allDescendantsOf: protoORtrait
Calls either `allCopyDownDescendantsOf:` or `allTraitsDescendantsOf:` depending on the type of the argument.

copiesORchildrenOf: protoORtrait Calls either `copiesOf:` or `childrenOfTraits:` depending on the type of the argument.

copiesORchildrenIncludingAncestorsOf: protoORtrait
Calls either `copiesIncludingCopyDownAncestorsOf:` or `childrenOfTraitsIncludingAncestors:` depending on the type of the argument.

copiesORchildrenIncludingDescendantsOf: protoORtrait
Calls either `copiesIncludingCopyDownDescendantsOf:` or `childrenOfTraitsIncludingDescendants:` depending on the type of the argument.

Perhaps the most interesting method in the mixins object is `copiesOf:`. "Strictly speaking, there isn't supposed to be an observable relationship between an object and what it was cloned from in SELF". This was stated by David Ungar himself (see [Int], message number 1819). However, discovering all the copies becomes important in the context of quantifying the variables in a rule. The fact that this way of grouping objects is not common in SELF is evidenced by the lack of a mechanism in the language itself for finding out such information, making necessary the use of a high level algorithm. This has a disadvantage in terms of performance. During (informal) profiling of the algorithms, it was discovered that SELF's method calls `browse childrenOfReflectee:` and `browseWellKnown childrenOfReflectee:` take up most of the running time of the algorithms for finding children or copies, creating a potential bottleneck when dealing with a large number of objects (more than one thousand). A solution to reduce the running time would be to implement a more efficient version of those methods, possibly at the primitives level. This proposal is stated as a possible future work.

Another possible solution for finding all the copies of a prototype was suggested by Jecel Assumpcao Jr. (see [Int], message number 1820): basically, a brute-fore approach involving iterating over all the objects in the system and performing a comparison at the data slot level: if two objects share the same data slots, then they are copies of the same prototype. Some (informal) time measurements were made, discovering that for a small number of copies FIRE's implementation is much faster, but interestingly, for a large number of copies Jecel's algorithm can be a little faster. A more formal approach for measuring the efficiency of the algorithms needs to be developed, but is out of the scope of this document.

Appendix C

Licensing Information

The source code in FIRE for SELF's distribution is covered by two licenses. The source code specific to FIRE for SELF is under the General Public License (GPL) of the Free Software Foundation [GNU]. The GPL only covers the following files:

- `applications/collectionsAddOn.self`
- `applications/fire4Self.self`
- `fire4Self.grm`
- `install-fire4Self.self`
- `ruleParser.behavior.self`

The full text of the GPL is included with the code, and the license's notice can be accessed from within the program by typing `ruleBase license` in any evaluation box and pressing the "Get It" button. A transcript of the notice follows.

```
FIRE for SELF: A Forward-Chaining Inference Rule-Based Engine
               for a Prototype-Based Language
Copyright (C) 2004 Oscar Andres Lopez Paruma

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.
```

```
You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
```

Mango is an essential part of FIRE for SELF and is distributed together with it, however it is covered by one of Sun Microsystems' licenses (also included with the code). This is Mango's license:

```
# Sun-$Revision: 30.4 $

Copyright  1993-2004 Sun Microsystems, Inc. and Stanford University.

All Rights Reserved. RESTRICTED RIGHTS LEGEND: Use, duplication, or
disclosure by the government is subject to restrictions as set forth
in subparagraph (c) (1) (ii) of the Rights in Technical Data and
Computer Software Clause at DFARS 252.227-7013 (Oct. 1988) and FAR
52.227-19(c) (June 1987).

Sun Microsystems, Inc.  2600 Casey Avenue, Mountain View, CA 94043
USA

LICENSE:

You may use the software internally, modify it, make copies and
distribute the software to third parties, provided each copy of the
software you make contains both the copyright notice set forth above
and the disclaimer below.

DISCLAIMER:

Sun Microsystems, Inc. makes no representations about the
suitability of this software for any purpose. It is provided to you
"AS IS", without express or implied warranties of any kind. Sun
Microsystems, Inc. disclaims all implied warranties of
merchantability, fitness for a particular purpose and
non-infringement of third party rights. Sun Microsystems, Inc.'s
liability for claims relating to the software shall be limited to
the amount, if any of the fees paid by you for the software.  In no
event will Sun Microsystems, Inc. be liable for any special,
indirect, incidental, consequential or punitive damages in
connection with or arising out of this license (including loss of
profits, use, data, or other economic advantage), however it arises,
whether for breach of warranty or in tort, even if Sun Microsystems,
Inc. has been advised of the possibility of such damage.
```

Appendix D

Installation Instructions

Installing FIRE for SELF is very simple once you have a copy of the source code distribution. It can be summarized in just four steps.

1. Copy the file `fire4Self.tar.bz2` to `$SELF_WORKING_DIR/objects`
2. Go to `$SELF_WORKING_DIR/objects` and execute the command

   ```
   tar -xjf fire4Self.tar.bz2
   ```

3. Start SELF
4. At SELF's command line prompt, type and execute the following commands

   ```
   'install-fire4Self.self' _RunScript
   ruleBase bootstrapParser
   ```

You can check that the `globals`, `traits` and `mixins` name spaces now have a new category, "fire4Self".