The Pennsylvania State University

The Graduate School

Department of Computer Science

OBJECT PROGRAMMING,

LINEAR LOGIC

AND

JAVA

A Thesis in

Computer Science

by

Alexandre A. Betis

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

December 1999

We approve the thesis of Alexandre A. Betis.

Date of Signature

Dale A. Miller
Professor of Computer Science and Engineering
Thesis Adviser
Head of the Department of Computer Science and Engineering

Catuscia Palamidessi
Professor of Computer Science and Engineering

John J. Hannan
Associate Professor of Computer Science and Engineering

Abstract

In this thesis I study the relationship between Linear Logic and Object-Oriented programming languages. I first start by building a small language named Popeye that allow the writing of simple object-oriented programs. Next, I prove that any Popeye program is equivalent to a collection of Linear Logic formulas. As a consequence, I provide an interpreter for Popeye written in λProlog. I then show that certain Popeye programs can be automatically translated to Java. I study the issues involved in such a translation and give the rules needed to guarantee meaningful Java code. In this formalism, I then build an object architecture that depicts logical gates and prove its correctness. The Java code generated is then guaranted to be correct. Some additional examples are also studied.

Table of Contents

List of Figures

Acknowledgments

I first would like to thank Celine, for letting me embark on this adventure and for her constant support even though an ocean separated us. I could not have made it without her.

I thank my family for having, once more, helped in the realization of a bold project through every possible means.

I would like to express my most enthusiast recognition to the support provided by Dale Miller, my advisor, and his wife, Catuscia. The highest standards they set forth, their patience and their tolerance to my impossible character made this work possible.

Finally, I'd like to extend my greetings to the various people I met in State College during my American life. They are too numerous to list here, but I consider every single one of them as a vital foundation for this work.

“Utopia, by Sir Thomas Moore...

(Flips pages temptatively)

I never could finish that book... ”

Hugo Pratt

Corto Maltese in Siberia

Chapter 1

Introduction

Any kind of work reflects its creators' beliefs and thought system. Like any monument or writing left by our ancestors, a program reflects many aspects of one's personality. Field archaeology also taught us that the tools used by our ancestors were very symptomatic of the people using them. If our initial analogy still holds, then it implies that our programming frameworks should also reflect this plurality. It is surprising, then, to notice how this is definitely not the case. The vast majority of modern APIs are defined in a context where a given program strictly enforces the succession of instructions to execute: an imperative context. Other approaches, using functional or logic paradigms are vastly underused.

In this thesis, I show how logic-based programming languages can be successfully used in writing object-based programs. To do so, I designed a small macro language named Popeye whose every construct can be tied to a formula expressed in Linear Logic as introduced by Girard in [7]. The initial point is then made stronger by proving that every object written in this environment can also be safely and automatically translated into Java. Popeye can thus be seen as a touchstone between two very different worlds. When standing on this stone, one can proceed in either direction (namely, Linear Logic or Java).

When moving to Linear Logic, the program becomes a set of strict mathematical formulas that model the behavior of the program. It is then possible to build rigorous proofs about the correctness of the initial program. In [4], Chiramar built such proofs when specifying DLX, a RISC-like architecture, though the logic used, FORUM [14], is a presentation of Linear Logic. One might also be interested into using this logical representation into a Prolog-like interpreter. This is why I provide one. In many ways, it can be closely linked to Lolli [8], a programming language that deals with Linear Logic in a direct way.

On the other hand, when moving to Java, and given that the initial program passes some checkpoints, we get a compilable code. Fig 1.1 presents a global overview of the system.

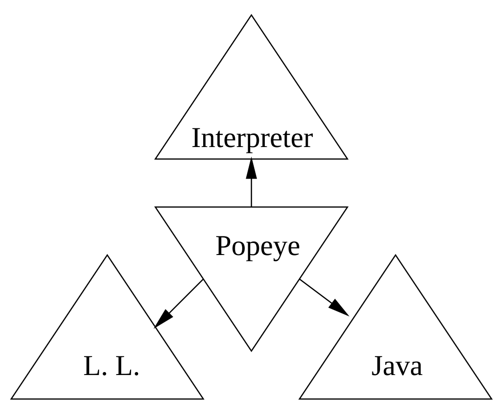

Fig. 1.1. An overview of Popeye and its neighbors

There are very good reasons for not having built such a tool before. In fact, a first attempt at building objects using logic was outlined in [9]. The authors used λProlog in order to build representations of simple objects. At the time, the authors were stuck because their logic could not express the idea of state, that is, some element with a value changing over time. Prolog-like interpreters are based on a logic very close to that of classical mathematical proofs, where once a fact has been proved true, it can be reused in any other subsequent proof. As a consequence, a switch could be both on and off. Linear Logic solved the problem. In Linear Logic, a linear fact can only be used in one subsequent proof. This allows us to re-write states in the middle of a proof. A valid switch object can then be expressed in Linear Logic. I will show how Popeye can express such a thing, and how it can be tied to Logic and Java. I also provide a bigger example where an object architecture of logical gates is built (among other things). The coming of Java is also an important issue. Its all-object approach brought a nice clean framework for our translation. The other reason we chose Java is related to its built-in support for threads and concurrency. As outlined in our final section, it is our hope that Popeye will prove very useful in that matter.

On a bigger scale, the Popeye approach could generate a new way of programming. In [18], Schach explains how modules can be split into two groups.

- Functional modules
- Logical modules

Functional modules are related to the realization of one precise action. For example **disk_IO()** is a typical functional module. They are usually easily implemented and debugged. On the other hand, logical modules are in control of the execution of the program. **CPU_playchess()** is a good example for this category. These modules use logical conditions in order to decide where the program should proceed next. There are two main issues about these modules. First, they are very expensive to test. The process usually involves the drawing of careful scenarios of execution. Each of them validate some point in the specification and check for errors. Second, such modules are usually located in the upper section of the program hierarchy. As a consequence, any error that slips through the previous examination can be assumed to be a design fault, hence a very costly one. Consider now a Popeye framework. Here, I can write down my logical modules and then feed them into the interpreter. Thanks to backtracking, I can explore very quickly a vast amount of scenarios, and thanks to Linear Logic, I might be able to build a mathematical proof that my specification is actually met. If so, then I can generate the Java code and be confident in its value. This is a surprising approach, yet its consequences could be enormous.

The major issue about building such a system is that one needs to think about two different things when introducing any new construct. Just like two intersecting circles generally define a rather small area when compared to the total area of the original circles, it has the consequence of making Popeye a very restrictive frame to work in. Especially now, since the code I have written has no debugging features whatsoever. On the other hand, the payoff and the elegant approach are, in my opinion, sufficient reasons for pushing in that direction. After all, everybody scorned objects for being too heavy a burden when the concept was first introduced. In fact, I would like to redirect all such individuals to a Theorem 1.1 my first real teacher in mathematics once claimed I demonstrated:

Theorem 1.1 (Shadok). Why make things simple when they can be made complicated? [1]

This thesis consists of six parts. In the first one, I describe the Popeye language by itself, as well as the intended meaning of the keywords introduced. In the second part, I give the logical background necessary to the translation of Popeye and prove that any program in Popeye can be translated into Linear Logic. In the third part, I express the main "cultural" problem that arises when translating Popeye into Java. I identify it with a semantic conflict between tests and actual instructions on the logical side. A typechecker-inferer is then built to enforce a property that will prevent this confusion. The fourth chapter show how easy the translation to Java is after that by actually giving the necessary rules. The fifth chapter rebuilds the example brought forth in [9] in Popeye. In the final section, I expose my views on how this research could be extended and where, in the end, it might lead us.

[1] The Shadoks were first introduced in a French comic book during the seventies. It depicted the laborious life of strange animals whose goal in life was to evade an impossible planet whose shape kept changing at random. The twisted fate of their world had a serious impact on the mind of the Shadoks, who soon developed an absurd theory of the universe. In their idea, for example, a one in a thousand chance of success implied success... as long as you were careful enough to methodically fail the nine hundred and ninety-nine first attempts. Very popular among French scholars, they incarnate what the universe would look like if some of its fundamental laws were ignored.

Chapter 2

Popeye

In this chapter, I introduce Popeye, a small programming language which I can use to write simple objects. Among the various things such a language could implement, I have selected a rather limited subset of things. This set includes:

- Class hierarchy and inheritance
- Overloading

Features like operator overloading or exceptions are not supported. Exceptions, in particular, require a logic with more expressive power, as shown in [4]. We will talk about this again in the final chapter. More imperative features are also limited to a reasonable size, namely:

- Built-in support for integers and booleans
- If-then-else statements
- Recursion

Overall, I introduced what I felt was a minimum for writing decently interesting code. The absence of loops is, in itself, not limiting and will be discussed in the final chapter.

MODULE ::= TYPEDECL
CLASSDECL
SELFDECL
STATICDECL*
PRIVATEDECL*
PROGDECL

CLASSDECL ::= **class** NAME .

STATICDECL ::= **static** NAME NAME CONST .

PRIVATEDECL ::= **private** NAME NAME .

SELFDECL ::= **self** NAME .

PROGDECL ::= CONSDECL METHDECL*

CONSDECL ::= **cons** NAME (NAME VAR*) $\succ\!\!-$ ICMD .

METHDECL ::= ATOM $\succ\!\!-$ CMD .

ICMD ::= CMD
| ICMD >> ATOM
| **init** NAME VAR
| **super** NAME (NAME VAR*)

CMD ::= ATOM
| CMD >> ATOM
| ATOM +> CMD |> CMD
| **new** (NAME VAR*) NAME \ CMD

ATOM ::= NAME VAR VAR*
| **rw** NAME VAR VAR
| ATOM == ATOM
| **empty**
| **cast** NAME TYPE

VAR ::= NAME
| **pbool** BOOL
| **pint** INT
| **pcst** NAME NAME

CONST ::= BOOL
| INT

TYPEDECL ::= NAME **extends** NAME.
| **otype** NAME $\{$ TYPE $\rightarrow$ $\}^+$ (class / aclass) .
| **ctype** NAME NAME TYPE .
| **ptype** NAME NAME TYPE .
| **mtype** NAME (in/out) $\{$ TYPE $\rightarrow$ $\}^+$ (cmd / acmd) .

Fig. 2.1. A grammar for Popeye

2.1 Context-free Grammar

Fig 2.1 shows a possible grammar for Popeye. NAME is a token representing character strings. VAR represents either NAMEs or numerical constants handled by Popeye (like, say, integers). I now move on to describing the operational semantics of the reserved words introduced. At some point, I will depict a construct as a "possible command", which could seem rather strange. The vagueness of this sentence will become clear in Chapter 4.

2.2 Reserved words

class

Syntax: **class** [*name*]

Role: At the beginning of a module, specifies the name of the object.

static

Syntax: **static** [*objname*] [*name*] [*value*]

Role: Declares a constant related to the object.

private

Syntax: **private** [*objname*] [*name*]

Role: Declares a private element of the object.

self

Syntax: **self** [*name*]

Role: Assigns a name to use as a reference to the object when defining its methods.

cons

Syntax: **cons** [*objname*] >– [*commands*]

Role: Defines the commands [*commands*] to execute when constructing the object referenced by [*objname*]. There must be one and only one declaration of this type in every program.

>–

Syntax: [*meth_name*] [*objname*] [a_1]...[a_n] >– [*commands*]

Role: Defines the [*commands*] to execute when calling method [*meth_name*] of the object referenced by [*objname*], with parameters [A_i] ($i \in N$)

>>

Syntax: [*command*] >> [*more commands*]

Role: Specifies a succession of commands. First [*command*] is to be executed, then [*more commands*] is considered

+> *etc.* |>

Syntax: [*condition*] +> [$commands_1$] |> [$commands_2$]

Role: Specifies a typical branching instruction.

If [*condition*] is true, then [$commands_1$] are executed,

otherwise, [$commands_2$] are executed.

rw

Syntax: **rw** [*private_variable*] [*old_value*] [*new_value*]

Role: A possible command, accesses a private variable named [*private_variable*], reads its old value in a variable named [*old_value*], then replaces the value of the private variable with the content of [*new_value*]

init

Syntax: **init** [*private_variable*] [*value*]

Role: This command is compulsory for all private variables in the object constructor. It assigns an initial value to them.

cast

Syntax: **cast** [*variable*] [*type*]

Role: Casts a variable to a given type. Used at type-checking time.

new

Syntax: **new** $[name]$ $[obj\_name]\backslash$

Role: Creates a new object of type $[name]$ with

the reference name $[obj\_name]$ to be used afterwards.

super

Syntax: **super** $[name]$ $[obj\_name]$

Role: This command is found only in constructors. It specifies that

the current object is an extension of $[obj\_name]$.

==

Syntax: $[ref\ or\ val_1]$ == $[ref\ or\ val_2]$

Role: A possible command. In a test, will test for the equality of the two

parameters. As a command, it will assign the value of $[ref\_or\_val2]$

to $[ref\_or\_val1]$.

2.3 Statements

In Popeye the classical object notation is rewritten without prefixing the object name: the object name is placed after the method name. Also, everything is written in a curried form. This gives the following equivalence:

$$\forall n \in N : \mathbf{o.method(a_1, a_2\ ...\ a_n)} \equiv \mathbf{method\ o\ a_1\ a_2\ ...\ a_n}$$

Here are a few examples:

$$\mathbf{switch.toggle()} \;\equiv\; \mathbf{toggle\ switch}$$

$$\mathbf{lemonade.mix(sug, ice, lem)} \;\equiv\; \mathbf{mix\ lemonade\ sug\ ice\ lem}$$

The other key aspect of the language is how it specifies methods returning values. It is pretty clear that the examples given above are non-returning functions. In Java, they would be respectively declared as follows:

$$\mathbf{public\ void\ toggle()}\{\ ...\ \}$$

$$\mathbf{public\ void\ mix(\ int\ sug,\ int\ ice,\ int\ lem)}\{\ ...\ \}$$

That is, with a **void** return type. Non-**void** methods are basically written the same way, except that the programmer should always assume that the last parameter of the Popeye expression is the returned variable. In that case, the equivalence expressed above does not hold anymore. It is the replaced by the following:

$$\forall n \in N: \ \mathbf{a_n} \;=\; \mathbf{o.method(a_1, a_2\ ...\ a_{n-1})} \equiv \mathbf{method\ o\ a_1\ a_2\ ...\ a_{n-1}\ a_n}$$

Here are additional examples:

$$\mathbf{s = switch.get()} \;\equiv\; \mathbf{toggle\ switch\ s}$$

$$\mathbf{c = lemonade.calories(sug)} \;\equiv\; \mathbf{calories\ lemonade\ sug\ c}$$

All methods fall into either one of these categories. I will now formalize this characterization.

Definition 1 (Categories of methods). A method is considered an...

... in-method when it does not return any value.

... out-method when it returns a value. ∎

From now on, I will characterize a method as either in or out.

2.4 Expressions

These statements constitute commands that can be sequenced by using the $>>$ operator. For example, if **ping b** is a command and **pong b** another one, then

$$\textbf{ping b} \;>>\; \textbf{pong b}$$

is a valid command. Some of the reserved words expressed above are also commands. **rw** is the only allowed way for the Popeye programmer to access private data. The **new** is used to create new objects when needed. The == is used to assign a value to a variable. Finally, **empty** is an empty instruction that can be seen as an **idle** command in assembly languages. Here are a few examples:

rw state old false	≡	Read the current state in **old**.
		write the boolean false as its new value
new switch s\ **toggle s**	≡	Create a new switch with name **s**,
		then toggle **s**

In an object declaration, methods are defined by using the $\succ\!\!-$ operator. To be more explicit, **meth** $\succ\!\!-$ **code** is tantamount to saying that **code** is what should be done when **meth** is called. For example, in a switch object, one might want to specify a **get** method that returns the current state of the switch. If we assume that the chosen reference name is **sw**, then here is how its declaration should look like:

get sw s $\succ\!\!-$ **rw state s s**

Notice how the **rw** statement is used to access the state and give a return value. This very common technique will be widely used through the remaining chapters.

All other keywords introduced are then more or less declaration elements that must be placed before everything, in the following order: **class**, **self** and **private**. I am now ready to completely write down a switch object. Some typing instructions should be added, but it is better for clarity to do so later.

2.5 First object: a simple switch

The switch can be loosely specified as an object with one private boolean variable representing its internal state. It has three methods. **get** is used to access the current state of the object. **set** rewrites this state with a new value. **toggle**, finally, performs the switching operation. That is, if the state was **true** then the new state is **false** and vice-versa. Fig 2.5 gives the corresponding program. There are various interesting things to see in this program. First, the **self** keyword can seem somewhat useless since it is quite obvious that all methods here are defined with regards to the switch object.

```
class switch.

self sw.

private switch state.

 cons switch sw >- init state (pbool false).

 get sw s >- rw state s s.

 set state sw s >- rw state o s.

 toggle sw >- rw state (pbool true) (pbool true) +> set sw (pbool false)
                                                  |> set sw (pbool true).
```

Fig. 2.2. Switch object in Popeye

The goal of such an addition is to make the use of the upcoming interpreter easier. By enforcing the fact that the **self** name should appear in the code, I make sure that the programmer is aware of the meaning and peculiar syntax of my language. The other striking thing about this code is the repetitive use of **rw** for accessing the private variable. An important point is that I use **rw** both as a command (in **get** and **set**) and as a test (in **toggle**). This, of course, is not possible in Java. We now move on to describing the typing commands.

2.6 Typing

Popeye is a loosely typed language, which means that it does not require the typing of all the identifiers introduced. The only elements that need to be typed are the methods, the private variables and the constants. As far as the objects are concerned, a special declaration is used to specify their necessary construction parameters. I thus need four instructions. I picked **otype**, **ctype**, **ptype** and **mtype**, respectively for objects, constants, private data and methods. Popeye uses λProlog-style typing declarations, which are themselves inspired by Church's Simple Theory of Types ([5]). The **mtype** directive has a special additional parameter that is used to specify whether the method returns a parameter or not. An additional directive is added to specify extension in an object architecture.

Typing keywords

Syntax:	**extends** [*objname*]
Role:	Declares that the current object extends [*objname*]
Syntax:	**otype** [*name*] [*type*]
Role:	Declares an object and the type of the parameters needed to create it. [*type*] must be concluded by a $\rightarrow$ class (or $\rightarrow$ aclass if abstract)
Syntax:	**ctype** [*objname*] [*name*] [*unary type*]
Role:	Declares a constant with its type.
Syntax:	**ptype** [*objname*] [*name*] [*unary type*]
Role:	Declares a private data variable with its type.

Syntax: **mtype** (**in/out**)[*name*] [*type*]

Role: Declares a method and the type of the parameters needed to call it.

[*type*] must be concluded by a $\rightarrow$ cmd (or $\rightarrow$ acmd if abstract).

If **in** is chosen, then the method will not return a value.

If **out** is picked instead, then the method will return a value whose type will be the one specified right before the $\rightarrow$ cmd.

Constants and base types

Popeye can deal with a very small set of base types, namely integers, booleans and references to some other object. Fig 2.3 lists them. Numerical constants are specified through the use of special keywords **pbool** and **pint** respectively. Constants defined in

ref [*object*]	:	References to objects.
boolean	:	Booleans.
int	:	Integers.

Fig. 2.3. Base types in Popeye

objects can be called upon through the **pcst** keyword. Fig 2.4 gives an exact syntax. To illustrate all this, Fig 2.5 shows the completed switch object.

pint [*integer*]	*e.g.* :	**pint** 5
pbool (*true* / *false*)	*e.g.* :	**pbool** *true*
pcst [*obj_name*] [*cst_name*]	*e.g.* :	**pcst** *switch on*

Fig. 2.4. Constants in Popeye

```
otype switch class

ptype switch state boolean

mtype get      out   ref switch -> boolean -> cmd
mtype set      in    ref switch -> boolean -> cmd
mtype toggle   in    ref switch -> cmd.

(.../...)
```

Fig. 2.5. Typing header for the switch object

At this point, I have described the Popeye programming language. Additional examples are deferred to Appendix 8.1. I will now proceed to outline the ties between this language and Linear Logic.

Chapter 3

Popeye in Linear Logic

In this chapter, I will build functions that will take a Popeye program and translate it into a Linear Logic formula. Linear Logic, first introduced by Girard in [7], is a superset of classical logic that introduces a new kind of implication, referred to as linear implication. It is noted $\multimap$. In this thesis, I will focus on a smaller and complete set of this logic, also known as the Lolli presentation. Fig 3.1 gives a grammar for this presentation, as expressed in [8]. It is important to understand that we might not need all these connectives. A good way of introducing these symbols is probably to start with

$$R := \top \mid A \mid R_1 \mathbin{\&} R_2 \mid G \multimap R \mid G \Rightarrow R \mid \forall x.R$$
$$G := \top \mid A \mid G_1 \mathbin{\&} G_2 \mid R \multimap G \mid R \Rightarrow G \mid \forall x.G \mid G_1 \oplus G_2 \mid 1 \mid G_1 \otimes G_2 \mid \,!G \mid \exists x.G$$

A are atoms.

Fig. 3.1. Grammar for Linear Logic formulas

the &. $A \mathbin{\&} B$ means that in regards to the resources available, A can be proved and B can be proved. From this, I can partially define the two other connectives I use through the two following equivalences:

$$B \otimes C \multimap A \quad \equiv \quad C \multimap (B \multimap A)$$

$$A \oplus B \multimap C \quad \equiv \quad (A \multimap C) \mathbin{\&} (B \multimap C)$$

The key idea behind Linear Logic is the possibility of context management. In short, a fact can be true or false depending on where we are located in the proof for a given formula. In more classical logics, proving $A \supset B$ from a given context Γ is done by first augmenting Γ with A and attempt the proof for B using the new context. In terms of sequent calculus, to prove $\Gamma \longrightarrow A \supset B$, attempt instead to prove $\Gamma, A \longrightarrow B$. Intuitionistic contexts are very useful in many domains, but they have the major limitation of being ever-growing. It is never possible for such contexts to be smaller at the top of the proof tree than at the root. In [10] Hodas details how this limitation can be a major obstacle in many aspects. One of them happens to be the key characteristic of object programming, that is, the encapsulation of states. To address this problem, it is necessary to have a finer management of the context. Linear Logic makes this management clear and natural. The context is split between unbound formulas (Γ) and bound formulas (Δ). Bound formulas can only be used for one proof, after which they are lost for following proofs. Fig 3.2 gives the associated proof system. These rules are used to build proof trees for formulas. I also give a cut-rule in Fig 3.3, which can be thought of as the ability to use lemmas in proofs. It is not part of the Fig 3.2 because Linear Logic was proved to be cut-free: any cut-rule can be replaced by a tree built from the base connectives. [1]

---

[1] In fact, the cut *was* part of the proof system introduced for Linear Logic. *Then*, the cut-free property was proved. As a result, cut can be removed without losing completeness.

$$\frac{}{\Gamma; A \longrightarrow A}\ identity \qquad \frac{\Gamma, B; \Delta, B \longrightarrow C}{\Gamma, B; \Delta \longrightarrow C}\ absorb \qquad \frac{}{\Gamma; \Delta \longrightarrow \top}\ \top R$$

$$\frac{\Gamma; \Delta, B_i \longrightarrow C}{\Gamma; \Delta, B_1 \,\&\, B_2 \longrightarrow C}\ \& L \qquad \frac{\Gamma; \Delta \longrightarrow B \qquad \Gamma; \Delta \longrightarrow C}{\Gamma; \Delta \longrightarrow B \,\&\, C}\ \& R$$

$$\frac{\Gamma; \Delta_1 \longrightarrow B \qquad \Gamma; \Delta_2, C \longrightarrow E}{\Gamma; \Delta_1, \Delta_2, B \multimap C \longrightarrow E}\ \multimap L \qquad \frac{\Gamma; \Delta, B \longrightarrow C}{\Gamma; \Delta \longrightarrow B \multimap C}\ \multimap R$$

$$\frac{\Gamma; \emptyset \longrightarrow B \qquad \Gamma; \Delta, C \longrightarrow E}{\Gamma; \Delta, B \Rightarrow C \longrightarrow E}\ \Rightarrow L \qquad \frac{\Gamma, B; \Delta \longrightarrow C}{\Gamma; \Delta \longrightarrow B \Rightarrow C}\ \Rightarrow R$$

$$\frac{\Gamma; \Delta, B[t/x] \longrightarrow C}{\Gamma; \Delta, \forall x.B \longrightarrow C}\ \forall L \qquad \frac{\Gamma; \Delta \longrightarrow B[y/x]}{\Gamma; \Delta \longrightarrow \forall x.B}\ \forall R$$

provided that y is not free in the lower sequent.

Fig. 3.2. The proof system for a fragment of linear logic (from [8])

$$\frac{\Gamma'; \Delta_1 \longrightarrow B \qquad \Gamma; \Delta_2, B \longrightarrow C}{\Gamma'; \Delta_1, \Delta_2 \longrightarrow C}\ cut \qquad \frac{\Gamma'; \emptyset \longrightarrow B \qquad \Gamma, B; \Delta \longrightarrow C}{\Gamma'; \Delta \longrightarrow C}\ cut!$$

Both rules have the proviso that $\Gamma \subseteq \Gamma'$.

Fig. 3.3. The cut rule for a fragment of linear logic (from [8])

My goal is thus to rewrite Popeye programs into a formula that, once added to the linear context managed by this logic, can be used to model the behavior of the expressed objects. I consider my Linear Logic proof system as a logic programming language, that is, a language where left-introduction rules are only used when we attempt to prove an atomic goal (see [8] for more accurate definitions). For clarity, I will stop referring to the left or the right of an implication. Instead, I will use the terms head and body. If H is the head and B represents the body, $H \circ\!- B$ and $B \multimap H$ show where each element is placed in regards to the implication. Execution of a program is then considered as the building of a proof for some initial sequent. This, of course, is a priori non-deterministic. Finally, at some point in my translation, I will introduce λ-terms. Consequently, I assume here that my logic is capable of dealing with higher-order terms, like for example, λProlog (which itself is inspired by results presented in [5]). I will consider a very simple typing of Linear Logic formulas, basically making the difference between terms (typed tm) and formulas (typed ll). All the binary connectives exposed above are then typed $ll \rightarrow ll \rightarrow ll$. I need one last type for objects, which I will note cl.

3.1 Translation from Popeye to Linear Logic

I start by defining some basic syntactic categories for terms in Popeye. It is useful to have these so that our translation functionals can be defined as applications from one category to another. A Popeye program, when looked from a distant point of view, is composed of declarations, commands and references (variables). This leads me to the introduction of the $decl$, cmd and ref types. An additional type is needed to represent full objects. I introduce the type $class$ for this purpose. Finally, I need a type for just a simple object name (without its constructing parameters). I thus introduce *classn*. The need for these two different types for objects arises from the fact that these entities will appear in two different forms. When instantiating an object, the operation will be possible if and only if I provide the name of the object and all the needed parameters. For example, the category for ($object\ a_1\ a_2\ ...\ a_n$) in the following expression:

$$\textbf{new}\ (object\ a_1\ a_2\ ...\ a_n)\ n\backslash$$

is *class*. On the other hand, at some points in my language, I will only pass *object* as a parameter. The type for *object* alone is more complicated. It is obviously something like $ref \rightarrow ref \rightarrow ... \rightarrow class$. This is embarrassing, because it implies that the methods receiving object names alone as parameters must accept infinitely many possible types, depending on the number of arguments necessary to the creation of a given *object*. This is why I introduce a generic *classn* category.

A Popeye program can then be categorized through the rules given in Fig 3.4. As an example, consider the switch example in Chapter 2. Its various particular elements can be typed as in Fig 3.5. Notice that a second-order type is used in the typing of the **new** command. This is directly linked to the upcoming translation for this construct. I then introduce some definitions used in the translation. First, I will call variables all elements typed *ref*. This seemingly strange choice of name is directly linked to the fact that in pure OO programming, there are no such things as values: everything is a reference to an object. Second, I define the notion of local variable.

.	:	$decl \to decl \to decl$
self	:	$ref \to decl$
private	:	$classn \to ref \to decl$
static	:	$classn \to ref \to ref \to decl$
$\succ\!\!-$	:	$cmd \to cmd \to decl$
cons	:	$ref \to class \to cmd$
$>>$	:	$cmd \to cmd \to cmd$
$+>...\|>$	:	$cmd \to cmd \to cmd \to cmd$
rw	:	$ref \to ref \to ref \to cmd$
init	:	$ref \to ref \to cmd$
new	:	$class \to (ref \to cmd) \to cmd$
super	:	$ref \to class \to cmd$
empty	:	cmd
pcst	:	$classn \to ref$
pint	:	$int \to ref$
pbool	:	$boolean \to ref$

., $>>$ and (+>...|>) are right-associative

Fig. 3.4. Syntactic categories for Popeye

$switch$	:	$class$
$sw, state, o, s$	:	ref
get	:	$ref \to ref \to cmd$
set	:	$ref \to ref \to cmd$
$toggle$	:	$ref \to cmd$

Fig. 3.5. Syntactic typing for the switch example

Definition 2 (Local variables). Local variables are those variables in a declaration that are limited to the scope of this declaration. Given a clause $A \succ\!\!- B$, I will informally write $Local(A \succ\!\!- B)$ to refer to this set. ∎

In the case of the switch example:

$$Local\ (get\ sw\ s \succ\!\!-\ rw\ state\ s\ s\ ) = \{s\}$$

Finally, I define atomic commands in Popeye.

Definition 3 (Atomic commands). Atomic commands are those commands that neither contain any of the Popeye connectives (>> , +>, |>) nor are a **rw**, a **init**, a **new** nor a **empty** command. ∎

Through this definition, I isolate those elements that are not built-in the language, but those introduced by the user. This includes the methods and constructors introduced. I am now ready to give the formal specification of the translation functions. These basically crop down to the definition of three functions: $\mathcal{R}$, $\mathcal{L}$ and $\mathcal{D}$.

3.1.1 Translation of variables

Definition 4 ($\mathcal{R}$). The function $\mathcal{R} : ref \longrightarrow tm$ is defined as: $\forall_{ref} A : \mathcal{R}(A) = A$ ∎

This straightforward definition simply shows that variables are untouched in the translation. In the remaining sections, I will systematically omit calling upon $\mathcal{R}$, simply writing the variables in Linear Logic as they appear in Popeye.

3.1.2 Translation of declarations

Declarations are translated through $\mathcal{D}$. The definition of this function is given in Fig 3.6. Its purpose is to guard adequately the variables introduced in the program.

$$
\begin{array}{rcl}
\mathcal{D}(\mathbf{self}\ \ N\ .\ Prog) & = & \lambda N.(\mathcal{D}(Prog)) \\
\mathcal{D}(\mathbf{private}\ \ C\ P\ .\ Prog) & = & \exists P.(\mathcal{D}(Prog)) \\
\mathcal{D}(\mathbf{static}\ \ C\ P\ V\ .\ Prog) & = & !(pv\ (pcst\ C\ P)\ V) \otimes \mathcal{D}(Prog) \\
\mathcal{D}(A \succ\!\!-\ B\ .\ Prog) & = & !(\forall_{cmd} K.\forall_{ref} Local(A \succ\!\!-\ B). \\
 & & \quad \mathcal{L}(A,\ K) \circ\!\!- \mathcal{L}(B,\ K)) \otimes \mathcal{D}(Prog)
\end{array}
$$

Fig. 3.6. Definition for $\mathcal{D} : decl \rightarrow ll$

The chosen **self** name becomes a λ-term, thus permitting to generate many instances of the same object by simple α-substitution. The private variables are guarded through the use of the existential quantifier. To find it at this place might seem problematic in the first place because of the grammar exposed in Fig 3.1. We implicitly intend to use this formula in a proof of the form

$$(\exists c.E) \Rightarrow G$$

which, in our logic, is not an allowed clause. But in this case (and only in this case), the meaning of the above implication can be easily seen as

$$\forall c.(E \Rightarrow G)$$

which, on the other hand, is a perfectly legal clause. This is by no mean a generality, and if I started to mix universal and existential quantifier on the body side of the above implication, it would soon become meaningless. Further discussion of the matter can be found in [12] and [13].

As for the local variables, they are guarded through the careful addition of universal quantifiers. In the end, this allows the formula to be adapted to a given use in the computation.

Another key aspect of Popeye arise in the translation of $\succ\!\!-$ terms: the use of continuation passing style (CPS). One of the main concern when moving over to logic is to make sure that the order of execution is still respected. In terms of proof, we must make sure that what happens first is proved first. Consider for example a typical sequence of operations in Popeye:

$$A_1 \;>>\; A_2 \;>>\; ... \;>>\; A_n$$

We must insure that the logical expressions equivalent to $A_1,\ A_2\ ...A_n$ are proved in the same order. CPS is a common technique to achieve this goal. Accurate description of this technique can be found in [17] and [19]. The bottom line is to consider A_1 as a function that takes as a final argument A_2. A_2 itself takes as a final argument A_3 and so on. In our framework, this equates to writing the formula above as:

$$\lambda K.A_1(\ A_2(\ ...\ (\ A_n\ K\ )\ ...\ ))$$

Milner's π-calculus, for instance, can be seen to use this style of control. As a consequence, a command A of type *cmd* is translated in Linear Logic to a term typed $ll \to ll$. The additional ll term needed is a continuation, i.e. a logical expression to be evaluated next. This explains why my declaration clause for $\succ\!\!-$ introduces a variable K that basically fills this hole in the definition of the method. When executing a program, we will be able to replace this K by the current continuation, that is, the current remaining instructions in the execution.

3.1.3 Translation of commands

Formulas are generated for commands through $\mathcal{L}$. Fig 3.7 gives the definition. One key element of this translation is the logical function

$$pv : tm \to tm \to ll$$

which is basically used for the sole purpose of storing values in the Linear Context, each of these values being attached to a private variable of the object being defined. The first argument is the private variable, the second is the value to be stored "in" it. By putting this fact into the linear context, we can allow its value to change over the proof (or execution) of a given program in logic. $\mathcal{L}$ takes two parameters, namely a command in Popeye and a goal in Linear Logic. The intended meaning is to generate the formula corresponding to the first parameter and put the second parameter as its continuation. The use of CPS is particularly visible in the definition related to $>>$. Notice that

$$
\begin{array}{rcll}
\mathcal{L}((A\ >>\ B),\ G) & = & \mathcal{L}(A,\ \mathcal{L}(B,\ G))) & \\
\mathcal{L}((A +\!> B \,|\!> C),\ G) & = & (\mathcal{L}(A,\ \top) \otimes \mathcal{L}(B,\ G)) \oplus & \\
 & & (not(\mathcal{L}(A,\ \top)) \otimes \mathcal{L}(C,\ G)) & \\
\mathcal{L}((\mathbf{rw}\ A\ O\ N),\ G) & = & (pv\ A\ O) \otimes ((pv\ A\ N) \multimap G) & \\
\mathcal{L}((\mathbf{init}\ A\ N),\ G) & = & (pv\ A\ N) \multimap G & \\
\mathcal{L}(\mathbf{new}\ C\ A\backslash\ K), G) & = & \mathbf{new}\ C\ \lambda A.\, \mathcal{L}\,(K,\ G) & \\
\mathcal{L}(\mathbf{super}\ A\ C), G) & = & \mathbf{super}\ A\ C\ G & \\
\mathcal{L}(\mathbf{empty}\ ,\ G) & = & G & \\
\mathcal{L}(A,\ G) & = & (A\ G) & A\ is\ atomic
\end{array}
$$

Fig. 3.7. Definition for $\mathcal{L} : (cmd \times ll) \to ll$

new and **super** are untouched by $\mathcal{L}$. For clarity, I prefer to write their meaning in logic, as part of the interpreter.

3.2 Interpreter

It is now very easy to build an interpreter for Popeye in Logic. If we consider that we already have an interpreter for Linear Logic (like, say Lolli), then all is needed is a few additional $\circ\!-$ clauses. The real challenge is to respect how objects are built. This implies being able to enforce inheritance and run the constructors each time an object is created. This is done by carefully writing the rule for

$$\mathbf{new}\ : cl \to (tm \to ll) \to ll \to ll$$

in Linear Logic. I first introduce a binding element that relates the name for an object to its actual logic formula: $lmod : cl \rightarrow ll \rightarrow ll$. For example, the solution to

$$? - interp\ (lmod\ switch\ P)$$

in a given Prolog-like interpreter would yield as a solution for P the actual formula defined in Fig 3.9. The clause for **new** can then be easily built and is shown in Fig 3.8. The idea is to enforce the execution of constructors whenever a new object is created. Given an object C, with all its constructing parameters, I generate a new name A which will be the object's self name during the rest of the program execution. I then call the code for a generic object of the type of C. I instanciate it with the fresh name A, then add it to the linear context. Next, I call the constructor. K is, in fact, a lambda term that puts an abstraction on the object's self name. I thus instanciate it with, once again, A. The issue of inheritance is treated by putting **super** statements in the constructors. This time, C is a mother class for the object being constructed. When such a statement is reached, the name A to use for the object is already known, hence the clause does not have a universal quantifier. I call the code for the required mother class, instanciate it with A and call upon the constructor in the mother class. As a consequence, I add further code bound to the same object name, which is exactly what I want. The careful reader might have notice two interesting consequences to this approach. First, objects are built bottom-up. That is, the first clauses added in the contexts are those of the lowest object in the inheritance hierarchy. Second, and mainly as a consequence, our system does not enforce overloading. To do so, it is necessary to add additional control

$$\begin{array}{rcl} \textbf{new}\ C\ K & \circ\!- & \forall A.\ lmod\ C\ Prog\ \otimes ((Prog\ A) \multimap \textbf{cons}\ A\ C\ (KA)) \\ \textbf{super}\ A\ C\ K & \circ\!- & lmod\ C\ Prog\ \otimes ((Prog\ A) \multimap \textbf{cons}\ A\ C\ (KA)) \end{array}$$

Fig. 3.8. Interpreter for Popeye

when we build the objects. Before adding a clause to the context, we check if some clause with the same head has not been added yet. It is not a very difficult feature to add, but it has the result of making the interpreter rules far less readable.

There is one last element that needs to be specified in logic. The **cons** traversed all my translations untouched. I simply define it as a fact in linear logic, without any intended operational semantic. A type is needed, though. **cons** can be easily typed as

$$\textbf{cons}\ : ref \to cl \to ll \to ll$$

in Linear Logic.

3.3 Switch example

I now give an example of how the formula for a given object is generated by building it for the switch object. For clarity, I will start with methods. First, let's generate the formula for the constructor.

$$\mathcal{D}(\textbf{cons}\ sw\ switch \succ\!\!-\ \textbf{init}\ state\ (boolean\ false)) =$$
$$!\,\forall K.\forall old.$$
$$\mathcal{L}((\textbf{cons}\ sw\ switch),\ K) \circ\!-$$
$$\mathcal{L}((\textbf{init}\ state\ (boolean\ false)),\ K)) \otimes \mathcal{D}(Prog) =$$
$$!\,\forall K.\forall old.$$
$$(\textbf{cons}\ sw\ switch\ K) \circ\!-$$
$$((pv\ state\ (boolean\ true)) \multimap K)$$

Notice how the **cons** keyword, treated as an atom, traverses the translation to end up untouched in the logic formula. On the other hand, the **rw** clause is pure linear logic. Also, it is interesting to see how the goal K propagates itself in the continuations. I will now give the formulas for the other methods.

$$\mathcal{D}(get\ sw\ s \succ\!\!-\ \textbf{rw}\ state\ s\ s) =$$
$$!\,\forall K.\forall s.$$
$$get\ sw\ s\ K \circ\!-$$
$$(pv\ state\ s) \otimes ((pv\ state\ s) \multimap K)$$

$$\mathcal{D}(set\ sw\ s \succ\!\!-\ \textbf{rw}\ state\ old\ s) =$$
$$!\,\forall K.\forall old, s.$$
$$set\ sw\ s\ K \circ\!-$$
$$(pv\ state\ old) \otimes ((pv\ state\ s) \multimap K)$$

$$\mathcal{D}(toggle\ sw\ \succ\!\!-\ \mathbf{rw}\ state\ (pbool\ true)\ (pbool\ true) {+}{>}$$

$$set\ sw\ old\ (pbool\ false)\ |{>}\ set\ sw\ old\ (pbool\ true)) =$$

$$!\forall K.\forall old.$$

$$toggle\ sw\ K \circ\!-$$

$$((pv\ state\ (pbool\ true)) \otimes ((pv\ state\ (pbool\ true)) \multimap \top) \otimes$$

$$(pv\ state\ old) \otimes ((pv\ state\ (pbool\ false)) \multimap K))$$

$$\oplus$$

$$(not((pv\ state\ (pbool\ true)) \otimes ((pv\ state\ (pbool\ true)) \multimap \top)) \otimes$$

$$(pv\ state\ old) \otimes ((pv\ state\ (pbool\ true)) \multimap K))$$

I can bring some simplifications to the last declaration by using the fact that

$$A \otimes (A \multimap \top) \equiv A \otimes \top$$

is provable in Linear Logic. It is then possible to rewrite the branch instruction in a shorter fashion. Once this is done, all that remains is to treat the **self** and the **private** declarations, and also connect all the above formulas with a $\otimes$. Fig 3.9 gives the completed formula for the switch object.

Appendix 8.1 provides a complete interpreter for Popeye written in λProlog. The code for the switch object can be found in Appendix 8.4.1. Inheritance will be demonstrated in Chapter 6.2, where we construct a complete architecture of logical gates.

$$
\begin{array}{l}
\lambda sw. \\
\exists state.(\\
\ (\\
\ !\forall K.\forall old. \\
\quad (\mathbf{cons}\ sw\ switch\ K) \circ\!- \\
\quad ((pv\ state\ (pbool\ true)) \multimap K) \\
\ ) \\
\otimes \\
\ (\\
\ !\forall K.\forall s. \\
\quad get\ sw\ s\ K \circ\!- \\
\quad (pv\ state\ s) \otimes ((pv\ state\ s) \multimap K) \\
\ ) \\
\otimes \\
\ (\\
\ !\forall K.\forall old, s. \\
\quad set\ sw\ s\ K \circ\!- \\
\quad (pv\ state\ old) \otimes ((pv\ state\ s) \multimap K) \\
\ ) \\
\otimes \\
\ (\\
\ !\forall K.\forall old. \\
\quad toggle\ sw\ K \circ\!- \\
\quad ((pv\ state\ (pbool\ true)) \otimes \top)\ \& \\
\quad\ (pv\ state\ old) \otimes ((pv\ state\ (pbool\ false)) \multimap K)) \\
\quad \oplus \\
\quad (not((pv\ state\ (pbool\ true))\otimes)\top)\ \& \\
\quad\ (pv\ state\ old) \otimes ((pv\ state\ (pbool\ true)) \multimap K)) \\
\ ) \\
)
\end{array}
$$

Fig. 3.9. Linear Logic formula for the switch object

Chapter 4

Types and ambiguities in Popeye

I will now study the "philosophical" issues that arise when we think of a Popeye program as an imperative program. Java is a strongly typed language, which means that all variables in the body of a function must be carefully defined. Popeye is definitely not so strong. I will need to infer the type of most variables in the body of my clauses. When I talk about types, I no longer deal with syntactic categories. I am now looking at the data-types of the objects being manipulated.

$tmod$	:	Object definition(module)
$tprog$	:	Set of clauses
$tclause$	:	Clause
$tcmd$	:	Command
$ttest$	:	Test
$theadin$	:	Head of a clause before having typed the body
$theadout$	:	Head of a clause after having typed the body
$tref\ [objname]$	:	Reference to an object
$tint$	:	Integer
$tbool$	:	Boolean

Fig. 4.1. Types for inference

4.1 Ambiguities in Popeye

Consider the following command in Popeye. Suppose, for instance, that it is part of the body of a method.

$$\mathbf{rw}\ private_1\ S\ S\ \ >>\ \ \mathbf{rw}\ private_2\ S\ S$$

In logic, this instruction makes perfect sense.

> *"First, unify S with the content of* $private_1$*, then attempt the unification of S with the content of* $private_2$*."*

This is, at the bottom line, some kind of test between the two private variables. If I now take the point of view of a Java programmer, this sentence means something completely different.

> *"First, get the value of* $private_1$ *in S, then rewrite S with the value of* $private_2$*."*

The two readings of the same instruction demonstrate the major conflict between these two paradigms. In Logic, a variable that as been unified cannot see its value change in a proof. Hence if it is re-used in the continuation, it can only be so in a test. If I want to translate logic to Java, I need to make sure that not only such ambiguities are eliminated but also that the two above point of views can be expressed somehow. I first identify my enemies. In an imperative context, a command can be seen as part of one of the following three categories: $\mathcal{T}$**est**, $\mathcal{R}$**ead** or $\mathcal{W}$**rite**.

Definition 5 (Categories of commands). A command is...

... a $\mathcal{T}$**est** command if it performs a test on the current program state.

... a $\mathcal{R}$**ead** command if it extracts a data from the current program state.

... a $\mathcal{W}$**rite** command if it modifies a data in the current program state. ∎

It is important to see that depending on the context, a command can be at some point in any of the three categories. Consider for example a single **rw** command. Depending on the context, it will either read a private variable, write it or test it against some other data.

Definition 6 (Ambiguities in Popeye). A command is ambiguous if it can belong to two categories of commands in two different contexts. ∎

From this point on, my goal will be to detect and forbid these ambiguities by enforcing additional restriction on the writing of Popeye programs.

4.2 Type inference

Consider a clause $A \succ\!\!- \; B$. The body B has been carefully typed by one of the typing commands given in Chapter 2. My goal is to infer a type for all the variables introduced in B. Initially, I can mark these variables as of type ".", that is, undefined. After having swept through B, I want to have all these local variables typed. For this, I will define a set of typing rules that will infer this (a classical approach that can be found in ML, for instance). I have, though, an additional claim. If I can infer the types of all local variables in $A \succ\!\!- \; B$, then I can guarantee that all ambiguities have been avoided.

The classical $\theta \vdash x : \tau$ typing sequent has been modified in order to perform a clean typing of continuations. The sequent

$$\theta \vdash x, cont : \tau, \tau_{cont}$$

must be read as follows:

> *"With initial type context θ, prove (infer) that expression x is of type τ, then with this information added to θ, proceed to prove that the next element in cont can be typed (inferred) as the first type in τ_{cont}"*

where *cont* and τ_{cont} are lists of commands and types respectively. These two lists represent a stack of things that need to be typed next. This strange structure is needed to account for the fact that the typing environment θ changes as we follow a continuation. It actually expands with new bindings that gives a type to previously untyped elements. By adding $(A : \tau)$ on top of θ, I hide the initial $(A : .)$ binding. Hence, if I try to type A afterwards against θ, it will be typed τ. The use of the stack becomes particularly understandable by looking at the *cont* rule.

$$\frac{\theta \vdash A, (B, Cont) : tcmd, (tcmd, TC)}{\theta \vdash (A \ >> \ B), Cont : tcmd, TC} \ cont$$

Notice how the continuation is broken among the various parameters of the typing sequent. The way my context grows in the typing of a continuation is different from what happens in a classical proof tree. In such a picture, the result of typing (solving) a branch of the proof tree cannot have any impact on the typing of some other branch. In my case, though, I want the information gathered in the typing of A in $A\ >>\ B$ to be used in the typing of B. This is closely linked to the other goal I am seeking.

I use this peculiar notation in another way. At many points in my type system, I will need to write sequents indicating that we should now proceed to type what is next in *cont*. For this purpose, I simply write:

$$\theta \vdash none, Cont : ., TC$$

I then provide my system with a *transfer* rule:

$$\frac{\theta \vdash B, Cont : TB, TC}{\theta \vdash none, (B, Cont) : ., (TB, TC)}\ transfer$$

And the next element B in *cont* gets typed afterward.

It is also very important to understand that the rules given here are not enough to provide a complete typing of a Popeye program. The type declaration given in 2.6 will be used to generate additional typing rules that will complete the system. Before giving these rules, I will first lay out how a module is to be typed from a global point of view. First, I start by looking at the initial declarations. The **self** declarations result into typing the name used for the object as a reference to that object. The private variables are also bound to their respective types, depending on how they are typed in the declarations. Finally, I reach the clauses that depict the methods of the object. As a whole, they are typed *tprog*, with each clause to be typed *tclause*.

Before entering the clause, I need to insure that all local variables in it get assigned an undefined type. In this idea, let me informally define $TLocal$ as follows:

Definition 7 ($TLocal$ function). The function $TLocal$ is defined as the function that, given a declaration $(A \succ\!\!- \ B)$, returns the set θ such that:

$$\forall_{ref} x : \ x \in Local(A \succ\!\!- \ B) \Rightarrow (x : .) \in \theta$$

∎

The typing process is very different depending on whether we have a in-method or a out-method (c.f. Definition 1). In the case of a in-method, I know how to type every parameters of the method, hence I can use that information to check on the body of the function. In the case of a out-method, the last parameter should have the type specified in the **mtype** declaration, but that is something we will be able to check only after having examined the body. As a conclusion, it can be seen that the head of a method needs to be examined twice, before and after having typed the body. This is why I have two types, *theadin* and *theadout* against which to check a clause head. This also justifies the *clause* typing rule:

$$\frac{TLocal(A \succ\!\!- \ B), \theta \vdash A, (B, A, Cont) : theadin, (tcmd, theadout, TC)}{\theta \vdash A \succ\!\!- \ B, Cont : tclause, TC} \ clause$$

Typically, the **mtype** command will generate the two head rules for the method.

The last important thing I need to define to make my system coherent is how to type-check objects that inherits from others. Consider for example a *lemonade* object inheriting the *beverage* abstract class. If A is a reference to *lemonade*, A should be able to access all methods of the *beverage* class without any difficulties. This compatibility can be expressed as follows: if A must be of type T_1 but is of type T_2, and that T_2 inherits T_1, then the type-checking must succeed. This gives rise to the *extension* rule:

$$\frac{\theta \vdash A : (tref\ TB) \quad inherits\ TB\ TA \quad \theta \vdash none, Cont : ., TC}{\theta \vdash A, Cont : (tref\ TA), TC}\ extension$$

The *inherits* predicate is better understood by looking at its potential definition in a Prolog-like language:

```
inherits A B :- extends A C, inherits C B.

inherits A B :- extends A B.
```

The `extends` predicates are generated by the Popeye **extends** keyword.

Fig 4.2 give the type-checking rules for Popeye reserved words, with the exception of the conditional construct that will be discussed in its own section.

4.3 Rules generation

I now explain how to generate the typing rules for objects, private variables, constants and methods. To do so, I examine each typing directive. First, some conventions must be outlined about the translation of types as defined in Chapter 2 and those used here. I first introduce the syntactic category *ty* of the types in Popeye. I call *tty* the category of types obtained by chaining multiple *ty* types with $\rightarrow$ (like in λProlog, for instance). Finally, let *io* be the syntactic category that can contain either the word **in** or **out**. This allows me to type the typing directives accurately in Fig 4.3. I then define

$$\frac{(A:\tau)\in\theta}{\theta\vdash A:\tau}\ fetch$$

$$\frac{}{\theta\vdash none,\emptyset:.,\emptyset}\ axiom$$

$$\frac{}{\theta\vdash dump,\emptyset:.,tdump\ \theta}\ dump$$

$$\frac{\theta\vdash B,Cont:TB,TC}{\theta\vdash none,(B,Cont):.,(TB,TC)}\ transfer$$

$$\frac{\theta\vdash A:TA\quad \theta\vdash none,Cont:.,TC}{\theta\vdash A,Cont:TA,TC}\ lookup$$

$$\frac{\theta\vdash A:(tref\ TB)\quad inherits\ TB\ TA\quad \theta\vdash none,Cont:.,TC}{\theta\vdash A,Cont:(tref\ TA),TC}\ extension$$

$$\frac{\theta\vdash A:\tau\quad \theta\vdash N:\tau\quad \theta\vdash none:Cont.TC}{\theta\vdash(\mathbf{init}\ \ A\ N),Cont:tcmd,TC}\ init$$

$$\frac{\theta\vdash A:\tau\quad \theta\vdash O:.\quad \theta\vdash N:\tau\quad (tp\ O\ \tau),\theta\vdash none,Cont:.,TC}{\theta\vdash(\mathbf{rw}\ \ A\ O\ N),Cont:tcmd,TC}\ rw-write$$

$$\frac{\theta\vdash A:\tau\quad \theta\vdash O:.\quad \theta\vdash N:.\quad (tp\ O\ \tau),(tp\ N\ \tau),\theta\vdash none,Cont:.,TC}{\theta\vdash(\mathbf{rw}\ \ A\ O\ N),Cont:tcmd,TC}\ rw-read$$

$$\frac{\theta\vdash A:\tau\quad \theta\vdash O:\tau\quad \theta\vdash N:\tau\quad \theta\vdash none,Cont:.,TC}{\theta\vdash(\mathbf{rw}\ \ A\ O\ N),Cont:ttest,TC}\ rw-test$$

$$\frac{\theta\vdash A,(B,Cont):tcmd,(tcmd,TC)}{\theta\vdash(A\ \ >>\ \ B),Cont:tcmd,TC}\ cont$$

$$\frac{TLocal(A\succ\!\!-\ \ B),\theta\vdash A,(B,A,Cont):theadin,(tcmd,theadout,TC)}{\theta\vdash A\succ\!\!-\ \ B,Cont:tclause,TC}\ clause$$

$$\frac{\theta\vdash A,\emptyset:tclause,\emptyset\quad \theta\vdash B,Cont:tprog,TC}{\theta\vdash A\ .\ B,Cont:tprog,TC}\ prog$$

$$\frac{\theta\vdash A:\tau\quad (tp\ \ A\ \tau),\theta\vdash Prog,Cont:tprog,TC}{\theta\vdash\mathbf{private}\ \ A\ Prog,Cont:tprog,TC}\ private$$

Fig. 4.2. Typing rules for Popeye

$$\begin{array}{rcl}
\textbf{otype} & : & ref \rightarrow tty \rightarrow decl \\
\textbf{ctype} & : & ref \rightarrow tty \rightarrow decl \\
\textbf{ptype} & : & ref \rightarrow tty \rightarrow decl \\
\textbf{mtype} & : & ref \rightarrow io \rightarrow tty \rightarrow decl
\end{array}$$

Fig. 4.3. Typing for typing directives

a binding between Popeye base types and those types I defined in Fig 4.1. This function $\mathcal{T}$ is defined in Fig 4.4. I am now ready to give the rules for each directive.

$$\begin{array}{rcl}
\mathcal{T}(ref\ N) & = & tref\ N \\
\mathcal{T}(int) & = & tint \\
\mathcal{T}(boolean) & = & tbool
\end{array}$$

Fig. 4.4. Definition for $\mathcal{T} : ty \rightarrow pt$

The figures whose titles start with "Generation rules..." must be understood as templates. These templates are to be instanciated with the types and names used in the Popeye type declarations. For example, consider the rules to generate for the **otype**declaration. This declaration creates five rules in the type system. **otype**takes various types (t_1. t_2 ... t_n) and a *name* as parameters. The type used as t_1 should replace every occurence of t_1 in the rules. The same goes for t_2, ... t_n and *name*. Everything else is to be written as it appears in the given rules. To check the parameters passed to the typing directive, check the expression given after "Initial directive is:" in each figure.

4.3.1 otype

For each object, there are four things that I need to be able to type correctly. Fig 4.5 gives the complete set of rules to generate. First, the object module itself. This is done through the *name* rule. I basically invoke the corresponding program and then give the correct type for all the parameters simply by looking at the **otype** directive. After that, I continue with the typing of the rest of the program. Second, I need rules to take care of the constructor. This is done by the $cons - headin$ and $cons - headout$ rules, the second rule being basically inactive. The *headin* verifies the validity of the parameters passed. Finally, we need to take care of the **new** command. This is done through the *new* rule which basically checks the parameters, then types the name to be used in the remaining program for the reference to the new object. Once this is done, the remaining program is typed in the light of this new environment. The *super* types the **super** command. The technique is similar to that of **new** , except that there is no slash nor program to look at afterwards.

4.3.2 ctype

Constants have easier rules. It is just a matter of verifying that the actual value passed to the constant is coherent with the type assigned to it in the declarations. Fig 4.6 gives the single rule to generate.

Initial directive is:
otype *name* $t_1 \rightarrow t_2 \rightarrow ... \rightarrow t_n \rightarrow$ class.

$$\frac{\begin{array}{c}(A_1 : \mathcal{T}(t_1))...\ (A_n : \mathcal{T}(t_n)), (S : (tref\ name)), \theta \vdash M, Cont : tprog, TC \\ lmod\ (name\ A_1\ ...\ A_n)\ (\textbf{self}\ S\ .\ M)\end{array}}{\theta \vdash name, Cont : tmod, TC}\ name$$

$$\frac{\begin{array}{c}\theta \vdash none, Cont : ., TC \\ \theta \vdash S : tref\ name \quad \theta \vdash A_1 : \mathcal{T}(t_1) \quad \theta \vdash A_n : \mathcal{T}(t_n)\end{array}}{\theta \vdash \textbf{cons}\ S\ (name\ A_1\ ...\ A_n), Cont : theadin, TC}\ cons - headin$$

$$\frac{\theta \vdash none, Cont : ., TC}{\theta \vdash \textbf{cons}\ S\ (name\ A_1\ ...\ A_n), Cont : theadout, TC}\ cons - headout$$

$$\frac{\begin{array}{c}(S : (tref\ name)), \theta \vdash P, Cont : tcmd, TC \\ \theta \vdash A_1 : \mathcal{T}(t_1) \quad ... \quad \theta \vdash A_n : \mathcal{T}(t_n)\end{array}}{\theta \vdash \textbf{new}\ (name\ A_1\ ...\ A_n)\ S\ P, Cont : tcmd, TC}\ new$$

$$\frac{\begin{array}{c}\theta \vdash none, Cont : ., TC \\ \theta \vdash A_1 : \mathcal{T}(t_1) \quad ... \quad \theta \vdash A_n : \mathcal{T}(t_n)\end{array}}{\theta \vdash \textbf{super}\ S\ (name\ A_1\ ...\ A_n), Cont : tcmd, TC}\ super$$

Fig. 4.5. Generation rules for **otype**

Initial directive is:
ctype *objname name* t_1.

$$\frac{\theta \vdash none, Cont : ., TC}{\theta \vdash \textbf{pcst}\ objname\ name, Cont : \mathcal{T}(t_1), TC}\ name - const$$

Fig. 4.6. Generation rules for **ctype**

4.3.3 ptype

Private variables need additionnal work. I introduce a *ppriv* keyword that I use to link the name of the private variable to the actual object. This is necessary in case we use the same name in two different objects. Fig 4.7 gives the corresponding rule.

Initial directive is:
ptype *objname name* t_1.

$$ppriv : classn \rightarrow ref \rightarrow ref$$

$$\frac{\theta \vdash none, Cont : ., TC}{\theta \vdash ppriv\ objname\ name, Cont : \mathcal{T}(t_1), TC}\ name - priv$$

Fig. 4.7. Generation rules for **ptype**

4.3.4 mtype

This is the heart of my type system, as it is the place where ambiguities will not be tolerated. To understand how this happens, it is better to consider what should be done in the case of a simple **rw** command. In a way, this can be seen as the simplest methods available to the programmer. The same goes for all other Popeye commands. In the typing of the body of a clause, we might reach the **rw** command with many different typing environment. Fig 4.8 summarizes these different environment on the left hand-side. The key idea is to notice that some of these environments make no sense at

Type before			Semantic	Type after		
priv	*O*	*N*		*priv*	*O*	*N*
τ	.	.	Read Only if N = O	τ	τ	τ
τ	τ	.	Forbidden	$\times$	$\times$	$\times$
τ	.	τ	Write	τ	τ	τ
τ	τ	τ	Test. Only in front of +>	τ	τ	τ

Fig. 4.8. Types and meanings for **rw**

all in regards to the actual role **rw** plays. As for those that do make sense, they are all linked to a very precise semantics.

The same goes for more general methods. For an in-method, the only sensible typing environment is to have all the parameters typed correctly before actually calling onto it. All other configurations are simply not allowed. An out-method can basically take two kinds of typing environments. One of them is when we simply use the method to get a value. In that case, everything but the returned parameter needs to be typed. On the other hand, all parameters might also be typed. In that case, it means we are attempting a test.

As one can see, by doing so, I have separated the diverse categories of commands. In fact, I can write the following property:

Property 4.1 (Typing rules for methods). For a given method, for each category of command it belongs to, one typing rule (and only one) is generated.

Proof: By construction of the generation rules. That's the way we built these rules: each of them reflect one possible meaning of the method being considered.

∎

Fig 4.9 gives the rules to generate in-methods. Fig 4.10 lists those necessary for out-methods.

Initial directive is:
mtype $name\ in\ \ (ref\ objname) \rightarrow t_2 \rightarrow ... \rightarrow t_n \rightarrow cmd.$

$$\frac{\begin{array}{c}\theta \vdash none, Cont : ., TC \\ \theta \vdash A_1 : (tref\ objname)\ \ ...\ \ \theta \vdash A_n : \mathcal{T}(t_n)\end{array}}{\theta \vdash name\ A_1\ ...\ A_n, Cont : tcmd, TC}\ name - write$$

$$\frac{\begin{array}{c}(A_2 : \mathcal{T}(t_2)), ..., (A_n : \mathcal{T}(t_n)), \theta \vdash none, Cont : ., TC \\ \theta \vdash A_1 : (tref\ objname)\end{array}}{\theta \vdash name\ A_1\ ...\ A_n, Cont : theadin, TC}\ name - headin$$

$$\frac{\theta \vdash none, Cont : ., TC}{\theta \vdash name\ A_1\ ...\ A_n, Cont : theadout, TC}\ name - headout$$

Fig. 4.9. Generation rules for **mtype** with in-methods

Initial directive is:
mtype $name\ out\ \ (ref\ objname) \rightarrow t_2 \rightarrow ... \rightarrow t_n \rightarrow cmd.$

$$\frac{\begin{array}{c}(A_n : \mathcal{T}(t_n)), \theta \vdash none, Cont : ., TC \\ \theta \vdash A_1 : (tref\ objname)\ \ ...\ \ \theta \vdash A_{n-1} : \mathcal{T}(t_{n-1})\ \ \theta \vdash A_n : .\end{array}}{\theta \vdash name\ A_1\ ...\ A_n, Cont : tcmd, TC}\ name - read$$

$$\frac{\begin{array}{c}\theta \vdash none, Cont : ., TC \\ \theta \vdash A_1 : (tref\ objname)\ \ ...\ \ \theta \vdash A_n : \mathcal{T}(t_n)\end{array}}{\theta \vdash name\ A_1\ ...\ A_n, Cont : ttest, TC}\ name - test$$

$$\frac{\begin{array}{c}\theta \vdash none, Cont : ., TC \\ \theta \vdash A_1 : (tref\ objname)\ \ ...\ \ \theta \vdash A_{n-1} : \mathcal{T}(t_{n-1})\end{array}}{\theta \vdash name\ A_1\ ...\ A_n, Cont : theadin, TC}\ name - headin$$

$$\frac{\begin{array}{c}\theta \vdash none, Cont : ., TC \\ \theta \vdash A_n : \mathcal{T}(t_n)\end{array}}{\theta \vdash name\ A_1\ ...\ A_n, Cont : theadout, TC}\ name - headout$$

Fig. 4.10. Generation rules for **mtype** with out-methods

4.4 Typing rule for the conditional operator

The conditional operator is not as easily typed as it could appear. Consider a classical conditional statement:

$$Before\ >>\ (A +> B \,|> C)\ >>\ After$$

It is perfectly possible that some variable will be used in B but not in C. But regardless of that, it is capital that the inferred type for this variable propagate in $After$. Hence, I must stick together the result of typing B and that of typing C as the new typing context to use for the typing of $After$. This requires some work. First, I type-check the condition A, then, type-check B and C, each time noting the environment obtained at the end. I then strip these environments of undefined bindings. This is done so that when I stick the two of them together, some residual undefined bindings do not hide other valid ones. A function *nodots* is introduced for this purpose, that given an environment θ, returns this environment without the undefined bindings. I glue the environments together and then proceed with this new type environment. Fig 4.11 gives the precise typing rule. My

$$\frac{\theta \vdash A : ttest \qquad \begin{array}{r} nodots(Leaf B), nodots(Leaf C), \theta \vdash none, Cont : ., TC \\ \theta \vdash C, dump : tcmd, (tdump\ Leaf C) \\ \theta \vdash B, dump : tcmd, (tdump\ Leaf B) \end{array}}{\theta \vdash (A +> B \,|> C), Cont : tcmd, TC}\ branch$$

Fig. 4.11. Typing rule for the conditional

choice for this operator is far from being completely satisfactory. Following our system, it is possible to use a variable in $After$ that has only be defined in B, for example. This is awkward since it is by definition not guaranteed that we will traverse B, and thus the variable can very well be a null pointer, for example. I thus have the possibility of runtime errors. In my opinion, though, this is not a real problem since this kind of behavior will undoubtedly fail to satisfy a correctness proof in Linear Logic. It will also crash in the interpreter, which is why it was designed, after all.

4.5 Typing the switch example

It is now high time to put these things into action. First, I will show how I go through the declaration part. Fig 4.12 shows the initial steps. Let $\theta = (sw : tbool); (S :$

$$\dfrac{\dfrac{\begin{array}{c}\vdots \\ (sw : tbool), (S : (tref\ switch)) \vdash M, \emptyset : tprog, \emptyset \\ \vdash (ppriv\ switch\ sw) : tbool\end{array}}{(S : (tref\ switch)) \vdash \textbf{private}\ sw\ .\ M, \emptyset : tprog, \emptyset}\ private \qquad lmod\ (switch)\ (\textbf{self}\ S\ .\ \textbf{private}\ sw\ .\ M)}{\vdash switch, \emptyset : tmod, \emptyset}\ switch$$

Fig. 4.12. Typing of the declarations in the switch example

$(tref\ switch))$. I can then proceed and show how to type, say, the constructor. I only draw the first steps, up to the point when the method has been split in the three passes (*theadin*, *tmcd* and *theadout*) (see Fig 4.13). From this point on, it is just a matter of calling the typing rules for the switch constructor and the **init** method. I will also type

$$\dfrac{\dfrac{\vdots}{TLocal(...), \theta \vdash cons..., ((init...), (cons...)) : theadin, (tcmd, theadout)}}{\theta \vdash \textbf{cons}\ switch\ sw \succ\!\!-\ ..., \emptyset : tclause, \emptyset}$$

$$\dfrac{\theta \vdash P : tprog}{\theta \vdash \textbf{cons}\ switch\ sw \succ\!\!-\ \textbf{init}\ state\ (\textbf{pbool}\ false).P, \emptyset : tprog, \emptyset}\ prog$$

Fig. 4.13. Typing of the constructor in the switch example

the *get* and *set* methods. Understanding the differences between these two methods is capital. The proof tree for *get* is given in Fig 4.14 and the one for *set* is in Fig 4.15. I have skipped the intermediary $transfer$ steps in order to enhance readability. The main difference is the fact that in the first case, s remains unknown before we enter the body of the clause whereas it is given a type in the other case. As a consequence, the clause used to infer the **rw** statement is different in each situation. In the first case, we call upon the $rw - read$ rule whereas in the second, we have to use the $rw - write$. Accidently, we have also inferred what the operational meaning of the **rw** command is in each case.

In my opinion, a lot of parallels could be drawn between this approach and the Curry-Howard isomorphism. In the case of Curry-Howard, types correspond to formulas in intuitionistic propositionnal logic. Hence the isomorphism exposes the close relationships between types and proofs in logic. What I am doing here is basically saying that types correspond to a meaning in an imperative context, which, in a way, start from the same idea.

$$
\dfrac{\dfrac{\dfrac{\begin{array}{c}\dfrac{}{(s : tbool), \theta \vdash get... : theadout}\ get - headout \\ (s : .), \theta \vdash s : . \\ (s : .), \theta \vdash state : tbool\end{array}}{(s : .), \theta \vdash \mathbf{rw}\ \ state\ \ s\ \ s, (get...) : tcmd, theadout}\ rw - read}{(s : .), \theta \vdash get\ \ sw\ \ s, ((\mathbf{rw}\ \ ...), (get...)) : theadin, (tcmd, theadout)}\ get - headin}{\theta \vdash get\ \ sw\ \ s \succ\!\!-\ \ \mathbf{rw}\ \ state\ \ s\ \ s : tclause}\ clause
$$

Fig. 4.14.   Typing of the get method in the switch example

$$
\dfrac{\dfrac{\dfrac{\begin{array}{c}\dfrac{}{(s : tbool), (o : tbool), \theta \vdash get... : theadout}\ set - headout \\ (s : tbool), (o : .), \theta \vdash s : tbool \\ (s : tbool), (o : .), \theta \vdash o : . \\ (s : tbool), (o : .), \theta \vdash state : tbool\end{array}}{(s : tbool), (o : .), \theta \vdash \mathbf{rw}\ \ state\ \ s\ \ s, (get...) : tcmd, theadout}\ rw - write}{(s : .), (o : .), \theta \vdash get\ \ sw\ \ s, ((\mathbf{rw}\ \ ...), (get...)) : theadin, (tcmd, theadout)}\ set - headin}{\theta \vdash set\ \ sw\ \ s \succ\!\!-\ \ \mathbf{rw}\ \ state\ \ o\ \ s : tclause}\ clause
$$

Fig. 4.15.   Typing of the set method in the switch example

4.6 Typing-Unambiguity correspondence

I now give the formal results that justify this type construction. The first important result is to realize the following property.

Property 4.2 (Mutual exclusion of typing rules). All the typing rules given are mutually exclusive w.r.t. a given typing environment. That is: if one rule can be used in a given context, it is the only one that can.

Proof: Let us consider the various possibilities:

- It is immediate when there is only one rule fitted for a given pattern.
- The situation where more than one rule could match a pattern arises only when we deal with possible commands or methods. By Prop 4.1, though, the result is immediate, since a given typing environment is unique.

∎

As a consequence, I can claim the following theorem.

Theorem 4.1 (Uniqueness of the solution). If an object declaration is typeable, then there is only one possible typing proof tree for it.

Proof: If a rule can be used at some point in the proof tree, then no other can by Prop 4.1.

∎

This theorem has two important corollaries. The first one basically states that I have reached my goal:

Corollary 4.1 (Ambiguity elimination). If a given object successfully type-checks, then none of the commands used in its definition is ambiguous.

Proof: If the object has been successfully type-checked, then the proof-tree is unique by Theorem 4.1. For each command, the corresponding typing rule used belongs to one category of command, since each rule is generated to match only one of them (Prop 4.1). Since this rule is the only possible one, no ambiguity remains on the semantic of each command.

∎

The second corollary is useful for our next chapter. It requires some additional formalism though. Consider the typing of individual $\succ$ clauses. Each typing is done individually, namely, the typing of each clause is done separately starting with the types of the self name, privates and constants. I define the concept of θ_{leaf} as:

Definition 8 (θ_{leaf}). The function $\theta_{leaf} : \mathit{decl} \longrightarrow \mathit{pt}$ is defined as the function that given a $\succ$ clause returns the typing environment obtained by type inference on this clause, including its local variables. ∎

The second corollary can then be expressed accurately:

Corollary 4.2 (Typing of variables). If the object is typeable, then for each $\succ$ clause, all local variable was inferred a type.

Proof: I will use a proof by contradiction. Suppose that there exists a variable x such that the only binding in θ_{leaf} for x is $(x : .)$. This basically means that x was never used in the body of the method nor passed as an argument. Hence it is not a local variable, which is contradictory. I have thus proved my initial claim.

∎

I have now achieved all the preliminary work needed to generate Java code automatically. On a practical point of view, the Popeye programmer is required to feed its programs into the type-checker, and succeed, before proceeding to the next step. Appendix 8.2 gives the code for the type-checker written in λProlog.

Chapter 5

Java code generation

In this chapter, I lay down the recursive analysis that will generate the Java code equivalent for a given Popeye program.

5.1 Notations

The first thing I need to introduce is an additional environment μ for storing the Java string corresponding to a variable. This environment is designed to work in exactly the same way as the typing environment. For all variable x, I will associate the string "$\mathtt{x}$" that is to be used in the Java code. This relationship will be represented as $(x : \mathtt{x})$. Notice that I will often omit the quotes around the $\mathtt{x}$. In a similar fashion as to what was done in the case of θ, I then introduce a $MLocal$ function as follows:

Definition 9 ($MLocal$). The function $MLocal$ is defined as this function that, given a declaration $(A \succ\!\!- \ B)$, returns the set μ such that:

$$\forall_{ref} x : \ x \in Local(A \succ\!\!- \ B) \Rightarrow (x : \mathtt{x}) \in \mu$$

∎

Everything in Popeye will be tied to a piece of Java code (a string). For example, consider the simple boolean constant true. The fact that it is written `true` in Java is written

$$\theta; \mu : (pbool\ true) \longrightarrow \mathtt{true}$$

In most cases, though, the Java code generated will be obtained by stitching together other pieces of codes obtained somewhere else. For example, consider $A\ >>\ B$. I first get the code for A in, say, `JA`. Then I get B's code in `JB`. Hence the code to generate is string `JA;JB`. I note this as follows:

$$
\begin{array}{l}
\theta;\mu : A\ >>\ B \\
\quad \left| \begin{array}{l} \theta;\mu : A \longrightarrow \mathtt{JA} \\ \theta;\mu : B \longrightarrow \mathtt{JB} \end{array} \right. \\
\longrightarrow \quad \mathtt{JA;\ JB}
\end{array}
$$

This notation can also be seen as a different presentation for a classical deduction rule in any logic. In this picture, the above expression would be re-written as follows:

$$
\frac{\theta;\mu : A \longrightarrow \mathtt{JA} \qquad \theta;\mu : B \longrightarrow \mathtt{JB}}{\theta;\mu : A >>\ \ \longrightarrow \mathtt{JA};\mathtt{JB}}
$$

I introduced this new notation for typographical readability.

There is one source of concern, though. As we have outlined in the previous chapter, there are many occasions where a given Popeye statement can be used for various purposes. Hence I need to introduce a few other mappings. I first need to be able to specify the simplest mapping.

Definition 10 ($\xrightarrow{typo}$). $\xrightarrow{typo}$ is defined as the relation that associates a Popeye expression to the exact same string in Java.

E.g.: $Lemonade \xrightarrow{typo}$ `Lemonade`

Then, I need to define the mapping for test expressions.

Definition 11 ($\stackrel{test}{\longrightarrow}$). $\stackrel{test}{\longrightarrow}$ is defined as the relation that associates a $\mathcal{T}$**est** method to the corresponding Java code. ∎

E.g.: $\theta, \mu : \textbf{rw}\ A\ O\ O \stackrel{test}{\longrightarrow} \texttt{A} == \texttt{O}$

Definition 12 ($\stackrel{head}{\longrightarrow}$). $\stackrel{head}{\longrightarrow}$ is defined as the relation that associates the head of a $>\!\!-$ clause to a triple. The first element in the triple is the set of variables that are passed as an argument to the method. The second element is the Java code for the declaration header needed for this method in the Java program. The third element is the code needed for an eventual `return` statement. ∎

E.g.: $\theta, \mu : get\ S\ V \stackrel{head}{\longrightarrow} \emptyset, \texttt{public boolean get}()\{, \texttt{return V};$

Definition 13 ($\stackrel{decl}{\longrightarrow}$). $\stackrel{decl}{\longrightarrow}$ needs three input parameters. The first one is a type environment as defined in Chapter 4. The second one is a naming environment as defined at the beginning of this chapter. The third one is a set of variables to exclude from the translation. The result is a declaration header for all the variables in the given environment, except those in the given set. ∎

E.g.: $((A : pbool), (B : pbool)),\ ((A : \texttt{A}), (B : \texttt{B})),\ B \stackrel{decl}{\longrightarrow} \texttt{boolean A};$

Definition 14 ($\stackrel{params}{\longrightarrow}$). $\stackrel{params}{\longrightarrow}$ is defined as the relation that associates a type environment to the set of variables that are typed into it. ∎

E.g.: $(A : .), (B : pbool) \stackrel{params}{\longrightarrow} B$

Typically, for each method introduced, we will have to generate one $\stackrel{head}{\longrightarrow}$ rule, and one basic $\longrightarrow$ rule. If it is an out-method, we will need to add a $\stackrel{test}{\longrightarrow}$. The last two relations introduced are easy to implement. In the case of $\xrightarrow{decl}$, what I actually do is look at a stack of variables, look at their type, and generate the corresponding declaration in Java if needed. The "if needed" is decided upon whether the considered variable belongs to a given list. In the case of the $\xrightarrow{params}$, the easiest way to represent it is as a simple filter that throws out untyped variables.

At this point, I am ready to give the rules in general.

5.2 Base rules

Fig 5.1 and 5.2 give the rules for Java code generation. The only really intricate one is the rule for methods declaration. The core idea can be outlined as follows: first, I extract the code for the Java declaration of the method. Along with it, I can extract other method-specific information. On one side, I get a list of the variables that are passed as parameters. On the other, I get the code for an eventual return statement. My next move is then to get a list of those variables already declared in the environment. By sticking together the two lists of variables, I have the list of the those variables that will not need to be declared in the scope of my method. As for the remaining variables, I then generate the appropriate declaration header. After this is done, I get the code for the body of the method. The final step is to stick everything together.

The other rules are straightforward.

$$\theta;\mu : tbool \longrightarrow \texttt{boolean}$$
$$\theta;\mu : (\textbf{pbool}\ true) \longrightarrow \texttt{true} \quad \theta;\mu : (\textbf{pbool}\ false) \longrightarrow \texttt{false}$$

$$\theta;\mu : (tint) \longrightarrow \texttt{int} \quad \theta;\mu : (\textbf{pint}\ I) \longrightarrow \texttt{I}$$

$$\theta;\mu : (tref\ object) \longrightarrow \texttt{Object}$$

$$\begin{array}{ll} \theta;\mu : A & \\ & \mid (A : \texttt{A}) \in \mu \\ \longrightarrow & \texttt{A} \end{array}$$

$$\begin{array}{ll} \theta;\mu : \textbf{init}\ A\ N & \\ & \mid \theta;\mu : A \longrightarrow \texttt{JA} \\ & \mid \theta;\mu : N \longrightarrow \texttt{JN} \\ \longrightarrow & \texttt{JA = JN;} \end{array}$$

$$\begin{array}{ll} \theta;\mu : \textbf{rw}\ A\ O\ N & \\ & \mid \theta;\mu : A \longrightarrow \texttt{JA} \\ & \mid \theta;\mu : O \longrightarrow \texttt{JO} \\ & \mid \theta;\mu : N \longrightarrow \texttt{JN} \\ \longrightarrow & \texttt{JO = JA;} \\ & \texttt{JA = JN;} \end{array}$$

$$\begin{array}{ll} \theta;\mu : \textbf{rw}\ A\ O\ O & \\ & \mid \theta;\mu : A \longrightarrow \texttt{JA} \\ & \mid \theta;\mu : O \longrightarrow \texttt{JO} \\ \xrightarrow{test} & \texttt{JO == JA;} \end{array}$$

$$\begin{array}{ll} \theta;\mu : A == B & \\ & \mid \theta;\mu : A \longrightarrow \texttt{JA} \\ & \mid \theta;\mu : B \longrightarrow \texttt{JB} \\ \longrightarrow & \texttt{JA = JB;} \end{array}$$

$$\begin{array}{ll} \theta;\mu : A == B & \\ & \mid \theta;\mu : A \longrightarrow \texttt{JA} \\ & \mid \theta;\mu : B \longrightarrow \texttt{JB} \\ \xrightarrow{test} & \texttt{JA == JB;} \end{array}$$

$$\begin{array}{ll} \theta;\mu : A \ >>\ B & \\ & \mid \theta;\mu : A \longrightarrow \texttt{JA} \\ & \mid \theta;\mu : B \longrightarrow \texttt{JB} \\ \longrightarrow & \texttt{JA JB;} \end{array}$$

$$\begin{array}{ll} \theta;\mu : A \mathrel{+\!>} B \mathrel{|\!>} C & \\ & \mid \theta;\mu : A \xrightarrow{test} \texttt{JA} \\ & \mid \theta;\mu : B \longrightarrow \texttt{JB} \\ & \mid \theta;\mu : C \longrightarrow \texttt{JC} \\ \longrightarrow & \texttt{if (JA)\{} \\ & \texttt{JB} \\ & \texttt{\} else \{} \\ & \texttt{JC} \\ & \texttt{\}} \end{array}$$

$$\begin{array}{ll} \theta;\mu : C.P & \\ & \mid \theta;\mu : C \longrightarrow \texttt{JC} \\ & \mid \theta;\mu : P \longrightarrow \texttt{JP} \\ \longrightarrow & \texttt{JC} \\ & \texttt{JP} \end{array}$$

$$\begin{array}{ll} \theta;\mu : \textbf{private}\ C\ N\ .\ P & \\ & \mid \theta \vdash ppriv\ C\ N : \tau \\ & \mid \emptyset;\emptyset : \tau \longrightarrow \texttt{JT} \\ & \mid (N:\tau),\theta;(N:\texttt{N}),\mu : P \longrightarrow \texttt{JP} \\ \longrightarrow & \texttt{JT N;} \\ & \texttt{JP} \end{array}$$

$$\begin{array}{ll} \theta;\mu : \textbf{static}\ C\ N\ V\ .\ P & \\ & \mid N \xrightarrow{typo} \texttt{N} \\ & \mid \theta \vdash V : \tau \\ & \mid \emptyset;\emptyset : \tau \longrightarrow \texttt{JT} \\ & \mid \emptyset;\emptyset : V \longrightarrow \texttt{JV} \\ & \mid \theta;\mu : P \longrightarrow \texttt{JP} \\ \longrightarrow & \texttt{public static JT N = JV;} \\ & \texttt{JP} \end{array}$$

Fig. 5.1. Rules for Java code generation

$$
\begin{array}{ll}
\theta, \mu : A \succ B & \\
& \left| \begin{array}{l}
TLocal(A \succ B), \theta; Mlocal(A \succ B), \mu : A \stackrel{head}{\longrightarrow} HeadParams, \texttt{Jhead}, \texttt{Jreturn} \\
\theta \stackrel{params}{\longrightarrow} AddParams \\
\theta_{leaf}(A \succ B), Mlocal(A \succ B), (HeadParams, AddParams) \stackrel{decl}{\longrightarrow} \texttt{Jdecl} \\
TLocal(A \succ B), \theta; Mlocal(A \succ B), \mu : B \longrightarrow \texttt{Jbody}
\end{array} \right. \\
\longrightarrow & \texttt{Jhead Jdecl Jbody Jreturn } \}
\end{array}
$$

Fig. 5.2. Generation rules for method declaration

5.3 Bindings to generate

As I have done for typing, I will generate rules for each declaration of a structure in Popeye.

5.3.1 otype

For each object created, I need to generate five rules. The first one takes care of the actual type in Java. It is basically rewriting the Popeye type with a first capital letter. The second rule generates the main object header. The third rule generate the constructor declaration. The fourth gives the code to use when a **new** is issued. Finally, I give the code for the **super** keyword, even though it is not technically necessary. Fig 5.3 give the corresponding set of rules. In the $\stackrel{head}{\longrightarrow}$ rule, the opening bracket in the corresponding Java code finds its match in the rule given in Fig 5.2.

5.3.2 ctype

Constants are a treat. The single rule to generate is given in Fig 5.4.

The code for the object *name* is (**self** S . P)
Object *name* extends object O

Initial directive is:
otype *name* $t_1 \rightarrow t_2 \rightarrow ... \rightarrow t_n \rightarrow$ class.

$\theta, \mu : name$
$\quad | \ \theta, \mu : tref\ O \longrightarrow$ `JO`
$\quad | \ \theta, \mu : tref\ name \longrightarrow$ `JN`
$\quad | \ (S : (tref\ gate)); \theta, (S : \texttt{this}); \mu : P \longrightarrow$ `JP`
$\longrightarrow$

```
public class JN extends JO {
JP
}
```

$\theta, \mu :$ **cons** $S\ (name\ A_1\ ...\ A_n)$
$\quad | \ \theta, \mu : tref\ name \longrightarrow$ `JS`
$\quad | \ \theta, \mu : t_1 \longrightarrow$ `JT1`
$\quad | \ \theta, \mu : A_1 \longrightarrow$ `JA1`
$\quad | \ ...$
$\quad | \ \theta, \mu : t_n \longrightarrow$ `JTN`
$\quad | \ \theta, \mu : A_n \longrightarrow$ `JAN`
$\xrightarrow{head}$ (A1;...;An),

```
JS ( JT1 JA1, ..., JTN JAN ){ ,
```

$\emptyset$

$\theta, \mu :$ **new** $(name\ A_1\ ...\ A_n)\ N \backslash\ P$
$\quad | \ N \xrightarrow{typo}$ `N`
$\quad | \ \theta, \mu : tref\ name \longrightarrow$ `JS`
$\quad | \ \theta, \mu : A_1 \longrightarrow$ `JA1`
$\quad | \ ...$
$\quad | \ \theta, \mu : A_n \longrightarrow$ `JAN`
$\quad | \ (N : (tref\ name))\theta, (N : \texttt{S})\mu : P \longrightarrow$ `JP`
$\longrightarrow$

```
JS N = new JS ( JA1, ..., JAN );
JP
```

$\theta, \mu :$ **super** $S(name\ A_1\ ...\ A_n)$
$\quad | \ \theta, \mu : A_1 \longrightarrow$ `JA1`
$\quad | \ ...$
$\quad | \ \theta, \mu : A_n \longrightarrow$ `JAN`
$\longrightarrow$

```
super( JA1, ...  ,JAN);
```

Fig. 5.3. Generation rules for **otype**

Initial directive is:
ctype *objname name* t_1.

$$
\begin{array}{ll}
\theta, \mu : \textbf{pcst}\ objname\ name & \\
\quad \mid \theta, \mu : tref\ objname \longrightarrow \texttt{JN} & \\
\longrightarrow \quad \texttt{JN.NAME} &
\end{array}
$$

Fig. 5.4. Generation rules for **ctype**

5.3.3 ptype

There is no need to have any code-generation rule for private variables. The basic idea is that in the scope of an object, this is just another variable, hence we just need to name it once and for all in μ. This is not the case of constants which can very well be used across objects.

5.3.4 mtype

As I explained in Section 5.1, the rules to generate depends on the type of the method considered. Fig 5.5 gives the rules for in-methods. Fig 5.6 gives those corresponding to out-methods.

5.3.5 A note on abstract classes

When the types *aclass* and *acmd* are used instead of *class* and *cmd*, this means that an abstract class is being defined. When this happens, the rules exposed above are

Initial directive is:
mtype $name\ in\ \ (ref\ objname) \rightarrow t_2 \rightarrow ... \rightarrow t_n \rightarrow cmd.$

$\theta, \mu : name\ A_1\ ...\ A_n$
| $\theta, \mu : t_2 \longrightarrow$ `JT2`
| $\theta, \mu : A_2 \longrightarrow$ `JA2`
| ...
| $\theta, \mu : t_n \longrightarrow$ `JTN`
| $\theta, \mu : A_n \longrightarrow$ `JAN`
$\xrightarrow{head}$ (A2;...AN),
| `public void name ( JT2 JA2, ..., JTN JAN ){` ,
| $\emptyset$

$\theta, \mu : name\ A_1\ ...\ A_n$
| $\theta, \mu : A_1 \longrightarrow$ `JA1`
| ...
| $\theta, \mu : A_n \longrightarrow$ `JAN`
$\longrightarrow$ `JA1.name ( JA2, ..., JAN );`

Fig. 5.5. Generation rules for **mtype** in the case of in-methods

Initial directive is:
mtype $name\ out\ \ (ref\ \ objname) \to t_2 \to ... \to t_n \to cmd.$

$\theta, \mu : name\ A_1\ ...\ A_n$
| $\theta, \mu : t_2 \longrightarrow$ `JT2`
| $\theta, \mu : A_2 \longrightarrow$ `JA2`
| ...
| $\theta, \mu : t_n \longrightarrow$ `JTN`
| $\theta, \mu : A_n \longrightarrow$ `JAN`
$\xrightarrow{head}$ (A2;...AN),
| `public JTN name ( JT2 JA2, ..., JT(N-1) JA(N-1) ){ ,`
| `return AN;`

$\theta, \mu : name\ A_1\ ...\ A_n$
| $\theta, \mu : A_1 \longrightarrow$ `JA1`
| ...
| $\theta, \mu : A_n \longrightarrow$ `JAN`
$\longrightarrow$ `JAN = JA1.name ( JA2, ..., JA(N-1) );`

$\theta, \mu : name\ A_1\ ...\ A_n$
| $\theta, \mu : A_1 \longrightarrow$ `JA1`
| ...
| $\theta, \mu : A_n \longrightarrow$ `JAN`
$\xrightarrow{test}$ `JAN == JA1.name ( JA2, ..., JA(N-1) );`

Fig. 5.6. Generation rules for **mtype** in the case of out-methods

basically unchanged, except that the keyword `abstract` should be added in the Java declaration of the abstract object and the abstract methods. This feature is demonstrated in Section 6.2.

5.4 Java code for the switch

I can now generate the Java code corresponding to the switch example. Appendix 8.3 give the λProlog module that performs the recursive analysis outlined above. Fig 5.7 gives the code obtained (the indentations are not generated, that is, the code is correct but crammed). The Popeye environment is now fully operational, and I can move on to applying it to more challenging examples.

```
public class Switch extends Object{

    boolean sw;

    Switch(){
      sw = true;
    }

    public boolean get(){
      boolean val;
      val = sw;
      sw = val;
      return val;
    }

    public void set( boolean val ){
      boolean old;
      old = sw;
      sw = val;
    }

    public void toggle(){
      if ( true == sw ){
        this.set(false);
      }
      else{
        this.set(true);
      }
    }
 }
```

Fig. 5.7. Java code generated for the switch

Chapter 6

Examples

In this section, I put Popeye into action by giving two programming examples. Both are supposed to represent the two classical relationships between objects: inclusion and inheritance. The first example defines a FIFO linked list. The second re-builds the logical gates example given in [9]. Each time, I will give example sessions in λProlog and show the corresponding Java code. In this chapter, I will only give the pure Popeye code. Code fitted for the λProlog interpreter is pushed back to the appendix.

A word, though, on the current state of the Popeye package. I have implemented everything in λProlog for direct use with the Terzo implementation. There are a few convenient elements still missing, though.

- There is no built in support for resolving imports. That is, I must manually specify which packages should be imported to resolve a given program.

- There is no parser for the language. That is, I haven't had the time to set up a program to generate the object-dependent typing and Java-generation rule. For each object, I have written three modules. The first module, extension `.sig` gives the signature of the object in λProlog. The second module, extension `.mod` give the actual code to be fed to the interpreter. The last module, extension `.spi` (as in spinach), contains all the rules that should be generated automatically (which I brutally typed myself). For some smaller program, though, I was sloppy and wrote everything in the same module. The automation of this process should be straightforward, since I have written all the rules needed in Chapter 4 and 5. I wrote these `.spi` files mechanically.

- The Java code generated is certainly not ideally presented. It is just a big string that I copy/paste to any text editor. Having it sent to a file is easily done, be we definitely need pretty-printing work for that matter. [20] gave good ideas on what could be done to enhance that aspect.

Needless to say, writing these three things would undoubtly be the next step into developing Popeye into a fully operational programming environment. The specification for the Popeye language is centralized in the file named `popeye.sig`. The interpreter is in `interp.mod`. The typechecker can be found in `tchecker.mod`. The Java code generator is placed into `spinach.mod`. Fig 6.1 give examples of utilization.

```
Terzo> #query interpreter, switch.
?- interp nil nil ( new switch s\ toggle s >> get s O ) end.

O = pbool false;

no more solution
?-
Ctrl^C
Terzo> #query spinach, typechecker, switch.
?- spinach nil nil switch Java.

Java = ``public class Switch{ ...
```

Fig. 6.1. Example of session

6.1 FIFO queue

In this example, I will build a linked list. Very classically, I first program an object named cell that will contain an object, and two pointers to two other cells, namely the one before and after in the list. There is nothing special about this object. We provide a get/set pair of methods to access these three elements. Fig 6.2 presents the corresponding Popeye code.

More interesting is the queue object itself. I provide a method to put an object in the queue, named `enqueue`, that basically adds an element to the queue by moving the various pointers around. I also have to create a new cell to store the object, which is done through a **new** statement. Finally, we take into account the fact that the queue could be empty by using an appropriate conditionnal. The method to dequeue is a little more demanding, in the sense that in a typical imperative language, I would need a loop to solve the problem. Here, I simply explore the queue recursively through a particularly brutal method that creates a queue one head shorter to explore the initial queue down to its final element. I would also like to point out the necessity of putting all **new** statements out of the scope of a conditionnal. A given program will not go through the typechecker if it features such a configuration. The reason is that the scope of the new statement should always extend to the whole body of the clause, since creating a new element in Java is not limited to the scope of a pair of brackets! There again, we have a typical conflict between the λ-calculus and Java this time. Fig 6.3 give the code for the queue.

```
otype cell class.

ptype data      ref object.
ptype next      ref cell.
ptype pred      ref cell.

mtype get_data out      ref cell -> ref object -> cmd.
mtype set_data in       ref cell -> ref object -> cmd.
mtype get_next out      ref cell -> ref cell -> cmd.
mtype set_next in       ref cell -> ref cell -> cmd.
mtype get_pred out      ref cell -> ref cell -> cmd.
mtype set_pred in       ref cell -> ref cell -> cmd.

class cell.

self c.

private data.
private next.
private pred.

cons c cell >- empty

get_data c d >- rw data d d.
set_data c d >- rw data o d.
get_next c n >- rw next n n.
set_next c n >- rw next o n.
get_pred c p >- rw pred p p.
set_pred c p >- rw pred o p.
```

Fig. 6.2. Popeye code for the cell object

```
otype queue class.

ptype head      ref cell.

mtype set_head in        ref queue -> ref cell -> cmd.
mtype enqueue  in        ref queue -> ref object -> cmd.
mtype dequeue  out       ref queue -> ref object -> cmd.

class queue.

self q.

private hd.

set_head q h >- rw hd oldh h.

enqueue q obj >- new cell c\
                 rw hd h h      >>
                 set_next c h   >>
                 set_data c obj >>
                 ( (h == null)  +> empty
                                |> set_pred h c ) >>
                 rw hd old c.

dequeue q obj >- rw hd c c       >>
                 new queue newq\
                 ( (c == null)   +> (obj == null)
                                 |> ( get_next c t >>
                                      ( (t == null) +> ( get_data c obj >>
                                                         get_pred c p   >>
                                                         (n == null)    >>
                                                         ( (p == null) +>
                                                           rw hd old n |>
                                                     ( cast n (tref cell) >>
                                                       set_next p n        )
                                                         )
                                                       )
                                                    |> ( set_head newq t  >>
                                                         dequeue newq obj )))).
```

Fig. 6.3. Popeye code for the queue object

I can then put my program to the test in the interpreter. The following example session shows how:

```
Terzo> #query interpreter, cell, queue.
?- interp nil nil ( new queue q\ enqueue q (pint 1) >>
                    enqueue q (pint 2) >> enqueue q (pint 3) >>
                    dequeue q What     >> dequeue q When      ) end.

What = pint 1
When = pint 2;

No more solutions.
```

Great! It works! And it appears that I don't have any nasty surprises like two possible outcomes of my program! My next move is to generate the corresponding Java object. First, I check that my object passes the typechecker. Then I generate the code. Remember that the typechecking rules are automatically generated. In the end, I should just have to feed my Popeye program to a transformation program that will merge these two steps and eliminate the need to write the `.spi` file!

```
Terzo> #query spinach, typechecker, cell, queue, cellspi, queuespi.
?- tcheck nil queue nil tmod nil.
(.../...)

Solved.
?- spinach nil nil queue Java.

Java = .....
```

Of course, I do the same thing for my cell object. Fig 6.4 and 6.5 give the corresponding Java programs. Hoping to have made a good impression with this first example, I then move on to applying this method to the example presented in [9].

```
public class Cell extends Object{
    Object data;
    Cell next;
    Cell pred;

    Cell(){
    }

    public Object get_data(){
        Object d;

        d = data;
        data = d;
        return d;
    }

    public void set_data( Object d ){
        Object old;
        old = data;
        data = d;
    }

    public Cell get_next(){
        Cell p;
        p = next;
        next = p;
        return p;
    }

    public void set_next( Cell p ){
        Cell old;
        old = next;
        next = p;
    }

    public Cell get_pred(){
        Cell p;
        p = pred;
        pred = p;
        return p;
    }

    public void set_pred( Cell p ){
        Cell old;
        old = pred;
        pred = p;
    }
}
```

Fig. 6.4. Java code generated for the Cell object

```
public class Queue extends Object{

    Cell head;

    Queue(){    }

    public void set_head( Cell h ){
        Cell oldH;

        oldH = head; head = h;
    }

    public void enqueue( Object obj ){
        Cell oldH; Cell h;

        Cell c = new Cell();
        h = head; head = h;
        c.set_next(h);
        c.set_data(obj);

        if ( h == null ){ }
        else{ h.set_pred(c); }
        oldH = head; head = c;
    }

    public Object dequeue(){
        Cell n; Object old; Cell p; Cell c; Cell t; Object obj;

        c = head; head = c;
        Queue newQ = new Queue();

        if ( c == null ){ obj = null; }
        else{
            t=c.get_next();
            if ( t == null ){ obj=c.get_data();
                              p=c.get_pred();
                              if ( p == null ){ old = head;
                                                head = null; }
                              else{ n = null;
                                    p.set_next(n); }
            }
            else{ newQ.set_head(t);
                  obj=newQ.dequeue(); }
        }
        return obj;
    }
}
```

Fig. 6.5.   Java code generated for the Queue object

6.2 Logical gates

As simple as it may seem, building an object architecture to represent logical gates interacting with each others basically implies using all the features offered by an object programming environmment. In particular, we will express abstract methods. The first step is of course to lay down a class hierarchy. Fig 6.6 represents this organisation. Before I start writing down the diverse objects in Popeye, I would like to point out the fact that it is necessary to think about the interfaces of these objects first. IDLs (Interface Definition Languages) could be used for this purpose. Also, although it is not visible in the class hierarchy, my implementation makes an intensive use of yet another object named `Poplist`. The implementation for this object is examined in the next section. It is sufficient to understand its use to represent it as Prolog-style list, with a head and a queue which can be extracted separately.

First, I write down the object for the wire. Fig 6.7 give the corresponding program. The real difficulty was to implement the notify function. This function is in charge of triggering the recomputing of states for gates that are tied to the output of the considered wire. Since there might be more than one of these gates, I typically would have inserted a loop in Java to take care of this. The restrictions for Popeye force me into using recursion, in very much the same way as what I did for the first example. Then, I need to write down the base class for my gates. Fig 6.8 gives the corresponding code. The common feature of all gates is an output wire and a current state. Apart from the corresponding get/set methods, I define the alert method that triggers the computation of the state of the gate. Notice that one of the method declared in the header is an

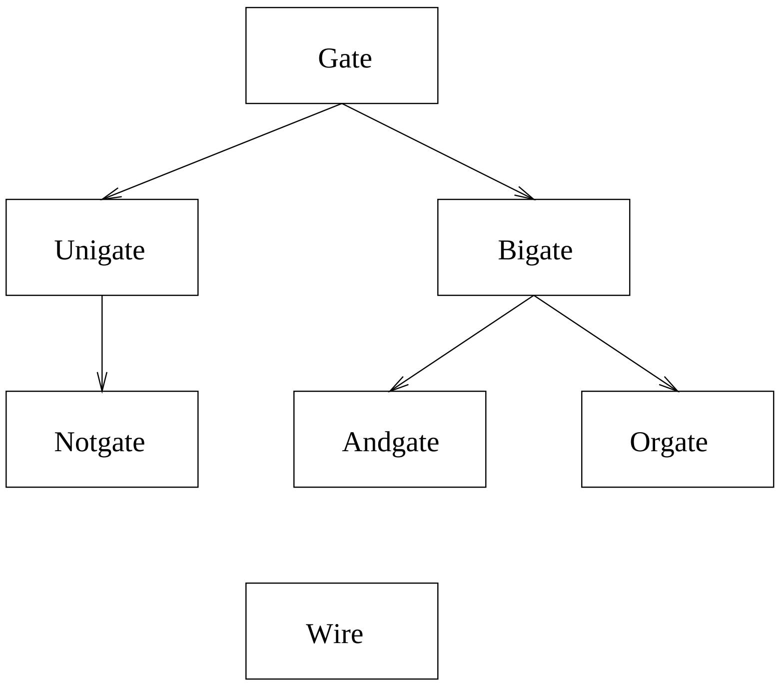

Fig. 6.6. Class diagram for Logical Gates

```
otype wire class.

ptype state boolean.
ptype queue poplist.

ctype on  boolean.
ctype off boolean.

mtype get_signal out     ref wire -> boolean -> cmd.
mtype set_signal in      ref wire -> boolean -> cmd.
mtype add_gate   in      ref wire -> ref gate -> cmd.
mtype del_gate   in      ref wire -> ref gate -> cmd.
mtype notify     in      ref wire -> ref poplist -> cmd.

class wire.

self w.

private wire state. private wire queue.

static wire on  (pbool true). static wire off (pbool false).

cons w (wire) >- new poplist l\
                 rw queue oldq l >>
                 rw state olds (pcst wire off).

get_signal w s >- rw state s s.

set_signal w s >- rw state olds s    >>
                  rw queue outs outs >>
                  notify w outs.

add_gate w g >- rw queue outs outs >>
                popbuild outs gate.

del_gate w g >- rw queue outs outs >>
                popdelete outs gate.

notify w outs >- new poplist outsleft\
                 popisnil outs (pbool true) +>
                 empty                      |>
                 pophead outs g             >>
                 cast g (tref gate)         >>
                 alert g                    >>
                 poptail outs outsleft      >>
                 notify wire outsleft.
```

Fig. 6.7.   Code for the wire object

abstract method (which can be seen at the `acmd`). As a consequence, I define the whole object as abstract (notice the `aclass`). The next step in my architecture is to write the code for each type of gate. These two abstract classes define the unigate and bigate objects whose respective code is given in Fig 6.9 and 6.10. Notice that these are abstract classes again since they lack the evaluation function. Once we are through with these, we can write the code for each specific gates. Namely, the not-gate, the and-gate and the or-gate. Fig 6.11, 6.12 and give the code for a not-gate and a and-gate. As one might expect, they implement the abstract method thus giving the rules to evaluate the state of each gate in regards to its inputs.

This looks rather easy when stated in such a casual way, but each time we write a method, special care must be taken in order to satisfy the type-checker. Consider, for example, the notify function in the wire object. As far as the interpreter is concerned, the following command could be used without any further problem:

```
notify w outs >- new poplist outsleft\
                 popisnil outs (pbool true) >>
                 pophead outs g             >>
                 alert g                    >>
                 poptail outs outsleft      >>
                 notify wire outsleft.
```

Of course, as far as the typechecker is concerned, this is unacceptable for two reason. First, the `popisnil` function is used as a test, hence the typechecker will not allow it out of a conditionnal. Next, the `g` variable is not typed correctly when we extract it out of the list. It is considered a generic object. Hence, a cast is needed (a common problem in OO languages that, of course, re-appears here). This justifies the final look

```
gate extends object.

otype gate     ref wire -> aclass.

ptype outw  ref wire.
ptype state boolean.

mtype get_output out      ref gate -> ref wire -> cmd.
mtype set_output in       ref gate -> ref wire -> cmd.
mtype get_state  out      ref gate -> boolean -> cmd.
mtype set_state  in       ref gate -> boolean -> cmd.
mtype alert      in       ref gate -> cmd.

mtype computestate out    ref gate -> boolean -> acmd.

class gate.

self gt.

private gate outw.
private gate state.

cons gt (gate Outwire) >- empty.

get_output gt w >- rw outw oldw w.

set_output gt w >- rw outw oldw w                  >>
                   rw state currs currs            >>
                   set_signal oldw (pcst wire off) >>
                   set_signal w currs.

get_state gt currs >- rw state currs currs.

alert gt >- computestate gt news >>
            rw outw w w          >>
            rw state olds news   >>
            set_signal w news.
```

Fig. 6.8. Code for the gate abstract class

```
unigate extends gate.

otype unigate  ref wire -> ref wire -> aclass.

ptype inw  ref wire.

mtype get_input out      ref unigate -> ref wire -> cmd.
mtype set_input in       ref unigate -> ref wire -> cmd.

class unigate.

self gt.

private unigate inw.

cons gt (unigate inwire outwire) >-
        super (gate outwire) >>
        init inw inwire      >>
        add_gate inwire gt   >>
        alert gt.

get_input gt w >- rw inw w w.

set_input gt w >- rw inw old w    >>
                  add_gate w gt   >>
                  del_gate old gt >>
                  alert gt.
```

Fig. 6.9. Code for the unigate abstract class

```
bigate extends gate.

otype bigate  ref wire -> ref wire -> ref wire -> aclass.

ptype inwa  ref wire.
ptype inwb  ref wire.

mtype get_inputa out      ref bigate -> ref wire -> cmd.
mtype get_inputb out      ref bigate -> ref wire -> cmd.
mtype set_inputa in       ref bigate -> ref wire -> cmd.
mtype set_inputb in       ref bigate -> ref wire -> cmd.

class bigate.

self gt.

private bigate inw.

cons gt (bigate inwirea inwireb outwire) >-
        super (gate outwire)  >>
        init inwa inwirea     >>
        init inwb inwireb     >>
        add_gate inwirea gt   >>
        add_gate inwireb gt   >>
        alert gt.

get_inputa gt w >- rw inwa w w.

set_inputa gt w >- rw inwa old w   >>
                   add_gate w gt   >>
                   del_gate old gt >>
                   alert gt.

get_inputb gt w >- rw inwb w w.

set_inputb gt w >- rw inwb old w   >>
                   add_gate w gt   >>
                   del_gate old gt >>
                   alert gt.
```

Fig. 6.10. Code for the bigate abstract class

```
notgate extends unigate.

otype notgate   ref wire -> ref wire -> notgate.

class notgate.

self gt.

cons gt (notgate inwa inwb outwire) >-
        super gt (unigate inwa inwb outwire).

computestate gt s >-
        get_input gt w    >>
        get_signal w curr >>
        (Curr == (pcst wire on)) +> (s == (pcst wire off))
                                 |> (s == (pcst wire on)).
```

Fig. 6.11. Code for the not-gate object

```
andgate extends bigate.

otype andgate   ref wire -> ref wire -> ref wire -> class.

class andgate.

self gt.

cons gt (andgate inwa inwb outwire) >-
        super gt (bigate inwa inwb outwire).

computestate gt s >-
        get_inputa gt wa    >>
        get_inputb gt wb    >>
        get_signal wa curra >>
        get_signal wb currb >>
        ( (curra == (pcst wire on)) +> ( (currb == (pcst wire on)) +>
                                         (s == (pcst wire on))     |>
                                         (s == (pcst wire off))    )
                                    |> (s == (pcst wire off)) ).
```

Fig. 6.12. Code for the and-gate object

of the function. The nice thing with our system is that the typechecker will reveal such bizarre feature to a potential programmer used to deal with logic.

Before I reach this point though, I can do some very interesting work in the interpreter. Basic simulation can be done. For example, I can build a notgate and play with its inputs:

```
?- new wire a\ new wire b\ new (notgate a b) g\
   set_signal a (pcst wire on) >>
   get_state g (pcst wire S)

S = off;

No more solutions.
```

But, of course, I can do far more interesting things such as checking the truth table for this gate!

```
?- new wire a\ new wire b\ new (notgate a b) g\
   set_signal a (pcst wire S) >>
   get_state g (pcst wire U)

S = on
U = off;

S = off
U = on;

No more solutions.
```

Which is kind of nice, since I thus don't have to generate all the possible scenarios! The system is exploring them for me. A word of notice, though: the current system is slow. A great deal of optimization is needed. For example, the above computation takes about five minute (!) on a 333Mhz Celeron running Linux. Also, I often get twice the same (correct) result, which comes from some hidden equivalent clauses. Notice, though, that the above system is really nice to write and test scenarios. Something that, frankly, is hellish to do in a classical imperative setting.

The same kind of simulation can be run on bigates and whole gates architectures. Simple simulations are for now pretty well done, but the kind of backtracking exposed above are for now way too costly. They have a tendency to consume all the ressources available to the system. I don't take this last fact *too* seriously. I am pretty sure that a special dedicated compiler for Popeye would make the delays acceptable and the demand for resources sustainable for most systems. After all, the system used, Terzo, is itself built on top of SML, which is already implemented in C. We have added costly features on top of that, so it is no surprise to get something so slow.

The cherry on the cake is now to generate the corresponding Java code for all these things. I give the obtained code in Fig 6.13, 6.14, 6.15, 6.16, 6.17 and 6.16.
I will now move on to demonstrate an interesting flexibility in the Popeye approach to programming. I will look at the implementation of the `Poplist` object that was used throughout this code.

6.3 The hacker's corner: using elegant λProlog programs in Popeye

In this section, I will show how to write a λProlog program that is a correct Popeye module, but makes all its computations using λProlog clauses. The best example where the programmer would give anything to be able to write his program in Prolog and not in a C-style language is the handling of lists. It is a classical exercise for a beginner

```
public class Wire extends Object{
    boolean state;
    PopList queue;

    public static boolean ON = true; public static boolean OFF = false;

    Wire(){
        PopList l = new PopList();
        queue = l; state = OFF;
    }

    public boolean get_signal(){
        boolean signal;
        signal = state; state = signal;
        return signal;
    }

    public void set_signal( boolean signal ){
        PopList outS;
        boolean oldS;
        oldS = state; state = signal;
        outS = queue; queue = outS;
        this.notify(outS);
    }

    public void add_gate( Gate gate ){
        PopList outS;
        outS = queue; queue = outS;
        outS.popbuild(gate);
    }

    public void del_gate( Gate gate ){
        PopList outS;
        outS = queue; queue = outS;
        outS.popdelete(gate);
    }

    public void notify( PopList outS ){
        Gate gate;

        PopList outSLeft = new PopList();
        if ( outS.popisnil() == true ){}
        else{ gate = outS.pophead();
              gate.alert();
              outS.poptail(outSLeft);
              this.notify(outSLeft); }
    }
}
```

Fig. 6.13. Java code for the wire object

```
public abstract class Gate extends Object{
    Wire outW;
    boolean state;

    Gate(Wire outWire){
        outW = outWire;
    }

    public Wire get_output(){
        Wire wire;

        wire = outW; outW = wire;
        return wire;
    }

    public void set_output( Wire wire ){
        boolean currS;
        Wire oldWire;

        oldWire = outW; outW = wire;
        currS = state; state = currS;
        oldWire.set_signal(Wire.OFF);
        wire.set_signal(currS);
    }

    public boolean get_state(){
        boolean currS;

        currS = state; state = currS;
        return currS;
    }

    public void alert(){
        boolean oldS;
        Wire wire;
        boolean newS;

        newS=this.computestate();
        wire = outW; outW = wire;
        oldS = state; state = newS;
        wire.set_signal(newS);
    }

    public abstract boolean computestate();
}
```

Fig. 6.14. Java code for the gate abstract class

```
public abstract class Unigate extends Gate{

    Wire inW;

    Unigate(Wire inWire, Wire outWire){
        super(outWire);
        inW = inWire;
        inWire.add_gate(this);
        this.alert();
    }

    public Unigate get_input(){
        Wire wire;

        wire = inW;
        inW = wire;
        return wire;
    }

    public void set_input( Wire wire ){
        Wire old;

        old = inW;
        inW = wire;
        wire.add_gate(this);
        old.del_gate(this);
        this.alert();
    }
}
```

Fig. 6.15. Java code for the unigate abstract class

```
public abstract class Bigate extends Gate{

    Wire inWA;
    Wire inWB;

    Bigate(Wire inWireA, Wire inWireB, Wire outWire){
        super(outWire);
        inWA = inWireA;
        inWB = inWireB;
        inWireA.add_gate(this);
        inWireB.add_gate(this);
        this.alert();
    }

    public Bigate get_inputa(){
        Wire wire;

        wire = inWA; inWA = wire;
        return wire;
    }

    public void set_inputa( Wire wire ){
        Wire old;

        old = inWA; inWA = wire;
        wire.add_gate(this);
        old.del_gate(this);
        this.alert();
    }

    public Bigate get_inputb(){
        Wire wire;

        wire = inWB; inWB = wire;
        return wire;
    }

    public void set_inputb( Wire wire ){
        Wire old;

        old = inWB; inWB = wire;
        wire.add_gate(this);
        old.del_gate(this);
        this.alert();
    }
}
```

Fig. 6.16. Java code for the bigate abstract class

```java
public class Notgate extends Unigate{

    Notgate(Wire inWire, Wire outWire){
        super(inWire, outWire);
    }

    public boolean computestate(){
        boolean curr;
        boolean state;
        Wire wire;
        wire=this.get_input();
        curr=wire.get_signal();
        if ( curr == Wire.ON ){
            state = Wire.OFF;
        }
        else{
            state = Wire.ON;
        }
        return state;
    }
}
```

Fig. 6.17. Java code for the not-gate object

```
public class Andgate extends Bigate{

    Andgate(Wire inWireA, Wire inWireB, Wire outWire){
        super(inWireA, inWireB, outWire);
    }

    public abstract boolean computestate(){
        boolean currB;
        boolean currA;
        boolean state;

        Wire wireB;
        Wire wireA;

        wireA=this.get_inputa();
        wireB=this.get_inputb();

        currA=wireA.get_signal();
        currB=wireB.get_signal();

        if ( currA == Wire.ON ){
            if ( currB == Wire.ON ){ state = Wire.ON; }
            else{ state = Wire.OFF; }
        }
        else{ state = Wire.OFF; }

        return state;
    }
}
```

Fig. 6.18. Java code for the and-gate object

in programming language to have to implement a list. When the `reverse` method is reached, the headache usually begins. Such classical structure have been fully studied in both paradigms. A great gain of time could then be achieved if Popeye could be able to bridge these two implementations. Suppose a Prolog programmer wants to use its code for manipulating lists. I will make this possible by introducing an intermediary Popeye program that will play an interface role between the Popeye code and the Prolog code. Of course, since the implementation is not in Popeye, it is not possible to generate the corresponding Java code. But, on the other hand, it is reasonnable to think that some Java code achieving similar result has already been created. By writing (or importing) a Java object that matches the interface of my Popeye program, I can thus write programs in Popeye that use my list and that will be directly useable in Java.

Let me take for example the infamous `reverse` function. In λProlog, it can be written as the following very elegant clause:

```
reverse L K :- pi rv\
(
 rv nil K,
 pi X\ pi N\ pi M\ rv (X::N) M :- rv N (X::M)
)
=> rv L nil.
```

The first step into making this compatible is to write this clause in a CPS form.

```
reverse_CPS L K G :- pi rv\
(
 rv nil K G :- G,
 pi X\ pi N\ pi M\ rv (X::N) M G :- rv N (X::M) G
)
=> rv L nil G.
```

Then, define a `poplist` Popeye object in λProlog that contains as a private variable the manipulated λProlog list.

```
type poplist     class.
type lplist      ref.
type val         list A -> ref     % A useful maping
type popreverse  ref -> cmd.

% Code to initialize the object ...
popreverse L >- rw lplist (val LPL) (val LPL) >>
                lamb( reverse_CPS LPL LPT )  >>
                rw lplist (val LPO) (val LPT)
% Additionnal code ...
```

The code I write as the body of the `popreverse` method just replaces the `lplist` private variable with the inverted list. To call upon the λProlog clause through the interpreter, I pass through the special command `lamb` that basically indicates to the interpreter that an outside procedure was used. And this is all I need to do in order to use my `reverse` in Popeye! The next step, of course, is to write the corresponding Java object, something that any hacker will gladly do, while higher order work can be done in logic. The code for the complete `poplist` object is given in Appendix 8.4.4.

Using these simple three examples, I think I have given a good starting point for any potential user of the Popeye system. In my idea, though, this is just a start. Bigger things are yet to come.

Chapter 7

Conclusion and insights

I have now completed the presentation of my work. There are many paths that could be followed from this point, leading to many important results. In this chapter, I try to enumerate them and evaluate the scope of the potential results. This section could also be useful for upcoming students that would be interested into pursuing this project further.

7.1 Language issues

The first point that needs a lot of refining is the language itself. In its present form, it has features that are closely related to its logical semantics. This had some appreciable advantages, the most important one being the easy translation to logic and the clear writing of an interpreter in λProlog. The problem, though, is that if we want to expand the pool of potential users of the system, it would be a better idea to make Popeye programs closer to a classical imperative syntax. For example, consider the **new** statement. This bizarre backslash is sure to confuse more than one programmer unfamiliar with the λ-calculus.

In the same idea, the next aspect that would require particular attention is the implementation of loops in the language. The core reason for not implementing loops in

Popeye comes from the problem of loop variables. Consider the following possible body for a loop, where `i` is a counter:

```
( inc i >> ... )
```

If I want to model the multiple invocation of such a sequence of instructions in logic, then I cannot leave `i` unprotected. If this is not done, then `i` will unify itself with its first value and stick to this unique value afterwards. Of course, I don't want this to happen, just like I did not want my local variable to work only once! I used a universal quantifier to guard local variable. I need to guard these "loop" variables with a similar technique. Of course, the next problem coming in line is the determination of which variable is a loop variable and which is not: it seems rather pitiful to ask for the programmer to manually specify these variables each time a loop is written. Thus, here, I will require some kind of flow analysis, something that, in my opinion, is not central to this dissertation.

Finally, there is one major element of OO programming that has not been treated in this thesis. Most modern systems are now event-drive. That basically means that objects communicate with one another through messages that are exchanged through a common, yet hidden, asynchronous mechanism. This is deeper than it may looks, as I need logic to address this problem completely.

7.2 Logical issues

The presentation of Linear Logic I worked upon, also referred to as the Lolli presentation, is a rather smallish subset of the global system presented by Girard. Also, one can notice that I use connectives for a very specific purpose. For example, I intensively used the universal quantifier to express the idea of scope (as outlined in [12]). But, among all the connectives Girard presented, I have overlooked the most intriguing of all: the par ($\parr$). Miller, in [14], gave a tangible meaning to this connective by showing how it can be used to model asynchronous threads. He describes FORUM, a presentation of Linear Logic into which one can describe what is supposed to happen when two entities (or threads) are put in the presence of each other. So, when I write:

$$A \parr B$$

the intended meaning is that the environment I am considering contains these two entities floating around. But, then, I can write clauses of the following form:

$$A \parr B \circ\!- C$$

This describes that when A and B are found together, they both disappear to be replaced by C. The hunch I have with Popeye is that this is the key to proving Java multi-threaded programs correct.

Consider a `button` object. The most common message triggered is, of course, the classical `click` event. Ideally, I could write in the interpreter something like this:

```
(new applet a\ new button b\ add a b >> start a)
|
(click b)
```

The | stands for the par symbol. This would basically specify that I create an applet with a button, which I have clicked. But then come harder problems. Even the simple session above is problematic: if we follow the way I specified the **new** statement in logic, the name `b` is out of the scope of the `click`.

The biggest problem, though, revolves more around how to write the specifications themselves rigorously. In Java, for instance, the code to execute for a given event is to be written in the scope of an event listener object. So, if I construct a button with a given listener, then somewhere in the constructor, a listener must be launched.

```
class button. self b.
.../...
cons b button >- new bActionListener b al\
                 start al.
.../...
```

Now, I need to write the specification for forwarding the control to the code written in the listener whenever the button is clicked. Instinctively, I would write something like this:

```
?(bActionListener b Al | (click b >> K) >-
actionPerformed Al     | K.
```

The ? is the Linear Logic "why not?", which means that the action listener traverses the clause untouched. The hard problem is then to be able to add more threads to this clause. The way Linear Logic is written, there can only be exactly the actors described in the clause to have the clause happening. In the immense majority of cases, this is not what happens. There will be various threads around that have nothing to do with this clause and thus need to be untouched by it. There is also the very difficult problem of deciding what to do when more than one of these clauses could be fulfilled. Which one do I satisfy first? Could I decide to satisfy both of them? So many questions need an answer!

So, if I want to specify Java threads properly, I have to go as far as to tamper with the structure of the logic itself. A stronger mathematician than I is needed here. The result, though, are sure to be fascinating. It is then possible to simulate asynchronous program in logic, tracking all possible scenarios and thus those that could go wrong. For example, consider a set of elevators used in the same building. This classical example must ensure maximum efficiency while basic safety property must be fulfilled. Typically, at all cost, no door must open without an elevator behind it. Using our system, we could generate the possible scenarios very quickly in the interpreter.

7.3 Implementation issues

Another interesting question when one looks at the code for the interpreter code is the issue of whether Linear Logic is 100% needed. It is quite clear that on a description level, Linear Logic made the description of my system crystal clear. But it is also true that I have implicitly shown that it is not necessary for writing Popeye in logic. A decent manipulation of lists representing the various environment is enough to represent Popeye computation in λProlog.

On the other hand, I cannot separate Popeye from Linear Logic completely. The interpreter for Popeye programs was written with the constant thought that it expressed a rigorous theoretical background. In short, Popeye would only be yet another hack without this solid ground to stand on. In this, Linear Logic is fundamental.

Was λProlog a good choice? In my opinion, it is a great tool for designing and manipulating a syntax. The definition of syntactic categories is a breeze, which consequently speeds up the writing of an interpreter for the language. For this reason, and at this early stage of design, I consider it the ideal tool. It is a blurred topic to try to define the kind of environment suitable for the next stages in the development of Popeye. For example, one of the element absolutely needed is a parser for the language. The only problem, though, is the absence of any parser generator in logic programming. In [11], Liang proposed a parser generator named λYacc that might solve this issue, but that is still in early development. The temptation, then, would be to switch to other contexts where such tools are widely available. I think this is premature. There are many obscure points that remain in the logic that need to be solved before we are confident enough to build a dedicated system.

Finally, let's not forget that there is also the need to write a program that generates the equivalent of the `.spi` files. This is probably easier in logic, since all we have to do is generate a module.

7.4 Insights

These implementations matters, though, are only the first steps to what this work could lead us to. One of the real goal for Popeye is to be able to feed Popeye programs in a theorem prover. Backtracking is nice, but the problem of induction is absolutely not handled by this method.

This is a huge problem, because what we typically would like to do is examine the behavior of a loop and infer whether it results in the output we expected out of it. This implies the use of proofs that do not explore that totality of execution traces, since the number of such traces is typically infinite. Consider, for example, a program that given a number n, generates the set of prime numbers smaller than n. Since n can be any natural number, there is no other proof but by induction. If we come back to the elevator example, then it is clear that such reasoning will definitely come handy.

In the end, it should be possible to build small lemmas on some representative programs and loops. Once this is done, these lemmas could be used in larger proofs, or re-used on similar programs. With a big enough set of such results, most programs could be proved correct rapidly. At this point, Popeye becomes a very powerful tool for writing programs with a high level of confidence.

The next step would be to expand the expressiveness of the language until we reach a point where bringing an existing program in, say, Java, can be done without too much effort. At this point, amazing things could happen, where we would be able to extract a program, analyze it in the context of Popeye, isolate errors and inefficiencies, and output a corrected version that would have been proven correct. This is still very far away, and much work needs to be done before we can hope approaching such capabilities.

7.5 Final words

I am nearly finished with this presentation of the relationships between Linear Logic and OO programming languages. I would like, though, to finish with a little personal reflexion.

As I developed this whole construction, many things intrigued me. But the most bizarre feeling I had was to be convinced that the parallels noticed between these two theories were too easily drawn to be pure strokes of luck. In fact, I have developed the most intimate conviction that the initial developers of the approach of programming known as object programming were indeed thinking in terms of logic. First, they fragmented their program in entities bound together by a high data cohesion. These entities can easily be understood as facts. A set of facts, or objects, defines a program. Writing the interactions between these entities then became a matter of writing a set of interaction rules triggered by a given configuration of the system. Schach, in [18] represented this fact as the classical implication:

$$Set\ of\ initial\ conditions \Rightarrow Operations$$

But let us not forget that implication is before and for all defined classicaly as follows:

$$A \Rightarrow B \quad iff \quad (not\ A)\ or\ B$$

That is, after all, a logical formula...

Here, maybe, lies the hidden relation that explains why the parallelism works so well. This means that those people working in Java, C or any other imperative setting, known to have a rather harsh opinion of other paradigms, are in fact using ideas deeply rooted in Logic.

Could irony be the only real meaning of a Popeye program?

Chapter 8

Appendix

Instead of listing all the code, I made it available at the following URL:

`http://www.cse.psu.edu/~dale/betis/`

8.1 Appendix A - Popeye Interpreter

The following files define the interpreter:

- `popeye.sig`: Signature for the Popeye language.
- `interp.mod`: Interpreter module.

8.2 Appendix B - Type-inferer

- `tchecker.mod`: Type-checker

8.3 Appendix C - Java code generator

- `spinach.mod`: Java code generator

8.4 Appendix D - Examples

8.4.1 Switch example

- `switch.pop.mod`: Switch object

8.4.2 FIFO example

- `cell.*`: Cell object
- `queue.*`: Queue object

8.4.3 Logical gates example

- `wire.*`: Wire object

- `gate.*`: Gate object
- `unigate.*`: Unigate object
- `bigate.*`: Bigate object
- `notgate.*`: Notgate object
- `andgate.*`: Andgate object
- `orgate.*`: Orgate object

8.4.4 List example

- `list.*`: List object

8.5 Appendix E - Additional references

Many readings should have their place in this work. This is why I dedicate this section to them. [15] was a starting point in this work, and raised many questions in my mind since it was the first time I faced such things. I looked into [16] to understand logic programming in its theory and [1] helped me focus on implementation issues. FORUM expressive power was also demonstrated in the massive PhD thesis from Manuel M.T. Chakravarty ([3]). Finally, in the course of this work, a very worrysome paper (for me) came out. [2] had the following familiar title: "Translating Linear Logic Programming Language into Java". As it turned out, the translation was more an implementation, and I could push forward in my direction. Finally, the best starting point as far as logic in computation is concerned is probably Gallier's book ([6]).

References

[1] Hassan Ait-Kaci. *Warren's Abstract Machine.* MIT Press, 1991.

[2] Mutsunori Banbara and Naoyuki Tamura. Translating linear logic programming language into java. In *Proceedigns of the 10th Exposition and Symposium on Industrial Applications of Prolog*, pages 56–63, October 1997.

[3] Manuel M.T. Chakravarty. *On the Massively Parallel Execution of Declarative Programs.* PhD thesis, Berlin University, February 1997.

[4] Jawahar Lal Chirimar. *Proof theoretic approach to specification languages.* PhD thesis, University of Pennsylvania, February 1995.

[5] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[6] Jean H. Gallier. *Logic for Computer Science.* Harper and Row, 1986.

[7] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[8] Joshua S. Hodas. Lolli: An extension of λprolog with linear logic context management. In Dale Miller, editor, *Proceedings of the 1992 workshop on the lambdaProlog programming language*, Philadelphia PA, Summer 1992.

[9] Joshua S. Hodas and Dale Miller. Representing objects in a logic programming language with scoping constructs. In David H.D. Warren and Peter Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 511–526, Jerusalem, June 1990. M.I.T. Press.

[10] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, May 1994.

[11] Chuck Liang. A formulation of deterministic bottom-up parsing and parser generation in logic programming. August 1999.

[12] Dale Miller. Lexical scoping as universal quantification. In *Sixth International Logic Programming Conference*, pages 268–283, Lisbon, Portugal, June 1989. MIT Press.

[13] Dale Miller. Abstractions in logic programming. In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, pages 329–359. Academic Press, 1990.

[14] Dale Miller. Forum: A multiple-conclusion specification logic. *Theoretical Computer Science*, 1996.

[15] Dale Miller. *Logic of Computation*, volume 165 of *Nato ASI Series*, chapter Sequent Calculus and the Specification of Computation, pages 399–444. Springer, 1999.

[16] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[17] G. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(1):125–159, 1976.

[18] Stephen R. Schach. *Software Engineering with Java*. IRWIN Book Team, 1997.

[19] Paul Tarau. Program transformations and WAM-support for the compilation of definite metaprograms. In *Logic Programming: Proceedings of the First and Second Russian Conferences on Logic Programming*, number 592 in LNAI, pages 462–473. Springer-Verlag, 1992.

[20] Philip Wadler. A prettier printer. March 1998.