MASTER IN COMPUTER SCIENCE

Towards a generic static analyser for Java
a compiler and a simple analyser for two sub-languages of Java

Hayez, Cecile; Hendrickx, Patrick

Award date:
2000

FUNDP
INSTITUT D'INFORMATIQUE

Towards a Generic Static Analyser for Java : A Compiler and a Simple Analyser for Two Sub-Languages of Java.

Cécile Hayez
Patrick Hendrickx

RUE GRANDGAGNAGE, 21 • B - 5000 NAMUR (BELGIUM)

Abstract:

This work has been done in the framework of a large project on abstract interpretation of Java. The aim of this project is to develop a compiler and a static analyser for Java, which allows some verifications and optimisations of the analysed programs. Isabelle Pollet has already defined the abstract syntax used for the analysis: the LAS (Labelled Abstract Syntax). Here, we have studied a sub-language of the Java language, the Vas-T'y-Frotte. In the framework of this project, we have created a compiler composed of a parser and a type checker for this sub-language. The type checker checks all the types of the program, and translates the program into its abstract form. Doing this, it creates all the objects corresponding to the LAS. As the subset of Java corresponding to the LAS has much more constraints than the VTF (Vas-T'y-Frotte), we had to define functions to translate the VTF program into an equivalent program verifying all the constraints of the LAS. To create the parser, we used the Java Compiler Compiler (JavaCC). We have coded the objects of the LAS and the compiler in Java. We did not have the time to create an analyser for this large sub-language, so we have created an analyser for a simple subset of Java, as a separate work. This analyser has been written in CaML. It takes a CaML representation of a Java program into its abstract form. We supposed that the phases of parsing, translating to the very simple subset of Java, compiling have already been performed. The analyser implements a multivariant algorithm, to create all the possible states of the program.

Résumé:

Cette thèse a été effectuée dans le cadre d'un projet plus vaste en interprétation abstraite. Le but de ce projet est de développer un compilateur et un analyseur statique pour Java, qui permettent l'optimisation des programmes analysés. Isabelle Pollet a déjà défini la syntaxe abstraite utilisée pour l'analyse: le LAS (syntaxe abstraite labellisée). Ici, nous avons étudié un sous-langage de Java appelé Vas-T'y-Frotte. Dans le cadre de ce projet, nous avons créé un compilateur composé d'un parseur et d'un vérificateur de types pour ce sous-langage. Le vérificateur de types vérifie tous les types du programme, et traduit le programme sous sa forme abstraite. Faisant cela, il crée tous les objets correspondant au LAS. Comme le sous-ensemble de Java correspondant au LAS a beaucoup plus de contraintes que Vas-T'y-Frotte, nous avons du définir des fonctions pour traduire un programme en VTF en un programme équivalent, mais qui respecte les contraintes de LAS. Afin de créer le parseur, nous avons utilisé le Java Compiler Compiler (JavaCC). Nous avons implémenté les objets de LAS et le compilateur en Java. Nous n'avions pas le temps de créer un analyseur pour ce grand sous-langage, nous avons donc créé un analyseur pour un sous-ensemble plus réduit de Java, en tant que travail séparé. Cet analyseur a été implémenté en CaML. Il prend une représentation en CaML d'un programme Java sous sa forme abstraite. Nous supposons que les phases de parsing, de traduction dans le sous-ensemble de Java et de compilation ont déjà été effectuées. L'analyseur implémente un algorithme univariant, pour créer tous les états du programme.

Acknowledgement:

We specially wish to thank Professor Agostino Corstesi, Professor Baudouin Le Charlier and Isabelle Pollet for the patience they have had, for the time they have spent to help us during those months of hard labour. We also wish to thank them for the knowledge they have shared with us and for all the work they have delivered to set this project up.

We wish to thank Karl Noben for the long hours of wrestling together with all the abstract notions we have encountered doing this work.

We wish to thank Gysèle Henrard for all the facilities she gave us, concerning the organisation of our internship in Venice – Italy.

We would finally like to thank Simone from Ottignies for the diverting hours we have spent and for the ambiance she created into our cottage.

TABLE OF CONTENTS

1. INTRODUCTION

Dear reader, in this introduction, we would like to talk about the project we are working on. In order to do that we first explain the reasons why we are analysing the Java language, we then introduce the project and we finally explain the structure of the thesis.

1.1. Java

Java is a simple object-oriented, platform-independent, multi-threaded, general-purpose programming environment (*[MCZQ96]*). It is best for creating applets and applications for the Internet and any other complex, distributed network. An ***H***yper ***T***ext ***T***ransfer ***P***rotocol ***server*** can send a Java program to a client. This client can easily execute that program. This execution can seem very simple. Thus for example, the executable code on a PC is not at all the same as that executable code on a Macintosh. Of course, we could invent a way of communicating between the PC and the Mach applications, but this would be less evolutionary.

Java goes well beyond this domain to provide a powerful general-purpose programming language suitable for building a variety of applications. Java is described as having the following features:

Simple: Java was designed to be like ***C++*** for easy learning.
Robust: Java works hard to check for problems at compile and run-time.
Secure: Java code passes several tests before actually executing on the machine.
Multi-Threaded: Java multi-threading allows many simultaneous activities in one program.
Dynamic: Java takes advantage of as much object technology as possible.

But Java is, for the moment, not perfect. In fact, Java has still got lots of lacks of efficiency and some lacks of security. Some of the problems due to the lacks of security of the Java language are the following: There are no limits for the assignment of the memory of applets. The applets are free to take control of the actions delivered by the client. For example, an applet could save all the keyboard touches of the client. The applet could also take control of the web-cam or other hardware configurations of the client. A badly disposed person could take advantage of these lacks to get one's credit card number or one's password (*[BCS97]*). Like we have explained above, the Java programs are thus been freed from the problems of compatibility between the instruction sets of the various processors and operating systems. A machine language called Java Byte codes is associated with the source language Java. The source language Java is compiled in this machine language, and it is in this representation that the program is interpreted. As Java Byte Code is interpreted, the execution time of a program written in Java is 5 to 10 times superior to the execution time of a program written in C and then compiled (*[OTE]*). It would be interesting to cure these lacks. In order to accomplish that, we decided to analyse the language. And that is how our project is born.

1.2. The Project

The project we are working on is an inter-university project between three universities: The universities of Louvain La Neuve, Namur and Venice. The actual group is composed of three students, three teachers and one teaching assistant. The teaching assistant is Isabelle Pollet[1], and most of our work is based on her ***DEA***-thesis *[IPO99]* (***DEA***: ***D***iplôme d'***E***tudes ***A***pprofondies) of last year (1999). The teachers involved into this project are Agostino Cortesi[2], Baudouin Le Charlier[3] and Pascal Van Hentenryck[4]. The three students are Karl Noben and of course the two of us.

Pascal Van Hentenryck is the internship tutor of Karl Noben, the student who works on the ***graphical interface*** of the project in Louvain La Neuve. Agostino Cortesi is our internship tutor in Venice. Baudouin Le Charlier is the supervisor of the project in Namur.

The goal of the project is to create a generic analyser for the Java language. This whole project is composed of four parts. The first part is the ***DEA*** thesis of Isabelle Pollet *([IPO99])*. This thesis makes the theoretical bases for our work.

The second part of the project is the implementation of a compiler for the sub-language of Java called ***V***as-***T***'y-***F***rotte *([LC99a])*. This language and the compiler are explained later in this thesis. The compiler is decomposed into two major parts: the implementation of a parser and the implementation of what we called a type checker. The language Vas-T'y-Frotte has been written by Professor Le Charlier in March 1999, in the framework of his programming course. We have created the compiler by our own.

The third part of the project is the ***graphical interface*** that can be applied on the analysis. Karl Noben makes this part. It is designed to allow us to click on a certain point into a given program in order to visualise the actual state of the local variables, the state of the parameters and lots of other things that constitute the abstract state of a program.

For the fourth part of the project, we crated a simple static analyser for another, even smaller, subset of the Java language called ***V***ery ***S***mall ***S***ubset. We have invented this ***V***ery ***S***mall ***S***ubset in order to have a subset that would be simple enough to create a first outline of a static analyser. For latter work, we could imagine a second static analyser onto a bigger subset or even onto the actual Java language definition.

We could of course also say that there is a fifth part of the project, because some of the parts we are working on are not finished and some other students are succeeding us next year, but we will discuss this into the conclusion of the thesis.

After having explained the scoop of the project, we try to explain, in the next chapter, what exactly an abstract interpretation is.

[1] Research and Teaching Assistant at the university of Namur.
[2] Professor at the university "Ca'foscari" of Venice.
[3] Professor at the university of Namur.
[4] Professor at the university of Louvain La Neuve.

1.3. Structure of the Thesis

Chapter 2: Preliminary notions

The second chapter gathers all the basic notions and the notations we use in this work. You will also find a summary on the bases of the abstract interpretation. These bases are useful for a novice in the subject, who wants to go further in the reading of this thesis.

Chapter 3: The Compiler: a Parser and a Type Checker

The third chapter of the thesis is the chapter about the compiler we have created. This chapter is divided into six major parts. The first part is the introduction. In this part we explain what is a compiler and why we need a compiler in order to make a generic static analyser. The second section of this chapter is devoted to the presentation of the concrete syntax of the sub-language we are analysing. This sub-language is the so-called ***V***as-***T***y-***F***rotte that we have introduced into the section about the project.

The third and the fourth sections of this chapter make the link between the concrete and the abstract syntax of the to analyse program. Indeed, a concrete representation of a program has got an abstract equivalent representation. And that is the reason why we need to define an abstract syntax that can be related to the concrete syntax by a concretisation function. It is difficult to make the link directly between the concrete and the abstract syntax and that is the reason why we have created an intermediate representation between the concrete and the abstract representation. The third section is the section about the abstract syntax (***L***abelled ***A***bstract ***S***yntax *[IPO99]*) and the fourth section deals with the intermediate representation.

Once we know all the basics about a compiler, we can carry on with the implementation of the compiler. This compiler is split into two parts: the parser and the type checker. The fifth and the sixth sections of the chapter are the section about this parser and the section about the type checker.

Chapter 4: Static Analysis by Abstract Interpretation

In the fourth chapter we explain the work of a static analyser. This chapter is divided into five sections. The first section is an introducing section.

In the second section we introduce the notion of a syntax. There are two syntax's into a static analysis. The first syntax is the concrete syntax. The second syntax is the abstract syntax.

There is of course a way to transform a program, written into the concrete syntax, into an equivalent program representation of the abstract syntax. Here we have a concrete syntax written into a subset of the Java language. This subset is called ***V***ery ***S***mall ***S***ubset of Java (Hence: ***VSS***). The abstract syntax is written in CaML.

The third section of this chapter explains everything about the semantics and the transition rules between the different states. We have also incorporated a sub-section explaining the manner of proving the correctness of the transition rules.

The fourth section of the chapter is devoted to the implementation of the static analyser. In this section you can find the algorithm we have chosen, the domain we are analysing, the implementation itself and some test programs we have written.

Thus we have had to write a translator between the concrete and the abstract syntax. In order to create an easy translator for those syntax's we had to create a simplified language. This language is a ***Fortran*** based equivalent language to the ***VSS***. The goal of this simplified language is to make the ***VSS*** language easier to parse and to translate into the abstract syntax.

Chapter 5: Conclusion

This is the fifth and also the last chapter of the thesis. In this chapter you can read a conclusion of our thesis. You can also find what exactly is the advancement of our work. And some tips for later works in this domain. This chapter could for instance be interesting for the students who are following us up in our research and work.

2. PRELIMINARY NOTIONS

2.1. Definitions

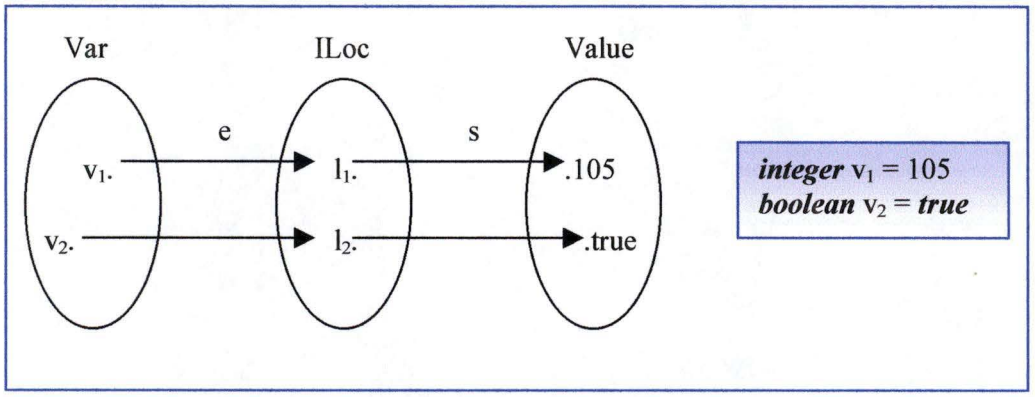
2.1.1. Store and environment

The environment and the store are the two functions that define the state of the memory.

The first function, the environment, associates a value to a variable. This value can be seen, for example, as the address of the variable in the memory. The domain of the environment consists of the list of the names of all the accessible variables: the local variables, the formal parameter names of the current method and the variable ***this***. ***null*** corresponds to a non initialised variable, or a variable with the value ***null***; ***undef*** corresponds to a non defined variable.

The store associates a ***float***, a ***boolean***, an ***integer*** -for basic types-, string, instance... to a value. With the store and the environment, you can find the instance,... of all the active variables of the program.

The store will most of the time be noted 's' (s_a for the corresponding abstract store) and the environment 'e' (e_a for the abstract environment).



2.1: Store and environment definition

2.1.2. State of a program

The state of a program, at a certain point of its execution, can be defined as all the information about the state of the memory at this point and the information needed to find the next state in the execution of the program. In general, the information contained is:

- The current statement (or a label which allows to find the statement)
- A stack containing all the information about the successive method and constructor calls. This information is needed to find the following statement when we are at the end of a method or a constructor.
- The current environment
- The current store

The state will be noted this way: < p, P, e, s >

2.1.3. Syntax and semantics

A general language is defined buy its syntax and its semantics. Those can be considered as given with the language. If you study a sub-language of an existing language, you will probably have to redefine the concrete syntax and semantics, with the constraints of the sub-language. In other cases, you will maybe want to add some information in the abstract syntax, in the aim to improve the analysis. The abstract syntax has to be redefined in that case.

The concrete syntax of a language is all the rules that define the way a program has got to be written in that language. These rules give the explanation of the 'text' of the program.

The semantics rules are rules about the way the language works. They tell what to do when you encounter one or another statement, they tell how the store, the environment are modified... They define the transitions between the states of the program.

2.1.4. Specialisation of a type

As there can be some inheritance between two types in Java, the structure of the types can be seen as a set of trees. We consider that a type t is a specialisation of another type t' if the type t' is an archetype of t. We use this notation:

$t \leq t' \Leftrightarrow$ t extends t' or $\exists$ t" tq t extends t" and $t'' \leq t'$

2.2. Notations

2.2.1. Syntax definitions

In our work, we often have to define some syntax's. In this aim, we use a representation close to the BNF (Backus-Naur Form). Here is an explanation of the syntax we use:

- Terms in bold and italic are keywords or symbols of the defined syntax.
- Terms just in italic represent non-terminal terms.
- (term)$^{?}$ represents 0 or 1 time the term.
- (term)* represents 0, 1 or several times the term.
- (term)$^{+}$ represents at least 1 time the term.
- | represents the notion 'or'

Here is a simple example, for a better comprehension of our notations:

Litterals: *firstname, name, streetname, townname* (strings), *number, pc, phone_number* (integers) , *box* (character)

somone ::= *firstname name , address , phone*

address ::= *street streetname , number* (*box*)? *; pc - townname*
| *road roadname , number* (*box*)? *; pc - townname*

phone ::= ***no_phone*** | (*phone_number*)+

Some examples of *someone* could be:

Anne Dupont, MainRoad, 2; 4345 - Florennes, no_phone
Christine Ferie, SwordStreet, 54 B ; 5000 – Namur, 081654731 071568435

This notation will be used all along our work.

2.2.2. Transition rules

A transition rule explains the passage between two successive states of the program during its execution.
In our work, we use this notation:

< initial_state > ⟶ < final_state >

Where { current_label } current_statement { following_label }

Constraints on final_state, using the initial_state and the current_statement

Example:

$< p, P, (e, s) > \longrightarrow < q, P, (e[v_1/Val(e, s, v_2)], s) >$

Where { p } ***affect*** v_1 v_2 { q }
$v_1 \in$ VarName

2.2.3. *Lists, sets...*

A set of objects will often be written { obj_1, ... obj_n }
When you have a set S, ***P***(S) represents the set of all the subsets of S.
For example:

If S = { a, b }
P(S) = { {}, {a}, {b}, {a, b} }

A stack of a list l will be noted this way: l = head::list_rest
where head is the first item of the list and list_rest is the rest of the list (and is a list itself).

2.2.4. *Notations of variables*

Concerning another notation, from now on, we use the terms 'return label', 'return environment'... to represent the label where we come back after the end of a method call (i.e. after the *return* statement), or the environment of the calling method. This environment becomes the current environment when we come back to the method at the end of the method or constructor call.

The target variable in a method call is defined like this:
A method call has the following syntax: var_1 = var_2.meth(param_l)
As var_1 is the return variable, var_2 is the target variable.

We also want to notice that when we want to speak about the local variables of a program, we will use explicitly the 'local' word. When we simply use the word 'variable', we mean a general variable, which can be a local variable or a field.

2.3. An Introduction to Abstract Interpretation

2.3.1. Aim of the Static Analysis

The abstract static analysis has got two principal aims: the optimisation and the correction of programs.

In almost every static analysis, a program execution is considered as a succession of states. It is a succession of concrete states in the concrete case, and a succession of abstract states in an abstract analysis.

The static analysis allows, for example, to give an approximation of the types of the variables at each point of a program. Instead of making an analysis of the types, you can analyse the values of the variables. If you are analysing integer variables, you can try to know, at a certain point of a program, if one of the variables is positive, negative, null, or if it can be any of them...

Keeping this in mind, you can optimise your program. If you see that, at a certain point, a variable is always positive. And you know that you have to apply a certain function on integers at that point of the program. Then you can simplify that function (this means that you can use another function that returns exactly the same results for positive values but that does not check the negative values) because lots of functions are easier to apply on positive integers than on negative integers.

You can prove the correctness of the pre- and the post-conditions of the methods of your program, studying the characteristics of the variables at the beginning and at the end of the methods.

2.3.2. Steps of a Static Analysis

In this introduction we give you an overview of what is the abstract analysis. It can seem a little bit boring for people being every day in this domain, but it certainly can do no harm to return all the way back to the basics.

The abstract analysis is made of three major parts:

- The definition of the concrete semantics
- The definition of the abstract semantics
- The derivation of a static analysis

2.3.3. The Definition of a Concrete Semantics

The execution of a program is considered as a succession of states of the program.

We first have to define, exactly, the information contained in a state of the program.

Afterwards, we have to define the transition rules between the concrete states, i.e. the semantics itself.

A program can have an infinity of different executions, and an execution can be infinite. The number of possible states of the program can be infinite.

A simple example is this one:

```
class class_name
        {
        int x = 0;

        public static void main (String args [])
                {
                while true
                        {
                        x = x + 1;
                        }
                }
        }
```

The execution of this program is infinite, and the store will be different (the variable x will take all the integer values one by one during the execution), at each passage in the loop. The program passes trough an infinity of different states.

As it is impossible to consider all the different executions of a program, in the concrete case, it is impossible to analyse the general behaviour of the program.

2.3.4. The Definition of an Abstract Semantics

To analyse the general behaviour of a program, we have to consider all the possible executions of the program. We define an abstract semantics, in order to do this.
The difference between concrete and abstract semantics is that we make some approximations to make all the sets finite, in the abstract case. The aim is to limit the number of possible states of a program. The most important set to define is the AType, i.e. the set of the abstract types. Most of time, this set is the same than the concrete set Type, or a set of subset of Type.

One way of defining all the sets of the abstract semantic is to create a function that would make the correspondence between the concrete and the abstract objects.

There are two ways to define this function:
You can define an abstraction function, which associates the corresponding abstract object to a concrete object (state, type, environment, store, and so on), or a concretisation function, which associates the corresponding set of concrete objects to an abstract object.

In fact, the two functions can be defined, but most of the time, only one is necessary.

Here is an example:

In the concrete case, a concrete store associates its value to a location of a variable.

Let's go back to the previous example:

```
class class_name
        {
        int x = 0;

        public static void main (String args [])
                {
                while true
                        {
                        1  x = x + 1;
                        }
                }
        }
```

The concrete store always changes during the execution, at the label *1*, associating all the possible natural numbers to the variable 'x'. As there is an infinity of different concrete stores (which belongs to the concrete state), there is an infinity of states.

Let's define an abstract domain that is finite, associated to the variables. It could be, for example, the set {-, 0, +}, corresponding to the negative/null/positive values of the variables.

The concretisation function is:

$C_C(-) = \{v \mid v < 0\}$
$C_C(0) = \{v \mid v = 0\}$
$C_C(+) = \{v \mid v > 0\}$

If we note s a store and s_a an abstract store, the concretisation function (C_C) for the stores is:

$$C_C(s_a) = \{s \mid \forall l \in ILoc : s(l) \in C_C(s_a(l))$$

The abstract store, in the abstract case, will associate:

the value 0 the first time
the value + afterwards
} to the location of 'x', at the label *1*.

In this simple example we have defined, there is only one label (one statement), we can see that there are only 2 possible abstract states with the given abstract domain. The states correspond to the two possible abstract stores.

Once we have defined the function for all the objects contained in the states, all the rules of the concrete semantics have got to be translated. The created rules will be based on the abstract states instead of the concrete sates.

2.3.5. Computing the Abstract Semantics

Once we have created all the possible states of a program, we can try to draw some interesting information.

If the abstract domain is defined as being all the types of the programs, we can get, for example, all the dynamic types of the variables at a certain point of a method.

In the example we have developed before, we can conclude that, in this program, the variable 'x' is always >= 0. This information is obvious in such a simple program, but can be less evident in large programs. You can use this information to prove an invariant of the program that could say that x >= 0, or simply use it to create the invariant of the program.

In fact, the algorithm, which creates all the possible abstract states of a program, already exists. Depending on the choice of the algorithm, the result is more or less precise but the speed of execution also depends on the choice of the algorithm. Once you have all that states, you can just takes all the states corresponding to a certain point of the program.

3. COMPILER: A PARSER AND A TYPE CHECKER

3.1. Introduction

Like explained in the introductory chapter, the aim of the project, at long term, is to make a static analysis of the Java language. It is not possible to achieve such an ambitious goal directly. So, we try to do it step by step. We therefore begin the analysis by constructing a compiler. This compiler is very important for latter work. In fact the compiler is a tool that allows us to create the required structures for the static analyse. This compiler is not a common compiler like we know them. It does not generate some executable code like most of the compilers do. This particular compiler generates a tree corresponding to the abstract syntax of the language we are analysing.

It is of course not possible to make, directly, an analysis of Java, so we decide to analyse a subset of this language. This language is a subset of the Java language and is thus easier to analyse than the real Java. In fact, for an analysis of the complete Java language, we just need some more time and some more experience in the Java finesses. But the biggest part of the delivered work is, and would stay, identical for a bigger language to analyse. So, in fact, we are making a first sketch of the big and ambitious project.

Some big parts of our work are based on existing materials. These materials are the concrete syntax and the abstract syntax we are using. The concrete syntax we are using is the ***VTF*** created by Professor Le Charlier in 1999 (*[LC99a]*). This language contains all the important features of an object oriented language. The language is very much based on the Java language definition. The only thing that makes the two languages differ is that the ***VTF*** is a subset of the Java language and that the ***VTF*** thus has got a smaller definition as the Java language. The abstract syntax we decide to use as correspondence to the ***VTF*** concrete syntax is the ***L***abelled ***A***bstract ***S***yntax that has been written by Isabelle Pollet in the framework of her ***DEA*** thesis of 1999 in Namur (*[IPO99]*). This thesis has been written in order to make some theoretical basements for a static analyse of the Java language (the project we are working on right now).

This chapter is divided into five major sections. The first section is the presentation of the concrete syntax. The second section is the presentation of the abstract syntax. For people who are interested into more detailed information about those syntax's we refer to: *[LC99a]* and *[IPO99]*. After these two sections, there is a section explaining an intermediate representation of the syntax's we needed in order to pass from the concrete to the abstract syntax. And finally, we have got the two sections explaining the implementation of the compiler. This compiler is split into two parts: the parser and the type checker. The parser takes a text of a program and translates this one into the intermediate representation. The type checker takes the intermediate representation and translates this one into the abstract syntax representation.

3.2. Presentation of the Concrete Syntax

Every parser or type checker needs a language to analyse. We decide to analyse the language called ***VTF***. In this section we try to explain what exactly the syntax of ***VTF*** is, what this syntax means and why we take this particular syntax.

3.2.1. The VTF Basic Types

The ***VTF*** language contains three basic types: booleans (***boolean***), integers (***int***) and floating point numbers (***float***). The operations defined on the basic types are:

- ***boolean*** :

equality	= =
difference	!=
logical and	&
logical or	|

- ***int*** :

addition	+
subtraction	-
multiplication	*
division	/
rest of the division	%
equality	= =
difference	!=
lower than	<
lower or equal	<=
greater than	>
greater or equal	>=

- ***float*** :

addition	+
subtraction	-
multiplication	*
division	/
equality	= =
difference	!=
lower than	<
lower or equal	<=
greater than	>
greater or equal	>=

3.2.2. VTF (long) Names

In the ***VTF*** language we decide to use two different sorts of names: the (*short*)-names and the long-names. From now on, we decide to use the word 'name' instead of short-name. A name is an identifier, this means that it is a set of letters and figures with the only constraint that a name must begin with a letter. A long-name is a set of names separated by dots.

The use of the (long)-names is defined as following. The local variables and the parameters have only got names, the packages have only got long-names. The other nameable notions have got the two sorts of names. These notions use names for the declarations and long names for the use of those notions. The constructors and the methods are not identified by their long-names but by their names and the list of types of the parameters.

The scoop of the names is the package in witch the notions are declared. If a variable *tree* is, for instance, declared in the package *Java.List*, the scoop of *tree* is the package *Java.List*. This means that the variable *tree* is not accessible from elsewhere than in this package.

The uniqueness of the (long)-names is defined as this: packages have got different long-names. Classes in the same package have got different names. Fields of the same class have got different names. Methods (and constructors) of the same class have got different *signatures* (a *signature* is like we have explained above: a name and the list of the types of the parameters). Field and method names of a class are different than the class-name itself. Variables declared in the same block of instructions have got different names. The parameters of a constructor or a method have all got different names.

There is a notion that is called *hide*. If a local variable is declared with the same name as a field of that class, then we say that the field loses his scoop. This means that, for using the field, we have to use the long name of the field, in the scoop of the variable, instead of the name. This rule is also applicable to the parameter names instead of the local variables.

The scoop of the nameable notions is defined by the accessibility attributes. These attributes have got the same names and the same functions as in Java: ***private***, ***public*** and ***protected***.

3.2.3. The Syntax of VTF

Atomic sets *litt*
nclasse, nvar, nmethode, nchamp, npackage

Build sets

prog ::= (***package*** *npackage*)? (***import*** *longnclasse* ***;*** | ***import*** *npackage.* ****;***)*
{ *defclass*+ ***}***

defclass ::= (***public***)? (***abstract***)? ***class*** *nclasse* (***extend*** (*longnclasse* | *nclasse*))?
{ (*declchamp declmethcon declmethabs declconstr*)+ ***}***

type ::= ***int*** | ***bool*** | ***float*** | *nclasse* | *longnclasse*

declchamp ::= (***public*** | ***protected*** | ***abstract***)? (***final***)? (***static***)? *type nchamp* (= *expr*)? ***;***

declmethcon ::= (***public*** | ***protected*** | ***private***)? (***final***)? (***abstract***)? (*type* | ***void***) *nmethode*
((*type nvar* (, *type nvar*)*)? ***)*** ***{*** (*Instr*)* ***}***

declmethabs ::= (***public*** | ***protected***)? (***abstract***)? (*type* | ***void***) *nmethode*
((*type nvar* (, *type nvar*)*)? ***)*** ***;***

declconstr ::= (***public*** | ***protected*** | ***private***)? *nclasse* ***(*** (*type nvar* (, *type nvar*)*)? ***)***
{ ((***super*** | ***this***) ***(*** (*expr* (, *expr*)*)? ***)*** ***;***)? (*Instr*)* ***}***

instr ::= *type nvar* (= *expr*)? ***;***
| *nvar* = *expr* ***;***
| (*desinst.* | *nclasse.* | *longnclasse.*)? *nmethode* ***(*** (*expr* (, *expr*)*)? ***)*** ***;***
| ***return*** (***(*** *expr* ***)***)? ***;***
| ***if (*** *cond* ***)*** *Instr*
| ***if (*** *cond* ***)*** *Instr* ***else*** *Instr*
| ***while (*** *cond* ***)*** *Instr*
| ***{*** (*Instr*)* ***}***

primdes ::= *nvar*
| *nchamp*
| *longnchamp*
| *nclasse.nchamp*

vardes ::= *primdes*
| *desinst.nchamp*

desinst ::= ***this*** | ***super*** | *vardes*
| ***new*** (*nclasse* | *longnclasse*) ***(*** (*expr* (, *expr*)*)? ***)***
| (*desinst.* | *nclasse.* | *longnclasse.*)? *nmethode* ***(*** (*expr* (, *expr*)*)? ***)***

expr ::= ***null*** | *litt* | *expr op expr* | *desinst*

*Im 3.1: Concrete syntax of **VTF***

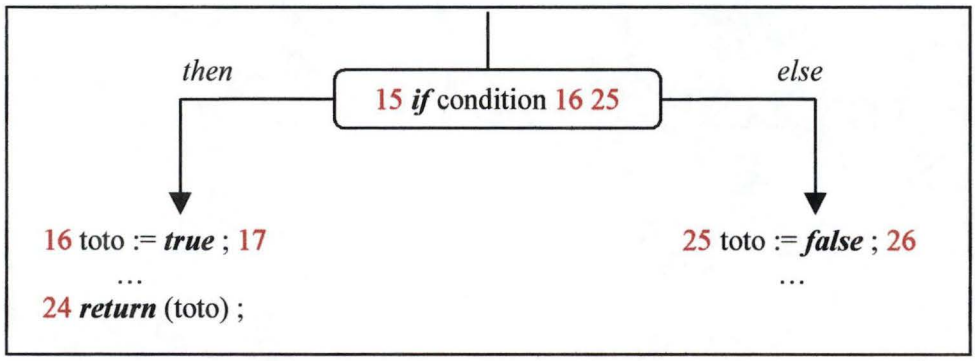
3.3. The *L*abelled *A*bstract *S*yntax

3.3.1. General Idea

The abstract syntax we are explaining in this section is called ***L***abelled ***A***bstract ***S***yntax, hence ***LAS***. We decide to make this syntax a "*labelled*" syntax. It is important to have a labelled syntax in order to make an analysis on the transitions of the operational semantics. We need the labels to make it possible to locate the statements in a univocal way in a given program. And that is the reason why the labels have got to follow a certain logic.

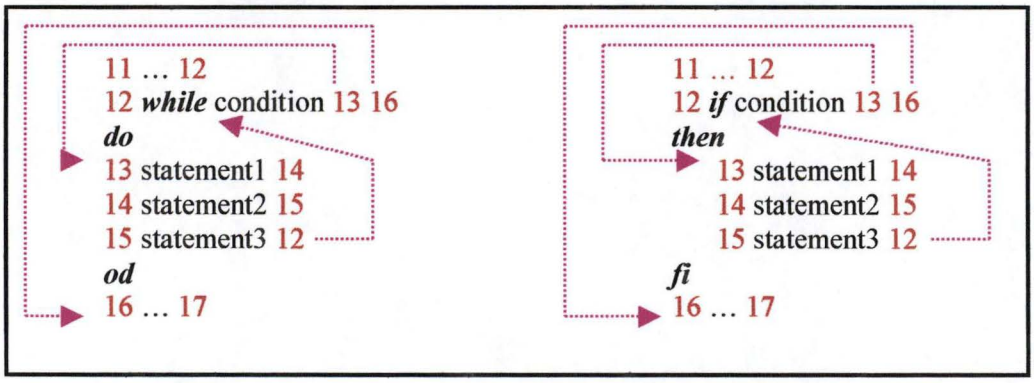
All the statements and all the method and constructor declarations contain a label. For each statement in the middle of a statement list we have got the label of the statement and the label of the following statement. There are two special statements, the ***return*** and the ***if***-statement. The ***return***-statement does only contain one label. This is the label of the statement itself. The ***if***-statement contains three labels: the label of the statement, the label of the if part of the statement and the label of the else part of the statement.

For instance:



Im 3.2: the three labels of the if-statement

The *LAS* does not correspond exactly to the concrete syntax *VTF*. In the *VTF* syntax there are loops (***while***) while the *LAS* does not accept those loops. We must simulate those by an ***if***-statement with some special labels. We can see that the following *VTF* loop is equivalent to the following *LAS* ***if***-statement. As we think a little bit about the ***while***-statement, we discover that the labels are placed in a special way. Indeed, we can see that the second label of the last statement refers to the beginning of the loop and that the condition of the loop in fact exactly looks like an ***if***-statement. We can more easily see that on the image *Im 2.3*.



Im 3.3: while versus if loops

In the *VTF* syntax there are access modifiers, ***static*** properties, ***package*** declarations and ***import*** declarations, while the *LAS* ignores those. Another difference between those two syntax's is that the *LAS* has only got abstract classes with at least one abstract method while the *VTF* can have an abstract class without any abstract method. An *LAS* class also contains an explicit constructor, what is not necessary in the *VTF*. In the *LAS* all the fields are initialised when they are declared, what, once again, is not necessary in the *VTF*. In the *LAS* there is another constraint that tells us that, in every method or constructor, the last statement is a ***return*** statement, *VTF* does not need this. The order of the variable declarations is also important in the *LAS* while not in the *VTF*. All the variable declarations are done in front of the rest of the statements of the methods in the *LAS*. A constructor call is always assimilated to an assignment in the *LAS* while it could be an expression in the *VTF* syntax.

In fact we have got to translate the programs written in *VTF* into the *LAS*. For this translation all the differences between the two language definitions (syntax definitions) must disappear. So instead of making a simple translation, we are creating a translator that makes some adaptations before the real translation job. This translator is explained in the section 3.6. about the type checker. In fact the type checker is made of two essential parts, the syntax checking part and the translation part.

Knowing all the differences between *VTF* and *LAS* we can take a look at the *LAS*-definition.

3.3.2. LAS Definition

Atomic sets *lab*
litt
nclasse, nvar, nmethode, nchamp

Build sets

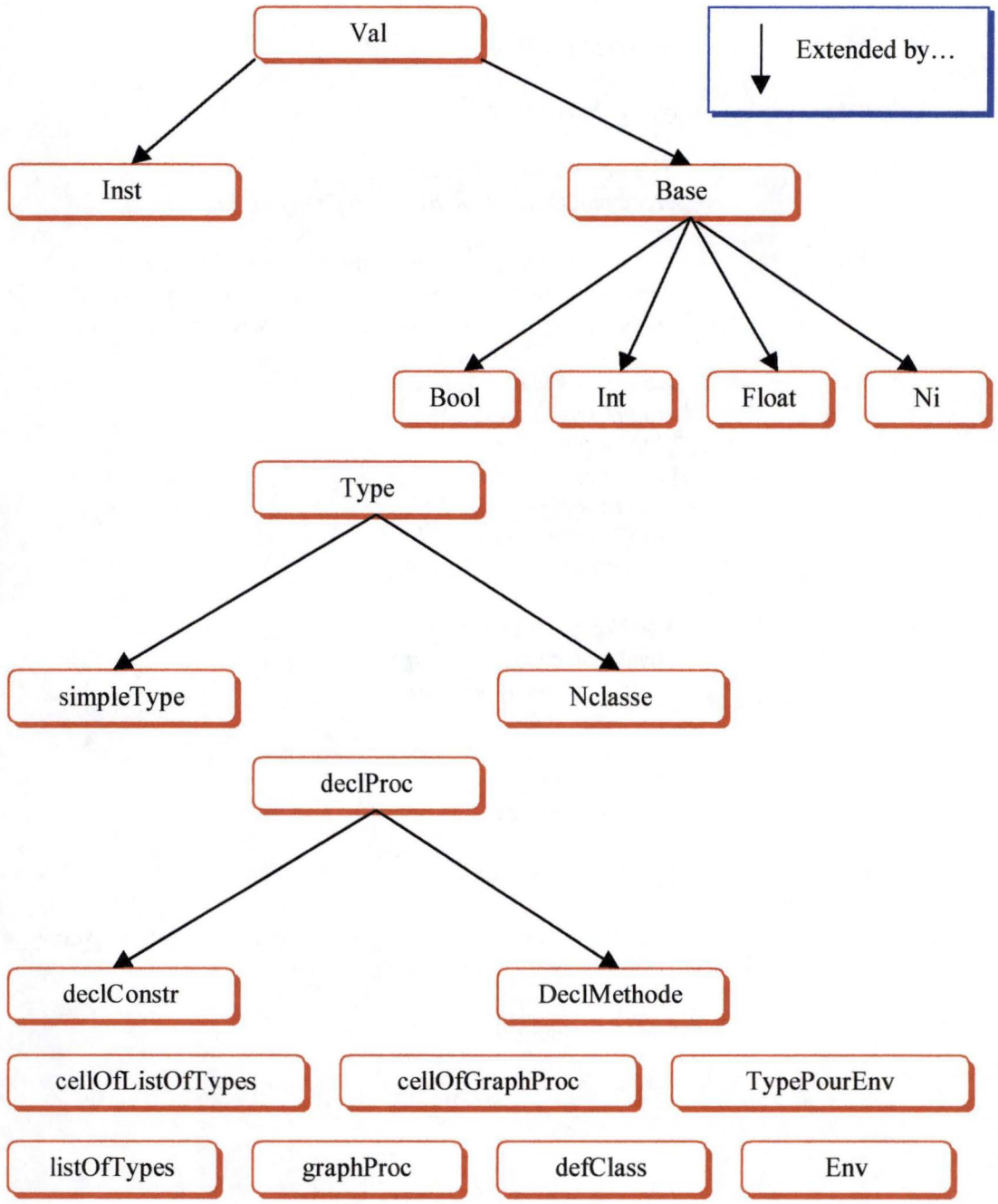
prog	*::=*	*defclass*$^{+}$
defclass	*::=*	*nclasse* [***extend*** *nclasse*] *declchamp** *declmethode** *declconstr*$^{+}$
type	*::=*	***bot*** | ***int*** | ***bool*** | *nclasse* | ***void***
declchamp	*::=*	*type nchamp expr*
declmethode	*::=*	*type nmethode (type nvar)** | *type nmethode (type nvar)** *(type nvar)** *lab instr*$^{+}$
declconstr	*::=*	*nclasse (type nvar)** *(type nvar)** *lab* ***prem*** *lab instr*$^{+}$ | *nclasse (type nvar)** *(type nvar)** *lab* ***super*** *expr** *lab instr*$^{+}$ | *nclasse (type nvar)** *(type nvar)** *lab* ***this*** *expr** *lab instr*$^{+}$
instr	*::=*	*lab* ***affect*** *des expr lab* | *lab* ***if*** *expr lab lab* | *lab* ***skip*** *lab* | *lab* ***proc*** *appel lab* | *lab* ***return*** *[expr]* | *lab* ***fonc*** *nvar appel lab* | *lab****constr*** *nvar nclasse expr** *lab*
appel	*::=*	*type* ***this*** *nmethode expr** | *type* ***super*** *nmethode expr** | *type nvar nmethode expr**
des	*::=*	*type nvar* | *type nchamp* | *type desinst nchamp*
desinst	*::=*	*type* ***this*** | *type* ***super*** | *des*
expr	*::=*	*type* ***null*** | *type litt* | *desinst* | *(expr)*$^{?}$ *op expr* | **(** *expr* **)** **(** *op expr)*$^{?}$

Im 3.4: LAS definition

3.3.3. Tree of the Created Classes

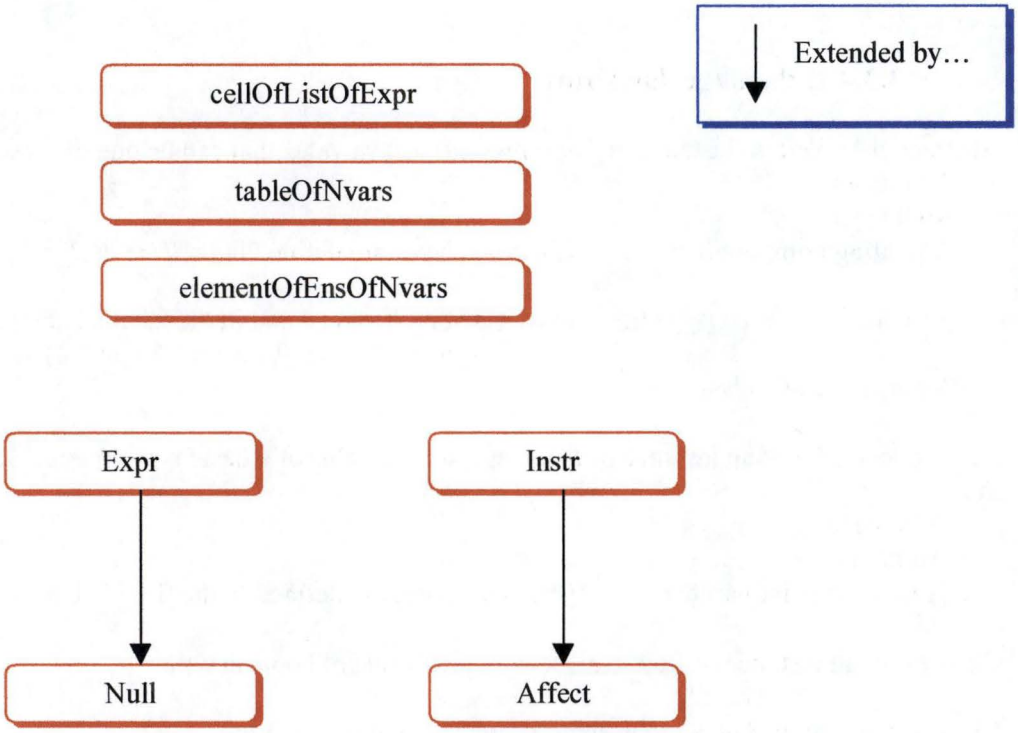
Here, we are developing the structure of all the classes created to represent the objects defined in the *LAS*. All these objects will be created by the type checker during its translation of the *IIR* program into *LAS*.
Afterwards, we present a brief description of all these classes, to explain exactly what they represent. A more detailed specification of all these classes can be found in the annexes.

3.3.3.1. Package JavAbInt



Im 3.5: Classes of the package JavAbInt

3.3.3.2. Package JavAbInt.SAP



Im 3.6: Classes of the package JavAbInt.SAP

3.3.4. Explanation of the Classes

3.3.4.1. Package JavAbInt

- Abstract class *Val*: an instance of *Val* represents a Java value that can be one of those :
  A boolean
  An integer
  A floating point number (the basic types are defined in the *[LC99a]*, p.3)

  An undefined value (type bot) (as defined in the typed and in the labelled abstract syntax in *[IPO99]*, parts 1.3 and 1.4)
  An instance of a class

- Abstract class *Base*: an instance of *Base* represents a value of a basic type that can be one of those:
  A boolean
  An integer
  A floating point number (the basic types are defined in the [LC99a], p.3)

- Class *Bool*: an instance of *Bool* represents a Java value of boolean type

- Class *Int*: an instance of *Int* represents a Java value of integer type

- Class *Ni*: an instance of *Ni* represents a not initialised value (for a basic type) or ***null*** (for a non basic type)

- Class *Inst*: an instance of *Inst* represents a Java value of a non basic type i.e. an instance of a class

- Abstract class *Type*: an instance of *Type* represents a Java type. This can be :

boolean, int, float	(the basic types defined in the *[LC99a]*, p.3)
void	(as used in *[LC99a]* and defined in *[IPO99]*, part 1.2.4)
bot	(as defined in the typed and in the labelled abstract syntax in *[IPO99]*, parts 1.3 and 1.4)
a class name	(as defined in the *[LC99a]*, part 2.2)

- Class *Nclasse*: an instance of *Nclasse* represents a Java class type. It contains all the information available for the class.

- Class *cellOfListOfTypes*: an instance of *cellOfListOfTypes* is a type of a list of types

- Class *listOfTypes*:
  -Implements a domain "list of types"
  -Implements the ordering induced by the ordering on types on lists of types:

  By definition,
  (T1, ..., Tm) <= (T'1, ..., T'n)
  iff
  m = n and
  Ti <= T'i (for all i: 1<= i<= m(=n)).

- Abstract class *declProc*: an instance of *declProc* represents a procedure (i.e. a method or a constructor) declaration

- Class *declConstr*: an instance of *declConstr* represents a constructor declaration.

- Class *declMethode*: an instance of *declMethode* represents a method declaration.

- Class *defClass*: an instance of *defClass* represents a class, with all its proprieties.

- Class *Env*: an instance of *Env* represents a local semantics environment.

- Class *TypePourEnv*: Contains the static information relative to an environment

- Class *cellOfGraphProc*: class that represents a cell in the list graphProc, corresponding to one procedure

- Class *graphProc*:
  - Implements a graph of procedure of "same kind" (constructors of same name and type).
  - Allows one to find a procedure with a given signature.
  - Allows one to find the list of procedures whose list of types is minimally greater that a given list of types.

3.3.4.2. Package JavabInt.SAP

- class *cellOfListOfExpr*: an instance of *cellOfListOfExpr* represents a cell of a list of expressions used in a method or a constructor call

- Abstract class *Expr*: This class implements the expressions of the *VTF* grammar. It's useful to have a type that gathers all the expression types.

- class *Null*: an instance of *Null* represents the ***null*** expression in Java

- Abstract class *Instr*: this class implements the set of statements accepted by the *VTF* grammar. Every statement is uniquely represented.

- Class *Affect*: this class implements the statement "assignment"

- Class *tableOfNvars*: This class is used when translating a declaration of procedure from *IRR* to *SAP*. All parameters must be added before the first local variable is (added).

- Class *elementOfEnsOfNvars*: this class implements an element of the set "ensOfNvar"

3.4. An *I*ntermediate *I*nternal *R*epresentation

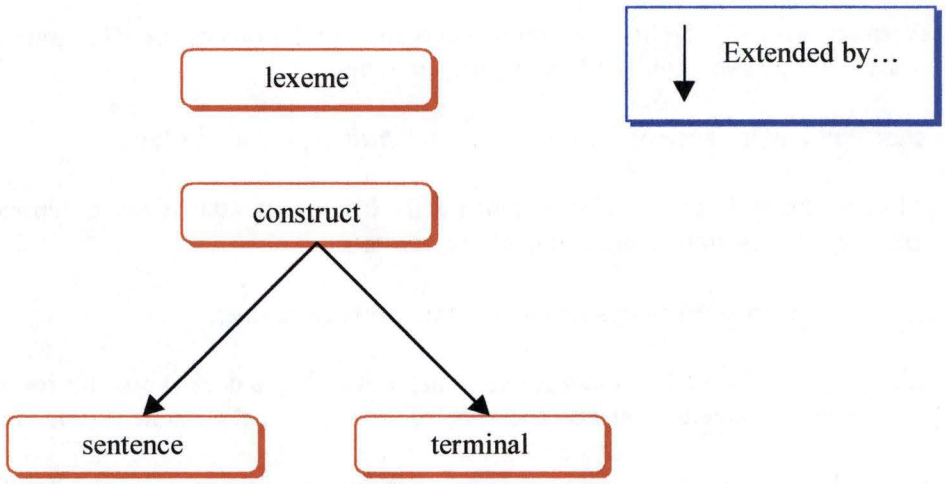
The internal representation we have defined is a direct translation of the *VTF* syntax into a tree structure form.

- The basic structure for the litterals, the identifiers, the operators and the keyword is the *lexeme* structure.
- Another structure is the *sentence* structure. It is used to represent more complex pieces of the program (an expression, a declaration, a statement...). It is a list of objects of type *construct*.
- A *terminal* is an occurrence of a *lexeme* in a *sentence*.
- The *construct* is a general structure that can be either a *terminal* or a *sentence*.

To verify the correctness of the *IIR* structures created by the parser, we need to visualise them. In that aim, we have created a class containing several display methods. In fact, we could have added those display methods into the classes *construct*, *lexeme* and *terminal*, instead of creating a whole new display-class. We did not do this because we did not create the classes of the package *JavAbint.concreteSyntax* by our own. Those classes have been created by Professor Le Charlier. So, because we did not want to change the inside of those classes, we had to create a new class that uses the *toString* methods created by Professor Le Charlier to display the wanted information.

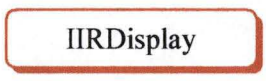
3.4.1. Tree of the Created Classes

3.4.1.1. Package JavAbInt.concreteSyntax



Im 3.7: Classes of the package JavAbInt.concreteSyntax

3.4.1.2. Package JavAbInt.concreteSyntax.Display



Im 3.8: Classes of the package JavAbInt.concreteSyntax.Diplay

3.4.2. Explanation of the Classes

3.4.2.1. Package JavAbInt.concreteSyntax

- Class *lexeme*: This class implements the set of lexical items that are relevant for the ***I***ntermediate ***I***nternal ***R***epresentation of *VTF* programs. These are:
  - Identifiers (*[JLS96]* chapter 3 section 8)
  - Keywords (*[JLS96]* chapter 3 section 9)
  - Literals (*[JLS96]* chapter 3 section 10)
  - Operators (*[JLS96]* chapter 3 section 12)

  Here we use the classification of Chapter 3 of Java Language Specification *[JLS96]*, from which we eliminate irrelevant symbols. Moreover, only the symbols defined in *VTF* are recognised. Every lexical item is uniquely represented.

- Class *construct*: A *construct* either is a *terminal* (*lexeme*) or a *sentence*. In the latter case, it is, in fact, an instance of a non-terminal, i.e. a data structure exhibiting the value and structure of this non-terminal instance.

- Class *sentence*: A *sentence* consists of
  - A "main cell" containing
    + The sort of the *sentence* (statement, expression, etc.)
    + The reference to the first *construct* of the *sentence*
    + The reference to the last *construct* of the *sentence*

      P.S.: both pointers are ***null*** if the *sentence* is empty
  - A sequence of objects of type *construct* represents the *sentence* in a structured way.

- Class *terminal*: A *terminal* (*lexeme*) is an occurrence of a *lexeme* in a *sentence*.

3.4.2.2. Package JavAbInt.concreteSyntax.Display

- Class *IIRDisplay*: this class contains all the methods that allow the displays an object of type *IIR*.

3.5. Implementation of the Parser: Newlook1

3.5.1. Lexical Analyser

The questions we try to answer in this sub section are:

- What is a lexical analyser?
- Why do we use those analysers?
- Do we really need a lexical analyser?

A lexical analyser is a tool that partitions an input program text into the smallest meaningful sequences of characters. This tool then attaches these smallest meaningful sequences of characters to the tokens. And this tool also eliminates the white spaces and the comments from the text program (*[SA98a]* slides 1 till 7). This enables us to resolve a large class of problems like text processing, code enciphering and compiler writing. In our case, we use such a lexical analyser in order to make a parser. Making a parser consists in two major steps. The first step is the lexical analysis and the second step is the generation of the parser. *lex* is one of the best known lexical analysers

It is not essential to use a pre-made tool like *lex* to handle problems of this kind. It is of course possible to write a program in a standard language to handle them. The advantage using this kind of tools is that it offers a faster and easier way to create programs to perform lexical analysis. Its weakness is that it often produces programs that are longer and execute more slowly than hand-coded programs that do the same task. In many applications size and speed are minor considerations, and the advantages of using pre-made tools considerably outweigh these disadvantages.

3.5.2. A Parser Generator

Here, we try to answer some questions about a parser generator:

- What is a parser generator?
- What do we use those tools for?

A parser generator is a tool that takes a language, as argument, and that produces a parser for this language. A parser is a tool that translates a program from one language into an other. The source languages are often programming languages like Fortran, Pascal, C or Java. And the derivation languages are often tree forms of the source programs. These tree representations can be analysed by a compiler. In our case, we can see that the tree representation of the source code is the *IIR*. The name of our parser is *Nlook1* and the explanation of this parser and the generation of it can be found in the next sub-sections.

3.5.3. Java Compiler Compiler Documentation

Like we try to explain in the sub-sections above, there are two parts in the making of a parser. The making of a parser is divided into the syntactical analyser and the parser generating part. In our case we use a tool called *Java Compiler Compiler* (*JavaCC* for more information and for a downloadable version of the Java Compiler Compiler, we refer to the following address *[JavaCC1]*) in order to generate the parser. This tool creates a parser for the language you give him as source file.

Now we know that the creation of a parser is split into two different steps. The first step is to decompose the language into tokens (lexical analysis), and the second step is to generate a syntactical tree corresponding to the language given as argument (parser generation). The language we chose to make a parser for is *VTF*. This language is explained at section 3.2. With this *JavaCC* tool we create our parser called *Newlook1*.

There are two different kinds of parsers: top down parsers and bottom up parsers. The major difference between the two is the way they look at the string of tokens. The top down parser starts with the start symbol and ends with the string of tokens, while the bottom up parser starts with the string of tokens and ends with the start symbol. For more information on the top down and bottom up parsers, we refer to section *3.5.5. Bottom Up versus Top Down Parsing*.

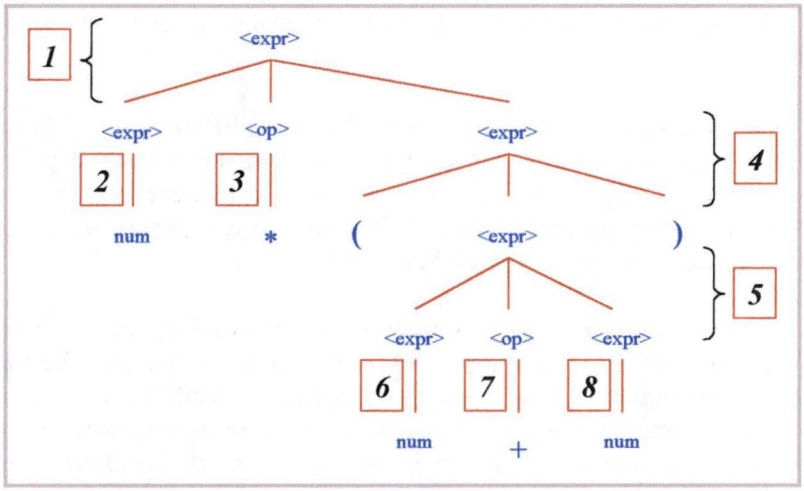
The source file for *JavaCC* is composed of a number of sections:

- *Option Parameters*
- *Main Methods*
- *Definition of the Tokens*
- *Parsing Methods*

JavaCC is much like *lex* and *yacc* (*[LexYacc]*) together because like those tools *JavaCC* creates a parser from a template file. However, while ***yacc*** produces a bottom up (*LALR(1)*) parser, *JavaCC* creates a top down (*LL(k)*) parser. But in our case we decide not to use the possibility of creating an *LL(k)* (for $k \neq 1$) parser but an *LL(1)* parser. We try to explain the meaning of the k in *LL(k)* in the sub-section 3.5.6. called *the k in LL(k)* (*[JavaCC2]*). The notions like *LL(k)*, top down and bottom up are explained in the next sub sections or for more detailed information we refer to *[INFO2108]*.

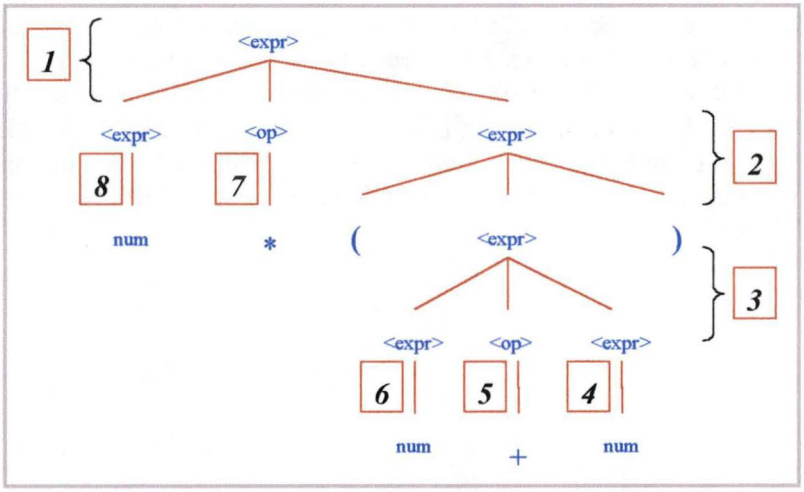
3.5.4. *Left-Most Derivation versus Right-Most Derivation*

In the left-most derivation you have got to find the leftmost non-terminal, in the string, and apply a production to it. This explanation is not that intuitive, but gets comprehensible with the following simple example (*[SA98b]*). In this example you can see how a left-most derivation takes place:



Im 3.9: Left-Most derivation

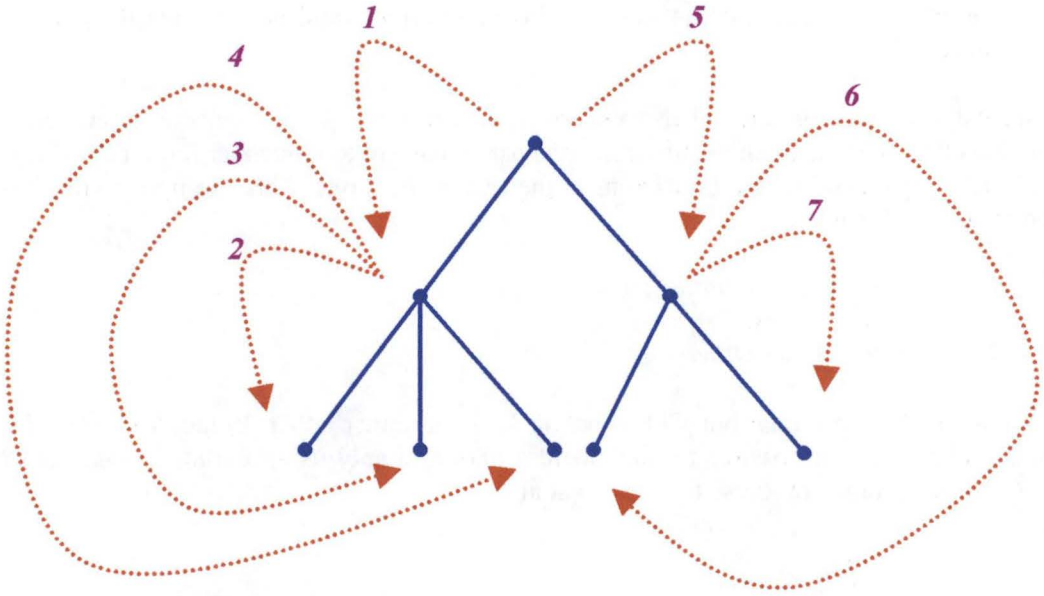
In a right-most derivation, on the other hand, you have got to find the right-most non terminal, in the string and apply a production on it. In the following example you can see how a right-most derivation takes place:



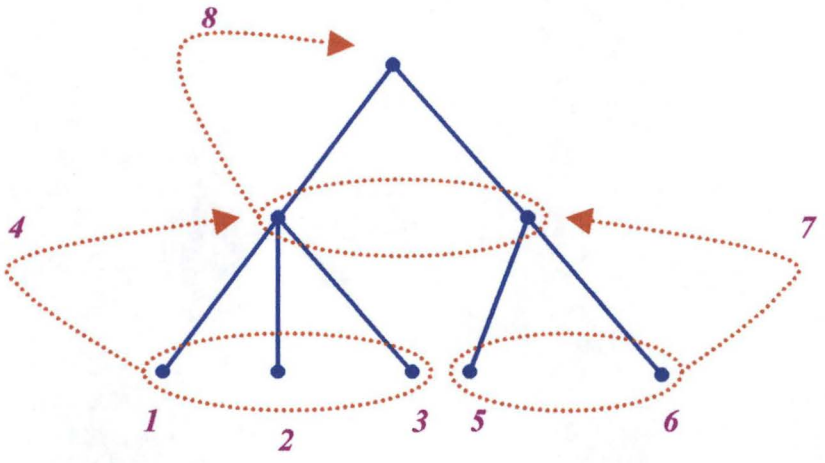
Im 3.10: Left-Most derivation

3.5.5. Bottom Up versus Top Down parsing

We normally scan from left to right. In these left to right parsers, we have got two different kind of parsers. The *LL* and the *LR* parsers. The *LL* parsers are what we call *L*eft to right, *L*eft most derivation. These parsers reflect the top down parsers. This means that the parser starts with the root (or the top) and processes the sentence all the way down to the leaves (or the bottom). This is explained on the *image 3.11.* LR on the other hand is the acronym for *L*eft to right, *R*ight most derivation. These kind of parsers reflect the bottom up kind of parsers. This means that the parser starts with the leaves (or the bottom) and processes the sentence until it comes to the root (or the top). This is explained on the *image 3.12.* The difference between the top down and the bottom up parsers is explained with some more detailed examples into *[SA98b]* slides 90 till 108.



Im 3.11: The top down way to look at a sentence



Im 3.12: The bottom up way to look at a sentence tree

3.5.6. The k in LL(k)

When we speak about a *L*eft to right, *L*eft most derivation parser, we often use the acronym *LL(k)*. In this term, the *k* means that the parser looks *k* tokens in advance before taking a decision. This means that the parser keeps a stack with the read tokens and that this stack can grow up to a height of *k* levels. This stack helps the parser decide when there is an ambiguity. When there is no ambiguity, it is clear that the parser can decide directly and that it does not need the stack.
For instance:

> ***class*** *identifier* ***extends*** *identifier*
>
> this example is easy to parse because there are some keywords to identify what sort of line this is. In fact, we can say that this production could be recognised by an *LL(1)* parser.

The second example, on the other hand, is more difficult. As we can see there are two productions that begin with an identifier, so, the parser has got to remember these identifiers and look ahead. The second token is different in the two productions. This example could thus be solved by an *LL(2)* parser.

1 *identifier = identifier ;*

vs.

2 *identifier identifier = ...*

VTF is not an *LL(1)* grammar but VTF is not an *LL(k)* grammar either. In fact VTF is an *LL(∞)* grammar. Thus it is not possible to implement a parser, simply using the *lookahead* option of *JavaCC*. So we decide not to use this option at all.

3.5.7. Structure of the Parser

3.5.7.1. Options Parameters

As we have seen in the previous sub section, it is not possible to make an *LL(∞)* parser directly using the options of *JavaCC*. Thus we decide to let all the parameters of JavaCC to their default values.

In the following example we show you that the *VTF* grammar is an *LL(∞)* grammar because even if the *k* is big, *i* could still be bigger. And thus, we do not know how many tokens we have to read before being sure that we have a field designator or a method call.

A field designator can have the following form:
$name_1.meth_1(param\_l_1).meth_2(param\_l2).name_2. \ldots name_i.field\_name$

And a method call
$name_1.meth_1(param\_l_1).meth_2(param\_l_2).name_2. \ldots name_i.meth\_name(param\_l)$

Im 3.13: LL(∞)

The only difference is that in the case of a method call, it ends with an identifier (a string) followed by a list of parameters (between brackets). While it only ends by an identifier, in the case of a field name. We can not know the number of tokens we have to read before to decide. In fact, we have got to adapt the number of tokens to look ahead dynamically during the parsing of the to analyse program.

From an implementation point of view, we decide to make a skeleton of the to parse program using the default value of the *lookahead* option of the *JavaCC*. And we add some code to this skeleton in order to take care of all the problems involved with the fact that the grammar is an *LL(∞)* grammar. *JavaCC* is implemented so that it allows us to add some Java code in the middle of the parser. So we use this opportunity to create an *LL(∞)* parser that accepts the *VTF* grammar.

3.5.7.2. Main Methods

The class *Nlook1*, i.e. the parser, contains two main methods:

- The *TestVarDesign* method tests if an object of type *sentence* representing a variable designator contains a parameter list. If the object does not contain such a list, then it means that the *sentence* can represent a long class name

  pre : the *sentence* in entrance is a variable designator
  it only contains identifiers and effective parameter lists
  post : true if the sentence does not contain a parameter list

- The *IIRParser* method is the main method of the class, which will be automatically called at the execution of the Parser. It can return a *ParseException*, which is a class of exceptions defined by the compiler compiler, JavaCC.

  Pre: this method takes the text file of a Java program, that is supposed to be syntactically correct, following this definition of "syntactical correctness", as argument:

  A program in entrance of this method is "syntactically correct" in our terms if it respects the rules that are not checked by the parser. These rules can be found in the documentation of the parser (sub-section *3.5.8. Nlook1 class documentation*).

  Post: this method returns the syntactic tree of the program given as argument, corresponding to the *IIR* definition. The return value is a sentence.

3.5.7.3. Definition of the Tokens

Before defining all the tokens of the language, we have got to define the characters the parser has got to skip. There are the white spaces, the end-of-line characters and the tabulations and, of course, the two types of comments ('// ..... *eol* ' or '/* ..... */').

Then, we have got to define all the tokens of the grammar.

- The tokens corresponding to the operator keywords: general mathematics operators and comparison operators.
- The tokens of the literal values of our language: booleans, floating point numbers and integers.
- The basic keywords of the language: the access modifiers (***public***, ***abstract***, ...) and the keywords associated to the statements (***if***, ***else***, ***super***, ...).
- The tokens corresponding to the basic types: ***int***, ***boolean*** and ***float***.
- The tokens of the identifiers in general: strings beginning by a letter, containing letters, digits or the character '_'.

3.5.7.4. The Parsing Methods

Each method corresponds to the treatment of a part of the program.

Our methods are:

- MainProg()
- Declaration()
- Statement()
- Design()
- Operator()
- ExprLitt()
- Expression()
- Input()

The only argument of the methods is the text of the to analyse program. The methods return the syntactic tree of the (part of) program, they have analysed, into the *IIR* form.

The Input method treats the whole program text followed by the end-of-file character. It calls the *MainProg* method and is called by the main method *IIRParser* of the parser.

Each method has the following form:

```
returned_structure_type method_name()
{
Local_variables
}
{
        code line       ( { Java code } )?
   ( |  code line         { Java code } )*
}
```

A *code line* is a succession of tokens or calls to other methods. It represents a sequence of words in the program text. The Java code is what has got to be executed when the parser encounters that sequence of words in the program. In our program, it is the creation of the corresponding ***IIR*** structure.

The best way to understand all this is to look at a concrete example.

3.5.7.5. Example

Here is a typical example you can find in the JavaCC documentation (*[InriaJavaccEx]*).

This simple grammar recognises a set of left braces followed by the same number of right braces and finally followed by an end of file.

A legal string has got the form:
"{}", "{{{{{}}}}}"...

Some illegal strings are:
"{{{{", "{}{}", "{}}", "{{}{}}"...

The method *Input( )* also prints the number of pair of braces on the screen.

```
PARSER_BEGIN(Simple3)

public class Simple3 {
 public static void main(String args[]) throws ParseException {
   Simple3 parser = new Simple3(System.in);
   parser.Input();
 } }

PARSER_END(Simple3)

SKIP :
{  " "
| "\t"
| "\n"
| "\r" }

TOKEN :
{  <LBRACE: "{">
| <RBRACE: "}"> }

void Input() :
{ int count; }
{
 count=MatchedBraces() <EOF>
 { System.out.println("The levels of nesting is " + count); }
}

int MatchedBraces() :
{ int nested_count=0; }
{
 <LBRACE> [ nested_count=MatchedBraces() ] <RBRACE>
 { return ++nested_count; }
}
```

Im 3.14:example of simple grammar

3.5.8. Nlook1 Class Documentation

public class *Nlook1*

An instance of the class *Nlook1* is a special object, containing the stream of characters with the code of the to parse program given as input and different information on the tokens and on the options of the parser.

sentence IIRParser (InputStream)

Pre: this method takes the text file of a Java program that is supposed to be syntactically correct, following the definition of "*syntactical correctness*" as argument.

Post: this method returns the syntactic tree of the program given as argument, corresponding to the *IIR* definition. The return value is a *sentence*.

In the *IIRParser* precondition, we use the notion of "*syntactical correctness*". Here is a definition of what this notion exactly means in this method. The parser methods check that the construction given in entrance follows the rules of the *VTF* syntax. But they do not check the following rules:

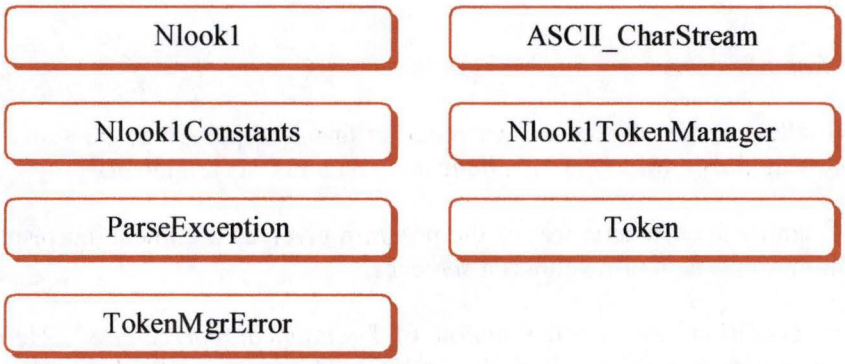
- They do not check that the declarations (after the ***package*** and the ***import*** declarations) are class declarations.
- They do not check that the declaration in entrance has got allowed access modifiers.
  Ex: the method would accept a class defined as ***protected*** and ***static***.
- They do not check that the declarations in the body of a class are allowed declarations (not a class declaration).
- They do not check that a field is not declared with the type ***void***.
- They do not check that a field or a method is declared with an identifier.
  Ex: you can declare int = 1;
- They do not check that the statements defined in a simple method (not a constructor) are different from a constructor call.
- They accept that a designator is only followed by a semicolon but they should not accepted that.
  Ex: it accepts toto; which does not mean anything.
- They do not check that a designator has got two consecutive lists of parameters.
  Ex: you can have toto.add(x,y)(z)

A program in entrance of this method is "*syntactically correct*" in our terms if it respects the rules that are not checked by the parser. Our parser could be improved by adding code that tests all these constraints. This is, of course, possible in a theoretical point of view, but in a practical point of view, it is much easier to make all these verifications in the type checking part of the compiler, while creating the objects of the *LAS*.

3.5.9. Tree of the Created Classes

package JavAbInt.concreteSyntax.Parser

All theses classes are created by JavaCC, using the file *Nlook1.jj*.



Im 3.15: Classes of the package JavAbInt.concreteSyntax.Parser

See more detailed documentation about JavaCC for an explanation of these classes (*[JavaCC1]* and *[JavaCC2]*).

3.6. Type checking and translating the *IIR*

3.6.1. Structure of the Type Checker

The type checker has got several functions. These are the syntax checking, the translation and the type checking. Thus, in fact, we have chosen a wrong name for this tool. But we actually implemented the tool using this name, so we decide to keep the name, keeping in mind that it does not only check the types of the *IIR*.

3.6.1.1. Syntax Checking

The type checker completes the syntax checking because the parser does not check everything. The list of things that are not checked by the parser can be found in the sub-section *3.5.8. Nlook1 Class Documentation.*

3.6.1.2. Translation

The type checker translates the *IIR*-program into a *LAS* form. This translation is not that easy because the *LAS* contains a lot of constraints. Therefore we have to transform the program and to make some restrictions.
For example, every class contains at least one constructor in the *LAS*. So if the initial program does not contain a constructor, a default constructor is created by the type checker. In fact that is exactly what the real Java compiler does.

The translation rules between *IIR* and *LAS* can be seen as the translation rules between *VTF* and *LAS* because the *IIR* is a direct translation of *VTF* in a tree structure.

For the rest of this sub section we speak about an abstract syntax called *SA*. This syntax is defined into *[IPO99]* (p.13-17 called *1.2 une première syntaxe abstraite*). It is a simple vertion of an abstract syntax and the *LAS* is derived from this simple syntax.

In the following table, you can find an easy comparison between *VTF* and *SA*, with all the translations rules. The translation rules between *SA* and *LAS* can be found in the thesis of Isabelle Pollet (*[IPO99]*). The blue text references to the different transformations that have to be done for the translations. These transformations are explained in more details after the tables. The structures written in red are the structures that undergo the transformations.

Here is an explanation of the translation rules:

Vas-T'y-Frotte	Abstract syntax (S. A.)
prog :: = (***package*** *npackage ;*)$^{?}$ (***import*** *nclasse ;* | ***import*** *npackage.* ;*)* (*defclass*)$^{+}$	*prog :: =* (*defclass*)$^{+}$ ***Restriction :*** *package, import* $^{(\alpha)}$
defclass :: = (***public***)$^{?}$ (***abstract***)$^{?}$ ***class*** *nclasse* (***extends*** *nclasse*)$^{?}$ (*declchamp* | *declmeth* | *declconstr*)*	*defclass :: =* *nclasse* (***extend*** *nclasse*)$^{?}$ *declchamp* declmethode* declconstr** ***Restriction :*** *access modifier* $^{(\alpha)}$ *abstract property* $^{(\beta)}$ ***Transformation :*** *explicit constructor* $^{(1)}$
declchamp :: = (***public*** | ***protected*** | ***private***)$^{?}$ (***final***)$^{?}$ (***static***)$^{?}$ *type nchamp* (*= expr*)$^{?}$ *;*	*declchamp :: =* *type nchamp expr* ***Restriction :*** *access modifier* $^{(\alpha)}$ ***Transformation :*** *initialisation is needed* $^{(2)}$
declmeth :: = ((***public*** | ***protected*** | ***private***)$^{?}$ (***final***)$^{?}$ (***static***)$^{?}$ (***void*** | *type*) *nmethode* ***(*** (*type nvar*)* ***)*** ***{*** (*instr*)* ***}*** | (***public*** | ***protected***)$^{?}$ ***abstract*** (***void*** | *type*) *nmethode* ***(*** (*type nvar*)* ***)***)	*declmethode :: =* *type nmethode* (*type nvar*)* | *type nmethode* (*type nvar*)* (*type nvar*)* *instr*$^{+}$ ***Restriction :*** *access modifier* $^{(\alpha)}$ ***Transformation :*** *at least one statement in a concrete method declaration* $^{(3)}$ *In S.A., declvar is not a statement but transformed into : (type nvar)** $^{(4)}$ ***Remarks :*** *In S.A.,* ***void*** *is a basic type*
declconstr :: = (***public*** | ***protected*** | ***private***)$^{?}$ *nclasse* ***(*** (*type nvar*)* ***)*** ***{*** ((***this*** | ***super***) ***(*** (*expr*)* ***) ;***)$^{?}$ (*instr*)* ***}***	*declconstr :: =* (*nclasse* (*type nvar*)* (*type nvar*)* ***prem*** *instr*$^{+}$ | *nclasse* (*type nvar*)* (*type nvar*)* ***super*** *expr* instr*$^{+}$ | *nclasse* (*type nvar*)* (*type nvar*)* ***this*** *expr* instr*$^{+}$) ***Restriction :*** *access modifier* $^{(\alpha)}$ ***Transformation :*** *Constructor type* ***prem*** $^{(5)}$ *At least one statement*

Vas-T'y-Frotte	Abstract syntax (S. A.)
type :: = (***int*** | ***boolean*** | ***float*** | *nclasse*)	*type :: =* (***int*** | ***bool*** | ***float*** | *nclasse* | ***void***) ***Remarks :*** *In S.A. and in V.T.F. : we don't consider the so-called type String* *In S.A. : addition of the type float*
instr :: = *type nvar* (= *expr*)? ; | *des* = (*type*)? *expr* ; | (*desinst* | *nclasse*)? *nmethode* ((*expr*)*) ; | ***return*** ((*expr*))? ; | ***if*** (*expr*) *instr* (***else*** *instr*)? | ***while*** (*expr*) *instr* | { (*instr*)* }	*instr :: =* ***skip*** | ***affect*** *des expr* | ***if*** *expr* (*instr*)* (*instr*)* | ***proc*** *appel* | ***return*** (*expr*)? | ***while*** *expr* (*instr*)* | ***cast*** *des type expr* | ***fonc*** *nvar appel* | ***constr*** *nvar nclasse expr** *appel :: =* ***this*** *nmethode expr** | ***super*** *nmethode expr** | *nvar nmethode expr** ***Transformation :*** *In S.A, declarations and affectations are split into two parts.* (4) *In S.A., all the variables are declared before the statements* (4) *In S.A., the function and the constructor calls are not considered as expression, unlike in V.T.F.* (6) *In S.A., in a call of a method of this class or of the super class, the* **this** *or the* **super** *is always explicit* (7) *In S.A., all the* ***while*** *statement are translated in their corresponding* ***if*** *form* (8)
des :: = *nvar* | *nchamp* | *desinst.nchamp*	*des :: = nvar* | *nchamp* | *desinst nchamp*
desinst :: = ***super*** | ***this*** | *des* | ***new*** *nclasse* ((*expr*)*) | (*desinst* | *nclasse*)?.*nmethode* ((*expr*)*)	*desinst :: =* ***super*** | ***this*** | *des*
expr :: = ***null*** | *litt* | *desinst* | (*expr*)? *op expr* | (*expr*) (*op expr*)?	*expr :: =* ***null*** | *litt* | *desinst* | *op expr** ***Restriction :*** *the brackets in the expressions operators are always prefixed*

- **Restrictions**

(α) The "package" and the "import" declarations and the access modifier information are not considered in the abstract syntax. There are no static methods or fields in S.A. The compiler simulates these with a dynamic method called *Static_Ref*. This method takes a picture of all the static fields and methods. With this picture, we are able to make dynamic references to all the static fields and methods. In this way, a static method call is transformed into a dynamic method call, but with a reference to the *Static_Ref* method. For the fields, we instanciate those, once and for all, into this *Static_Ref* method.

(β) The abstract property of a class is not completely saved in the abstract syntax. A class is abstract if and only if it contains at least one abstract method. On the other hand, a class can contain no abstract method, but being declared as an abstract class in *VTF*. However, a class that contains an abstract method must be declared as an abstract class.

- **Transformations**

(1) A class contains at least one explicit constructor in the *SA*. If the class does not contain one, the compiler creates a default constructor.

In the case of a class extending another class, which contains a constructor without arguments, the default constructor is:

```
ClassName ( )
{
super ( );
}
```

In the other cases, it is:

```
ClassName ( )
{
}
```

(2) All the fields of a class are declared and initialised at the same time in the abstract syntax. If a field declaration in ***VTF*** does not contain an initial value, the compiler assigns one to this field. If the field type is a class type, the initial value is ***null***.
For the basic types, its default value is:

int : 0
float : 0f
boolean : false

(3) A constructor or a concrete method contains at least the ***return*** statement in the abstract syntax. If needed, this statement is added at the bottom line of the statement list by the compiler. For a constructor, or a method with ***void*** as return type, the added statement is:

return ;

(4) All the local variables are declared before the statements in the abstract syntax. In the case of a *VTF* program containing a variable declaration combined with an assignment, this statement is transformed into two parts, during the translation from the *VTF* into the abstract syntax *LAS*.

For instance:

type nvar = expr ;

Becomes:

type ...
type nvar (in the other variable declarations)
type...

affect nvar expr (in the statements)

(5) A constructor is one of the three different types in the abstract syntax:
prem: when it is a primal constructor (it does not call another constructor)
this: when the constructor calls another constructor of the same class
super: when the constructor calls a constructor of the super class

(6) A function call or a "***new***" call is not considered as an expression in the abstract syntax. This call is always associated to an assignment. When the compiler encounters one of these procedure calls elsewhere than in an assignment, while translating a *VTF* program into its *LAS* form, it creates a new internal variable and decomposes the statement as follows:

return (x.toString ()) ;

becomes:

fonc Tmp_1 x toString
return Tmp_1

Similarly,

NewList = list.addCell (***new*** Cell (info, next));

becomes:

constr Tmp_2 Cell info next
fonc NewList list addCell Tmp_2

(7) In a call of a method of this class or of the super class, the ***this*** or the ***super*** keyword is explicit in the abstract syntax. That keyword is added by the compiler when it is absent in the *VTF* program.

For instance,

MethCall (EffParam);

Becomes:

proc this MethCall EffParam

(8) In the definition of the abstract syntax, the ***while*** statements do not exist anymore. To translate a program from its *VTF* form to its abstract syntax form, we have to translate the ***while*** statements into their corresponding ***if*** statement.

For instance:

```
while ( bool )
        { stat }
```

Becomes:

```
01 if ( Tmp_3 )
        {
        stat
        jump to 01
        }
```

3.6.1.3. Type Checking

The type checker checks all the types in the program and translates the program at the same time. The type checker verifies different things:

- It verifies that the value returned by a method corresponds to the return type found in the declaration of the method.
- It verifies, in the case of a method or a constructor call, that the types of the real parameters correspond to the types of the formal parameters of the called method or constructor. This verification includes the target parameter of the method.
- It verifies, in the case of an assignment, that the type of the assigned value (or variable) corresponds to the type of the target variable.
- It verifies the type of the conditions (they must be of type boolean, of course) in the ***if*** statements.
- It verifies that all the types used in the program (extended type, type of a declared variable or field, type returned by a method…) have been declared before their use.
- It verifies the validity of the types of the expressions used with the arithmetic operators.

It also verifies other things:

- It verifies that a variable is not declared two times somewhere in the program
- It verifies that there are no duplicate definition of a method in the program
- It verifies that there are no unreachable statements in the program. The type checker can detect some unreachable statements but it can not detect all of them.

When the type checker encounters an error in the program, it throws an exception. We have created a sub-class of the basic class ***exception*** (*CheckTypeException*) so we can add some useful information in the exception message returned. In this way, we have optioned for simplicity, because it could have been possible to continue parsing the whole program and return a list of errors instead of stopping after the first encountered problem. This seemed to be too difficult to handle for a first version of our type checker. It can, of course, change in later versions of this type checker.

3.6.2. Algorithm of the Type Checker

The first method of the type checker, the method *CheckProgram*, checks the validity of the given program, creating all the *LAS* structures corresponding to the program. It calls several other methods, including the most important one: the method *CheckStatement*. The other methods do not use specific algorithms; they are intuitive (*CheckExpr*, *CheckClass*...).

Our method *CheckStatement* receives a statement as argument. At the first call, this statement is the first statement of the considered method (constructor). This statement is a sentence of sort "*STATEMENT*". It returns the *LAS* structure *Instr* corresponding to the *IIR* statement.

Before applying this method, we suppose that another method has already been applied.
This other method, *CheckReturn*, checks that:

- All the ***return*** statements, contained in the method, are well placed
  I.e.: all statements of the method can be reached
  To check that, we have got to check that any of the ***return*** statements and any of the "blocked ***if*** statement" are followed by other statements.
  We call a "blocked if statement" an ***if*** ... ***else*** where the ***then*** AND the ***else*** statement lists finish with a ***return*** or a "blocked if".
- If the treated method must return a result, it finishes with a ***return*** statement or with a "blocked if".
  Here, we do not check if the type of the returned expression corresponds to the return type of the method.

The method *CheckStatement* considers the case of an empty list of statements (the given sentence is ***null***) as a special case. We have not implemented this part of the method yet, but it does not seem to be too difficult to do. The principle is:

- At the level 0, the method creates a ***return*** statement
- At a higher level, it returns a ***skip*** statement found in the *hashtable* defined just below.

The current ***if*** structure information is kept in a *hashtable* (used as a stack) in which:

- The key is the level of the corresponding ***if***
- The object is a structure containing:
  - Two ***skip*** statements, corresponding to the last statements of the ***then*** and the ***else*** statement lists
  - A ***boolean*** which says if we are treating the ***then*** or the ***else*** part of the ***if***.

The current level is the size of the *hashtable* (corresponding to the number of nested ***if*** statements encountered).

The method *CheckStatement* is a recursive method, for which the basic case is when the given statement is the last one (no following statement).

In the basic case,

- If the current statement is not a ***return***, we have got to check the current level.
  - If we are at level is 0 we have got to add a ***return*** statement, with a default value, if needed. This comes from the fact that in the *LAS* definition, all methods are supposed to finish with a ***return*** statement. The *VTF* language does not have this constraint.
  - Otherwise, we have to check if we are in a ***then*** or an ***else*** part of an ***if***.
    We put the ***skip*** statement found in the *hashtable* corresponding to the ***then*** or the ***else***, as following statement.
- In the case of a ***return***, there is no following statement.
  The label of the ***return*** is the first statement of the method. This information is kept in a global variable.

In the general case,

- The method treats the current statement.
- It calls the method recursively on the following statement.
- It adds the reference to the *Instr* structure of the following statement as label of the current one.

It is simple to put the label of a basic statement, but it is quite difficult for ***if*** (and nested ***if***) statements.

- In the case of an ***if***, this corresponds to put the statement following to the ***if*** as label for the two associated ***skip*** statements.
- If an ***if*** is the last statement of a ***then*** or an ***else*** statement list, the statement following to the nested ***if*** is the ***skip*** of the ***if*** at the upper level.

We are going to show here how the *hashtable* with the ***if*** structure is built.

Ex:
if we have :

```
stat1;
stat2;
if expr1
  {stat3;}
else
  {stat4;
   if expr2
     {}
   else {stat5;}
   }
stat6
...
```

Its ***SAP*** representation will be:

```
1 stat1 2
2 stat2 3
3 if expr1 (4, 6)
    4 stat3 5          // then part
    5 skip 11
    6 stat4 7          // else part
    7 if expr2 (8, 9)
        8 skip 11      // then part
        9 stat5 10     // else part
        10 skip 11
11 stat6 12
12 ...
```

While entering an ***if***,

- A level is added in the *hashtable* :
  - The key is the size of the *hashtable* + 1 (the new current level)
  - The object is a structure with two new *skip* statements
- The method is recursively called with the ***then*** and the ***else*** statement lists. The boolean in the *hashtable* is put to the corresponding value before these calls.
- The method puts the statement following to the ***if*** as labels of the *skip*. The level is removed of the *hashtable* when we leave the ***if*** statement.

We have not thought about the ***while*** statement yet. One possible way to resolve the problem is to transform the *IIR* ***while*** statement into the corresponding *IIR* ***if*** statement, and then call the method on the *IIR* ***if***.

Another point to consider is the treatment of the variables. We use two global variables.
One of type *tableOfNvars*, which contains:

- All the internal variables
- The fields of the current class
- The accessible local variables

One of type *AllVariables*, which contains:

- All the internal variables
- The fields of the current class
- All the encountered variables

The *AllVariables* structure is a *hashtable* in which:

- The key is the index of the variable
- The object is a structure that contains its name, its type…

On this *hashtable*, we have defined methods that return the information needed to create the *typePourEnv* structure.

Keeping in mind that we have to keep all the information about the translation from *IIR* to *LAS*, we are thinking about using an *hashtable* with:

- For key, the *LAS* structure a statement
- For object its *IIR* structure

But we have not implemented that yet. Later we will probably just have to add those lines into the code.

3.6.3. Left to do

The program is not finished yet. If most of the algorithm has already been implemented, we need to finish it and to test it.

As the whole algorithm explained above has been implemented in only one method, the code is quite hard to read, and we have to divide the code into several smaller methods, to make the code more comprehensible.

Anyway, all the classes and methods we have built have been specified. You can find those specifications in annexe. You can also get a copy of the code (*[ZAP00]*). In fact, the code is too large to be included in this work.

4. STATIC ANALYSIS BY ABSTRACT INTERPRETATION

4.1. Introduction

This second part of our work, the syntactical analyser, has been done in the framework of the course of abstract interpretation of Professor Le Charlier.

An analyser in the framework of the large project would have been impossible to build in the small time we had. We chose to analyse a more simple language (the *VSS*: very small subset of Java), to be able to go further in the implementation of the analyser.

This work is an implementation in CaML of an analyser. In this case, we consider that the program has already been transformed to verify the constraints of *VSS*, parsed and translated in its abstract form. The program is supposed to be correct.
The analyser takes the tree of the program (abstract form), applies the multivariant algorithm (which is explained in details in this chapter) and returns all the possible states of the program. The treatment of all that states, to draw useful information, is not performed buy the analyser, and has to be done by the user. This improvement of the analyser could be interesting in the framework of a further work.

4.2. Syntax

4.2.1. A Very Small Subset of Java

4.2.1.1. Constraints of the Language

The programming language we are studying here is a very simple sub-language of the programming language Java, the *VSS*.

In fact, this language has got the following limitations. Most of them have be done, among other things, because we did not have a lot of time for this part of the work, and we preferred to concentrate on the essential.

- There are no basic types anymore. We only work with class instances. This basic limitation is very useful. We don't have to treat anymore with all sorts of value. All values are instances.

- There are no static fields or static procedures. The only static method is the main method. For an easy treatment, we consider that the *main* method must be found in a special class: the *main class*. This class only contains the *main* method, she does not contain any fields or constructors and no instance of that class can be created.
  This limitation really simplifies the analysis. In fact, the treatment of static fields and methods is a specific treatment, quite complex. We chose to forget all that and to concentrate our efforts on the analysis of general fields and methods.

- There are no conflicts between method names. It is impossible to find two methods with the same name except in the case of the redefinition of a method in a subclass. In that case the two methods have thus the same parameter types.
  This simplifies the treatment of method call. The methods can be identified by their name and the name of the class in which there are defined. It is easier to find the method to apply, as we don't have to compare the types of the arguments to find the good one. This comparison is not very difficult to code in a theoretical point of view, but the code added would be heavy and it would make the code less comprehensible, without interesting advantages.

- All the fields of the classes are private. The only way to access to them is using methods. In this way, we do not have to handle long-name (instance_name.field_name).

- There are no more access modifiers considered (private, protected, public...). All the methods are considered as public. This limitation was already present in the first part of the works. The parser accepted the access modifiers but they were completely forgotten by the compiler. The access modifiers complicate the analysis, as you always have to verify that you are allowed to use the field or the method you want to.

- There are no procedures, but only functions. All the methods return a value. From the syntactical point of view, all the methods end with the statement ***return*** *expr*. The constructors always ends with the statement ***return this***. The only method that does not end with a ***return*** statement is the method ***main***. This allows us to treat only one kind of ***return***. Anyway, a program can always be transformed to return something. You can always returns ***null*** at the end of the method, and at the method call, put the mock returned value in a mock internal value.

- All classes contain exactly one constructor. The fact that there is maximum one constructor has the same reason than the fact that the methods have all different names. In fact, we decide to have at least one constructor, in the same way than a real Java compiler, which creates a default constructor in classes without constructor.

There is a constraint that we should have added, but we thought about it too late:
We could say that the return variables in the method and in the constructor calls have to be local variables. This is not very restrictive, as all calls in which the return variable is a field can been translated to respect this constraint.

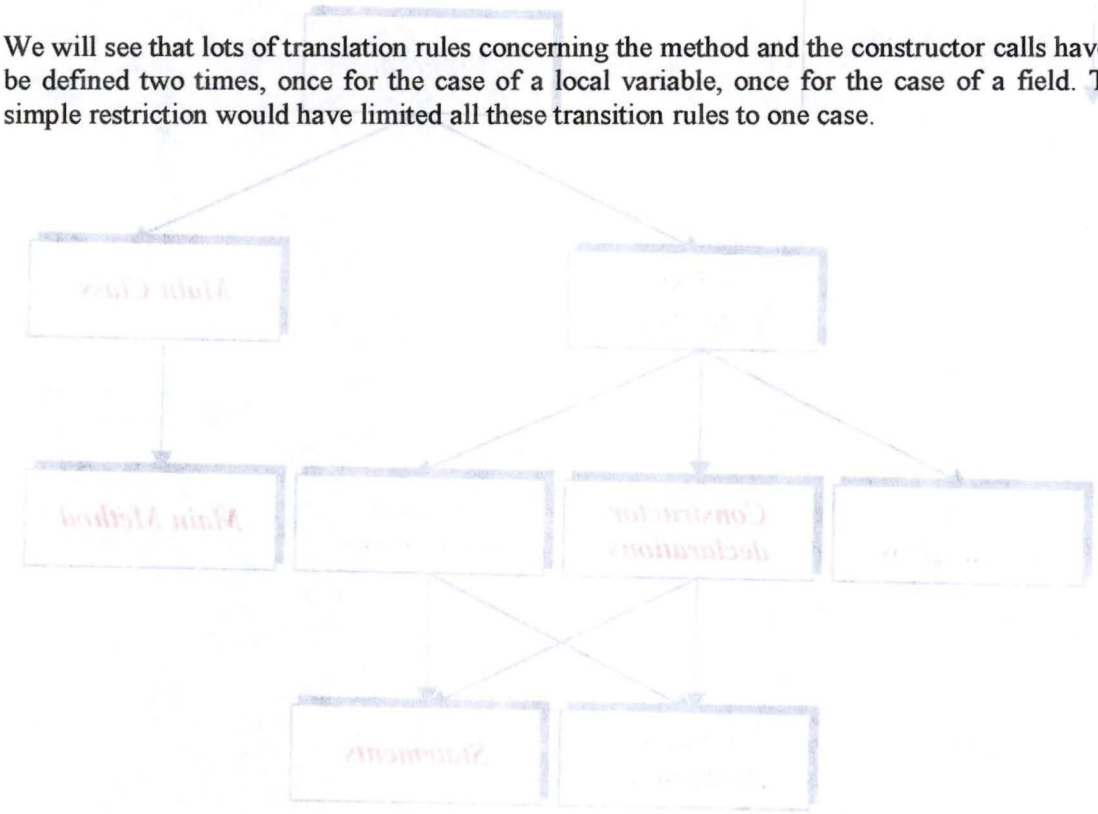
For instance:

```
...
field_name = designator.method_name (param_list);
...
```

Can be translated into:

```
...
var_name = designator.method_name (param_list);
field_name = var_name;
...
```

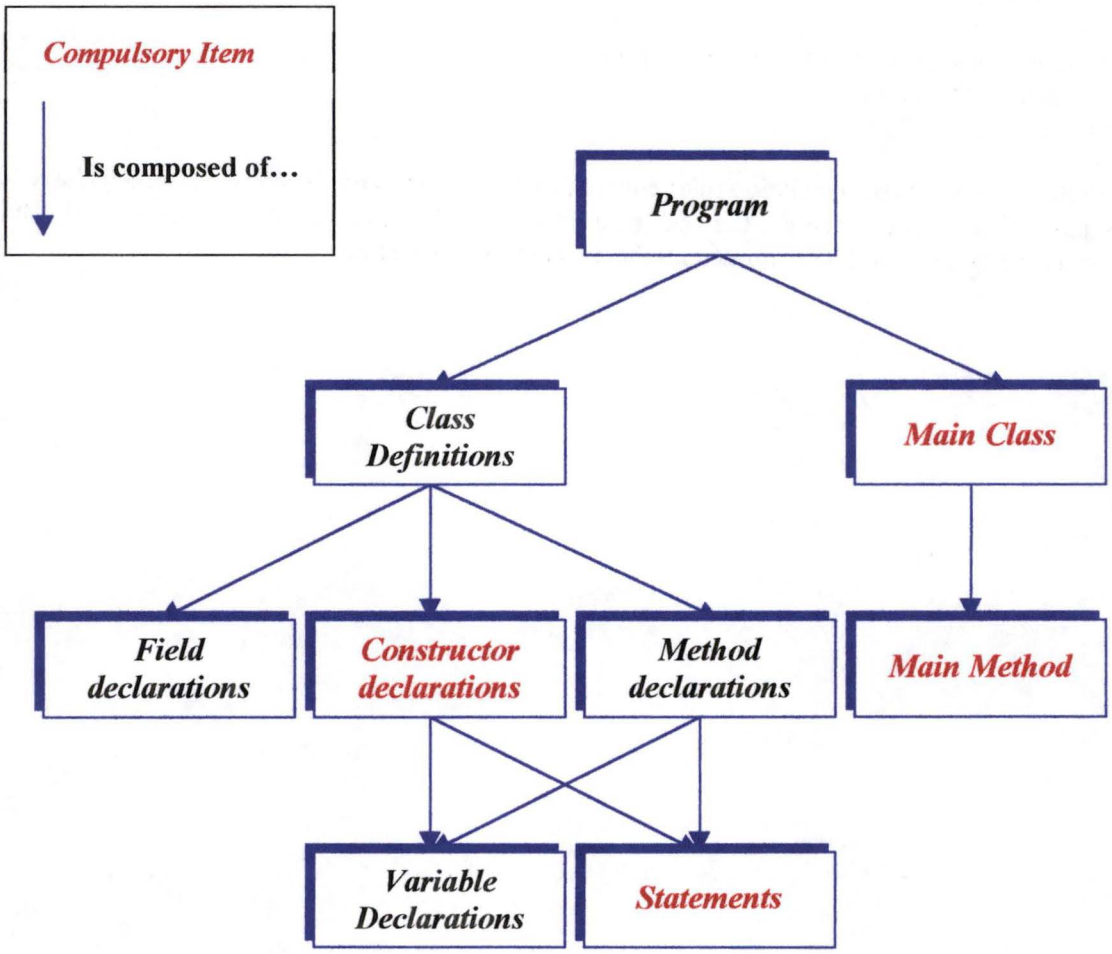
We will see that lots of translation rules concerning the method and the constructor calls have to be defined two times, once for the case of a local variable, once for the case of a field. This simple restriction would have limited all these transition rules to one case.



4.2.1.2. The Syntax of *VSS*

The syntax of our language is the following:

- A *program* is a number of class declarations followed by a Main Class.
- A *class definition* is a number of field declarations, a constructor declaration followed by a number of method declarations.
- A *constructor declaration* is a list of statements.
- A *method declaration* is a list of statements.
- A *list of statements* is a number of variable declarations and a set of statements.
- The *main method* is the only method that is really required, this method is made of a list of statements.
- The *Main Class* is the only indispensable class. This class is made of the main method.



Im 4.1: syntax of VSS

4.2.2. The Concrete Syntax

Id ::= [a..z,A..Z] ([a...z,A...Z,0...9])*

c ::= ***true*** | ***false*** | ***DontKnow***

ClassName ::= *Id*

MethName ::= *Id*

VarName ::= *Id*

FieldName ::= *Id*

FieldDecl ::= *ClassName FieldName* ;

VarDecl ::= *ClassName VarName* ;

FormalParamList ::= *ClassName VarName* (, *ClassName VarName*)*

Des ::= *VarName* | ***this*** | *FieldName*

Expr ::= *Des* | ***null***

Cond ::= *c*
| *Des*.***instanceOf*** (*ClassName*)

Stat ::= *Des* = ***new*** *ClassName*((*Expr* (,*Expr*)*)?) ;
| *Des* = *Expr* ;
| *Des* = *Des*.*MethName* ((*Expr* (,*Expr*)*)?) ;
| ***if*** (*Cond*) { (*Stat*)* } (***else*** { (*Stat*)* })? ;

ConstrDecl ::= *ClassName* ((*FormalParamList*)?)
{ (*VarDecl*)* (***super***((*Expr* (,*Expr*)*)?);)? (*Stat*)* ***return this*** ; }

MethDecl ::= (*ClassName*) *MethName* ((*FormalParamList*)?)
{ (*VarDecl*)* (*Stat*)* ***return*** (*Expr*) ; }

ClassDef ::= ***class*** *ClassName* (***extend*** *ClassName*)? { (*FieldDecl*)* *ConstrDecl* (*MethDecl*)* }

MainMethod ::= ***void main*** ((*FormalParamList*)?) { (*VarDecl*)* ***Init*** (*Stat*)* ***Fin*** }

MainClass ::= ***class ClasseMain*** { *MainMethod* }

Prog ::= (*ClassDef*)* *MainClass*

Im 4.2: concrete syntax of VSS

4.2.3. The Abstract Syntax

Here is the abstract syntax we used.

ClassName ::= Id

*MethName ::= Id + {**main**}*

VarName ::= Id

FieldName ::= Id

FieldDecl ::= ClassName FieldName

*VarDecl ::= pt **var** ClassName VarName pt*

*Des ::= VarName | **this** | FieldName*

*Expr ::= Des | **null***

Cond ::= c
*| **instanceOf** Des ClassName*

*Stat ::= pt **affect** Des Expr pt*
*| pt **new** Des ClassName Expr* pt*
*| pt **proc** Des Des MethName Expr* pt*
*| pt **if** Cond pt pt*

ConstrDecl ::= ClassName (ClassName VarName) (VarDecl)**
*(pt **super** (Expr)* pt)? (Stat)* (pt **return this**)*

MethDecl ::= ClassName MethName (ClassName VarName) (VarDecl)* (Stat)**
*(pt **return** Expr)*

*ClassDef ::= ClassName (**extend** ClassName)? (FieldDecl)* ConstrDecl (MethDecl)**

*MainMethod ::= **main** (ClassName VarName)* (VarDecl)* **Init** (Stat)* **Fin***

MainClass ::= MainMethod

Prog ::= (ClassDef) MainClass*

Im 4.3: abstract syntax of VSS

4.3. Semantics

4.3.1. Concrete Semantics

4.3.1.1. Definitions

In our concrete semantics, a state looks like:

State = ILabel X Stack X (IEnv X Store)

where

IEnv = VarName + {***this***} → ILoc + {***null***} + {***undef***}

We can assume that: $\forall$ e $\in$ IEnv: (e ***this***) $\in$ ILoc

We can consider that the store returns an instance, because the simple language we are studying in this work has not got any basic types. The ClassName value is the type of the instance.

The instance is defined here as a function from the field names to the locations. In fact, it can be seen as the environment associated to the fields.

Store = ILoc → (ClassName X IInstance) + {***undef***}
IInstance = IFieldName → ILoc + {***null***} + {***undef***}

Remark: In this work, the instance of a variable contains all the fields of the class, and the fields of the inherited classes, even if those are private. The access will just be limited to the allowed fields. For that reason, in the case of calls to the ***super*** constructor, the variable ***this*** stays the same when you enter in the constructor. The only difference will be that you will not have access to the same fields.

Stack = $\{(\upsilon_1, \upsilon_2, \ldots \upsilon_n) \mid n \in \mathbb{N} \ \Lambda \ \forall i : 1 \leq i \leq n : \upsilon_i \in$ (IEnv X IPtsProg X ReturnVar)$\}$

Each item of the concrete stack contains the following information:

- An environment
- A return label
- The name of the return variable and its descriptor (***var*** or ***field***)

The descriptor of the return variable is useful for the treatment of the ***return*** statement, which corresponds to an assignment of the returned value into the return variable.

ReturnVar = {***field*** | ***var***} X Des

IDes = VarName + {***this***} + IFieldName

4.3.1.2. Useful Functions

The function *Val* simply returns the value of the given designator, using the given environment and the given store.

If the function *Val* is applied on a local variable or ***this***, the function simply applies the environment function. If the given variable is a field, we use the environment and the store to get the location function associated to the current instance. We apply this location to find the value of the field.
As the program is supposed to be correct, we can assume that the variable belongs to the domain of the environment i.e. that we are looking for the value of a declared variable.

$$\text{Val} : \text{IEnv} \times \text{Store} \times \text{IDes} \rightarrow \text{ILoc} + \{\textbf{\textit{null}}\} + \{\textbf{\textit{undef}}\}$$

$$(e, s, v) \rightsquigarrow \begin{cases} e\ v & \text{if } v \in \text{VarName} + \{\textbf{\textit{this}}\} \\ l\ v \quad \text{where} \quad s\,(e\ \textbf{\textit{this}}) = (t, l) & \text{if } v \in \text{IFieldName} \end{cases}$$

The function *getType* takes an instance as argument. It returns the type of the given instance. This type is the name of a class.

$$\text{getType} : \text{IInstance} \rightarrow \text{ClassName}$$

$$(t, e) \rightsquigarrow t$$

The function *Type* returns the dynamic type of any variable. In the case of an instance, this function uses the *getType* function defined just above.

$$\text{Type} : \text{IEnv} \times \text{Store} \times \text{IDes} \rightarrow \text{ClassName} + \{\textbf{\textit{null}}\} + \{\textbf{\textit{undef}}\}$$

$$(e, s, v) \rightsquigarrow \begin{cases} \textbf{\textit{undef}} & \text{if } s(\text{Val}\ (e, s, v)) = \textbf{\textit{undef}} \\ \textbf{\textit{null}} & \text{if } s(\text{Val}\ (e, s, v)) = \textbf{\textit{null}} \\ \text{proj1}\ (s\ (\text{Val}\ (e, s, v))) & \text{otherwise} \end{cases}$$

There is a notion of inheritance between the types in Java. When you define a class, you can assume that it extends another class. The aim of the function *archeType* is to return the inherited (i.e. extended) type of a given type. If the given type has no inherited type, the function returns *null*. The function is determined by the program to be analysed.
We assume that *ClassName* only contains the names of the classes declared in the program.

$$\text{archeType} : \text{ClassName} \rightarrow \text{ClassName} + \{\textbf{\textit{null}}\}$$

$$t \rightsquigarrow \begin{cases} t' \text{ tq } t \text{ extend } t' \\ \textbf{\textit{null}}\ \text{else} \end{cases}$$

In a program, there are several method calls. To treat them, we have to know at which statement we have to jump (the first statement of the called method). A method is identified by its name and by the name of the class in which it is defined. A same method can be redefined in a sub-class. The *getMeth* function takes a method as argument, and returns the label of the first statement of the called method.
The program is supposed correct and the method is supposed to exist.

getMeth	IMethName X ClassName → ILabel	
	(m, t)	~> p : first label of the method m in the class t

The *getConstr* function is similar to the *getMeth* function. It returns the label of the first statement of the constructor of the given type.

getConstr	ClassName → ILabel	
	t	~> p : first label in the constructor of the class t

In the case of a constructor call (***new*** statement), a new instance of the given type has got to be created. In this instance, all the fields have got the default value ***null***. When the instance is created, the statements in the constructor are applied on it. The function *newInst* creates a new instance of the given type and returns the associated value. The store is modified as it now associates the created instance to the new value.

newInst	Store X ClassName → ILoc X Store		
	(s, t)	~>	(l, s')
			where dom (s') = dom (s) U {l}
			l ∉ dom (s)
			s' (e) = < t, i >
			getType i = t
			∀ f ∈ dom (i) : i (f) = ***null***

4.3.1.3. Operational Semantics

Variable declaration

In the case of a simple declaration, only the environment is modified. A new local variable has got to be created. At the beginning, this local variable has got the default value ***null***. The environment is modified as it now associates ***null*** to the new variable. The new label (i.e. the following statement) is the one following the declaration.

$< p, P, (e, s) > \longrightarrow < q, P, (e[v/\textbf{\textit{null}}], s) >$

Where { p } ***var*** t v { q }

Affectation

There are two different cases for an assignment, depending on the descriptor of the modified variable (***field*** or ***var***). The new value of the variable is the one corresponding to the assigned value (that value is an instance because there are no basic types).

In the case of an assignment to a local variable, we only modify the environment. It associates the value of the assigned instance to the target variable. The new label is the one following the assignment.

$< p, P, (e, s) > \longrightarrow < q, P, (e[v_1/Val(e, s, v_2)], s) >$

Where { p } ***affect*** v_1 v_2 { q }

$v_1 \in$ VarName

In the case of an assignment in a field, we have to modify the "environment of the fields": the function of the instance associated to ***this***. The new function of the instance associates the value of the assigned instance to the modified variable. The new store is modified to replace the old instance of ***this*** by the new one. The new label is the one following the assignment.

$< p, P, (e, s) > \longrightarrow < q, P, (e, s') >$

Where { p } ***affect*** v_1 v_2 { q }

v1 $\in$ IFieldName

(t, e') = s(e ***this***)

e'' = e'[v_1/Val(e, s, v_2)]

s' = s[e(***this***)/(t, e'')]

*Statement "**if**"*

A ***if*** statement consists of a condition followed by two labels. The first one corresponds to the statement to carry out if the condition is evaluated to ***true***, the second label corresponds to the statement to execute if the condition is evaluated to ***false***.

In this work, we consider two kinds of condition.

The first one is the condition ***"istanceOf*** v t", which returns the value ***true*** when the variable is of the type t or a specialisation of the type t, ***false*** elsewhere.

The second kind of condition is the use of boolean constants c which are evaluated with an evaluation function ***C***. We consider this function as written. The function takes the boolean constant, the environment and the store as argument.

An ***if*** corresponds to a simple jump to the first or to the second label, following the fact that the condition is evaluated to ***true*** or ***false***. The only thing that will be modified in the state will be the label of the current statement.

InstanceOf condition evaluated to ***true***.

$< p, P, (e, s) > \longrightarrow < q, P, (e, s) >$

Where { p } ***if instanceOf*** v t { q } { r }

Type (e, s, v) ≤ t

InstanceOf condition evaluated to ***false***.

$< p, P, (e, s) > \longrightarrow < r, P, (e, s) >$

Where { p } ***if instanceOf*** v t { q } { r }

not (Type (e, s, v) ≤ t)

Boolean constant evaluated to ***true***.

$< p, P, (e, s) > \longrightarrow < q, P, (e, s) >$

Where { p } ***if*** c { q } { r }

C (c, e, s) where ***C*** is the evaluation function

Boolean constant evaluated to ***true***.

$< p, P, (e, s) > \longrightarrow < r, P, (e, s) >$

Where { p } ***if*** c { q } { r }

not ***C*** (c, e, s)

Method Call

The method/constructor calls and the ***return*** statements are treated using a stack. Each time we encounter a method or a constructor call, we add some information on the stack: the label of the statement following the call, the 'return variable' (with its descriptor: field/local variable) which receive the returned value and the current environment. This information is used when we encountered the ***return*** statement of the called method. It is useful to come back in the calling method and continue after the call statement, to find the current environment, the next statement to execute and to perform the assignment of the returned value into the return variable.

The new environment, in the called method, corresponds to an empty environment (or $\perp$) which has been updated with the local variable declarations corresponding to the parameters of the method and the assignments of the effective parameters to these formal parameters.

The new label corresponds to the label of the first statement of the called method.

If the return variable is a local variable:

$< p, P, (e, s) > \longrightarrow < r, <e, q, (\textbf{\textit{var}}, v\_ret)> : :P, (e', s) >$

Where $\{ p \}$ ***proc*** v_ret v m $v_1, v_2, ..., v_n \{ q \}$

$v\_ret \in VarName$

$r = getMeth (m, Type(e, s, v))$

$e' = \perp[\textbf{\textit{this}}/Val(e, s, v), u_1/Val(e, s, v_1), \ldots u_n/Val(e, s, v_n)]$

Where the u_i are the formal parameters of the method m

If the return variable is a field:

$< p, P, (e, s) > \longrightarrow < r, <e, q, (\textbf{\textit{field}}, v\_ret)> : :P, (e', s) >$

Where $\{ p \}$ ***proc*** v_ret v m $v_1, v_2, ..., v_n \{ q \}$

$v\_ret \in IFieldName$

$r = getMeth (m, Type (e, s, v))$

$e' = \perp[\textbf{\textit{this}}/Val(e, s, v), u_1/Val(e, s, v_1), \ldots u_n/Val(e, s, v_n)]$

Where the u_i are the formal parameters of the method m

*Constructor Call ("**new**")*

The case of a constructor call is similar to the case of a method call.

If the return variable is a local variable

$< p, P, (e, s) > \longrightarrow < r, <e'', q, (\textbf{\textit{var}}, v\_ret)> : :P, (e', s') >$

Where $\{ p \}$ ***new*** $v\_ret\ t\ v_1, v_2, ..., v_n \{ q \}$

$v\_ret \in VarName$

$r = getConstr\ (t)$

$s' = proj_2(newInst(s, t))$

$e'' = e\ [v\_ret/proj_1(newInst(s, t))]$

$e' = \perp[\textbf{\textit{this}}/\ proj_1(newInst(s, t)), u_1/Val(e, s, v_1), \ldots u_n/Val(e, s, v_n)]$

where the u_i are the formal parameters of the constructor of the class t

If the return variable is a field

$< p, P, (e, s) > \longrightarrow < r, <e'', q, (\textbf{\textit{field}}, v\_ret)> : :P, (e', s') >$

Where $\{ p \}$ ***new*** $v\_ret\ t\ v_1, v_2, ..., v_n \{ q \}$

$v\_ret \in IFieldName$

$r = getConstr\ (t)$

$s' = proj_2(newInst(s, t))$

$e'' = e\ [v\_ret/proj_1(newInst(s, t))]$

$e' = \perp[\textbf{\textit{this}}/\ proj_1(newInst(s, t)), u_1/Val(e, s, v_1), \ldots u_n/Val(e, s, v_n)]$

where the u_i are the formal parameters of the constructor of the class t

*"**return**" statement*

The case of a ***return*** statement corresponds to a simple assignment of the returned value into a variable. The "return" information is found on the top of the stack. The new environment is the one found in the stack, modified by the assignment of the returned value in the 'return variable'. The new label is the one found on the top of the stack. The top of the stack is removed from the stack.

In the case of a ***return*** in a constructor, the returned variable is the variable ***this***, which is the instance created and initialised by the constructor.

If the return variable is a local variable

$< p, <e', q, (\textbf{\textit{var}}, v\_ret)> : :P, (e, s) > \longrightarrow < q, P, (e'[v\_ret / Val(e, s, v)], s) >$

Where $\{ p \}$ ***return*** v

If the return variable is a field

$< p, <e', q, (\textbf{\textit{field}}, v\_ret)> : :P, (e, s) > \longrightarrow < q, P, (e', s') >$

Where $\{ p \}$ ***return*** v

$(t, linst) = s(e(\textbf{\textit{this}}))$

$linst' = linst[v\_ret/Val(e, s, v)]$

$s' = s[e(\textbf{\textit{this}})/(t, linst')]$

*"**super**" constructor call*

This last case is the case of the constructor call of the inherited class. This case is similar to the constructor call. The information added in the stack is the same in both cases, the environment is also modified in the same way. The new label corresponds to the label of the first statement of the called constructor.

The internal_v variable is a new internal local variable that is only used because we have to collect the variable ***this*** returned by the called constructor. As, when we call the constructor, we let the variable ***this*** as the current one, it will be directly modified, and we do not need the internal variable.

$< p, P, (e, s) > \longrightarrow < r, <e, q, (\textbf{\textit{var}}, internal\_v)> : :P, (e', s) >$

Where $\{ p \}$ ***super*** $v_1, v_2, ..., v_n \{ q \}$

$internal\_v \in VarName$

$t = archeType(Type(e, s, \textbf{\textit{this}}))$

$r = getConstr(t)$

$e' = e[u_1/Val(e, s, v_1), \ldots u_n/Val(e, s, v_n)]$

Where the u_i are the formal parameters of the constructor of the class t

4.3.2. Abstract Semantics

4.3.2.1. Definitions

We should first define an abstract domain. For the theoretical definition of the semantics, we have decided to define a generic abstract domain AType, which will be chosen later, at the implementation.

Here, we are defining an abstract state, in order to define a transposition from the concrete semantics to the abstract semantics.

Concrete state: $< p, P, (e, s) >$
Abstract state: $< p, P_a, (e_a, s_a), ncl, nm >$

AEtat = IPtsProg X Stack X (IEnv X Store) X ClassName X MethName

- We add the names of the current class and of the current method in the abstract state. This information is often useful, for example to get the abstract instance corresponding to *this* but is not accessible anymore. In an abstract state, we can only get the abstract type of *this*, not its concrete type.

- p: We decide not to do any abstraction on the labels. Thus we represent the labels by labels in the abstract syntax. Anyway, the number of labels is finite in a program.

- e_a: the abstract environment is a function which associates an abstract type (the dynamic type) to variable names (locale variable or the pseudo-variable *this*). The number of local variables and the number of possible abstract types are finite.

  $e_a \in$ AEnv: VarName + { ***this*** } → AType

- s_a: the abstract store takes a concrete type (the name of a class) as argument and returns an abstract instance (AInst), which is an aggregate of all the concrete instances of the given type.

  Obviously, the number of concrete types is finite. It is the number of classes defined in the program (without the main class).

  The number of abstract instances is finite, as there is only one instance by concrete type.

  An abstract instance and an abstract store are defined as follows:

  AInst: IFieldName → AType
  $s_a \in$ AStore: ClassName → AInst + { ***undef*** }

  The first time that a concrete instance of a certain type is created, we create the corresponding abstract instance. It corresponds to the abstract environment for the types. At the beginning, we set the abstract types of the different fields to a default value corresponding to $\perp$.

When we encounter an assignment of a value into a field, we do update the existing abstract instance. The new abstract type of a field is the union of its previous abstract type and the abstract type of the assigned value.

- P_a: We are going to gather all the information of the concrete stack associated to a method. In fact, we aggregate all the abstract environments corresponding to the method.
  We have to notice that what we consider as the abstract environment of a program is the abstract environment after the declarations of all the local variables of the program.

  All the information about the return points in the corresponding method (found in the concrete stack) is put into a list.

  We also keep the name of the class in which the method is defined. This information is useful to update the 'current class' field of the abstract state after a *return* statement.

  $P_a \in$ APile : (IMethName X ClassName) → (AEnv X ***P***(ILabel X TypeAppel)) + {***undef***}

  TypeAppel = {***field*** | ***var***} X Des

In this way, the size of the stack is limited to the number of different methods, and the size of the list in the items of the stack is limited to the number of label of the method.

We can notice that, in the abstract case, the stack is defined as a function, which takes a method name and a class name as argument (i.e. the identifier of a method). The method name is not an identifier as a method can be redefined in an inherited class. The function returns the abstract environment of the method and a list of information about return points (the label of the return point and the information on the variable that will get the returned value).

At the beginning, the function returns ***undef*** for all the method identifier.
The first time we encounter a method / constructor call in a given method, we create an entrance in the function for the method, adding all the information (abstract environment, ...). When we encounter others calls in the method, we aggregate the new abstract environment and the one that was already associated to the method. Then we add the information on the return variable in the set.

4.3.2.2. Useful Functions

We can first assume a function of abstraction on the types, which defines the correspondence between the sets ClassName and AType. The implementation of this function will depend on the choice of the abstract domain. We will do that choice later.

Abs	ClassName → AType
t ~>	the abstract type corresponding to the concrete type t

The *TypeAbs* function takes a designator as argument. It returns the abstract type of this designator. As the concrete environment associates values to variables and the abstract environment associates abstract types to designator, this function is implemented in the same way than the function *Val* in the concrete case.

TypeAbs AEnv X AStore X IDes X TypeCour → AType

$$(e_a, s_a, v) \sim> \begin{cases} e_a(v) & \text{if } v \in \text{VarName} + \{\textbf{\textit{this}}\} \\ (s_a\ \text{TypeCour})\ v & \text{if } v \in \text{IFieldName} \end{cases}$$

The *SousType* function just verifies if two abstract types are compatible. We only know the abstract dynamic type of the variables. This function is used in the case of a ***return*** statement. We will see that we do not have enough information in the abstract stack to find exactly the return point. We have to test most of the possible return points of the stack. To perform a first selection, we decided to test the compatibility between the return variable and the value returned by the method / constructor. The *SousType* function is used here to verify the compatibility between the two types.

SousType AType X AType → Boolean

$(t_1, t_2) \sim>$ true $\Leftrightarrow \exists\, t_3$ tq $t_1 \leq t_3 \ \Lambda\ t_2 \leq t_3$

The *NewAbsInst* function is similar to the *newInst* function in the concrete case. It is used in the case of a ***new*** statement. It returns a new abstract instance (*AInst*) of the given type. The returned *AInst* is initialised as follows: it associates the dynamic type *bottom* to all the field of the instance. *bottom* is the AType which corresponds to the concrete type of ***null***.

NewAbsInst	ClassName → AInst
t ~>	the new abstract instance of type t

The *UnionAbs* function implements the abstract union of two abstract types. This union corresponds to a superior limit of the two types in the abstract domain. Once again, the implementation of this function depends on the choice of the abstract domain.

UnionAbs AType X AType → AType

$(t_1, t_2) \sim> t$

We also use a function that aggregates two environments.
The two given environments must have the same domain. What does the function is creating a new environment on the same domain that the two given ones. For each variable in the domain, the new environment will return the abstract union of the two abstract types associated to the variable, by the two environments in entrance.

This function is used when we update the stack in a method/constructor call. We have seen that all the return information in an abstract stack is gathered for each method. The first time we encounter a method/constructor call in a method, we create an entrance in the abstract stack for this method (do not forget that the abstract stack is a function), using the current abstract environment.

When we encounter other method/constructor calls, we have to add the information on the return point in the list associated to the current method. The abstract environment in the stack is the union of the current abstract environment and the environment that already contained in the stack.

$$\begin{array}{lll} \text{UnionEnvAbs} & \text{AEnv X AEnv} \rightarrow & \text{AEnv} \\ (e_a, e_a') \sim> & e_a'' \mid & \mathrm{dom}(e_a'') = \mathrm{dom}(e_a') = \mathrm{dom}(e_a) \\ & & \Lambda\ \forall x \in \mathrm{dom}(e_a'') : e_a''(x) = \mathrm{UnionAbs}(e_a'(x), e_a(x)) \end{array}$$

The *archeType* function of the concrete case is still used in the abstract case, to get the inherited class in the case of a ***super*** constructor call.

4.3.2.3. The Concretisation Function

To create the link between the abstract and the concrete objects, we need to define either an abstraction or a concretisation function. Here, we chose to use a concretisation function. The concretisation function associates an abstract object to a set of concrete corresponding objects (state, environment, store, type…).

$$C_C (< p, P_a, (e_a, s_a), ncl, nm >) =$$
$$\{ < p, P, (e, s), ncl, nm > \mid (P, s) \in C_C (P_a) \wedge (e, s) \in C_C (e_a) \wedge s \in C_C (s_a) \wedge ncl = Type (e, s, \textbf{\textit{this}}) \}$$

The information on the current method and the current class has been added afterwards, after implementation. It is why it does not appear in the concrete states. It should be added there to be able to add some constraints in the concretisation function, which would define the 'nm' item in the abstract state.
We did not add it because we did not have the time to remake all the transition rules of the concrete semantics. The fact to add the name of the current method in the state obliges us to modify the concrete stack. We must add the name of the calling method in the stack to find the current method at the end of a method or a procedure call.

- A concrete store belongs to the concretisation of an abstract store if the instance associated to each variable in the concrete store belongs to the concretisation of the abstract instance associated to the concrete type of the variable, in the abstract store.

  The concretisation function for the instance is defined here with an intermediate function: the *AbsType* function. This function transforms a given concrete instance into a semi-concrete instance (CInst). A semi-concrete instance associates its concrete type to a field. It is a compromise solution between the concrete instance (IInstance: IFieldName → ILoc) and the abstract instance (AInst: IFieldName → AType).

  - The *AbsType* function is defined as follows:

    AbsType: IInstance X Store → CInst
    Where CInst = IFieldName → ClassName + {***null***} + {***undef***}

    $$AbsType\ (i, s) = i_c \text{ while } \begin{cases} i_c\ x = i\ x & \text{if } x \in \{\textbf{\textit{null}}, \textbf{\textit{undef}}\} \\ i_c\ x = proj_1(s\ (i\ x)) & \text{else (hyp : i = instance of s)} \end{cases}$$

  - Here we define the concretisation function for the instance. It transforms an abstract instance into a semi-concrete instance.

    C_C: AInst → ***P*** (CInst)

    $C_C (i_a) = \{ i_c \in CInst \mid dom (i_c) = dom (i_a) \wedge \forall x \in dom (i_c) : i_c\ x \in C_C (i_a\ x) \}$

  - The concretisation function for the stores is:

    C_C: Astore → ***P*** (Store)

    $C_C (s_a) = \{ s \in Store \mid \forall l \in dom (s) : AbsType\ (i, s) \in C_C (s_a\ t) \quad \text{where } s\ l = (t, i) \}$

The concrete types of the fields of the concrete instance have to belong to the concretisation of the abstract types (respectively) of the field of the abstract instance.

- The concretisation function for the environment is defined as follows:

  C_C: AEnv → IEnv X Store

  $C_C(e_a) = \{ (e, s) \mid dome(e) = dom(e_a) \quad \Lambda \quad \forall v \in dom(e) : Type(e, s, v) \in C_C(e_a(v)) \}$

  The concrete (found using the concrete environment and store) type of each variable has got to belong to its abstract type (returned by the abstract environment).

- The concretisation function for the abstract stack is:

  C_C: AStack → Stack X Store

  $C_C(P_a) = \{ (P, s) \mid \forall m \in IMethName, t \in ClassName : \forall < e, p, v > \in P \text{ tq } e \approx (m, t) :$
  $\exists e_a, t, L, tap \mid < e_a, L, t > \in P_a \Lambda < p, (tap, v) > \in L \Lambda (e, s) \in C_C(e_a) \}$
  where tap = type de v (***var*** or ***field***)

  In this definition, "$e \approx (m, t)$" means that e is an environment associated to the method m in the class t.

  For all triplet < environment e, label l, "return variable" v > of the concrete stack, the abstract stack associates to the pair method_name, class_name:
  - The abstract environment found with the concrete environment e
  - A list of labels / abstract type of the "return variable", which contains at least the pair (l, abstract type of v).

4.3.2.4. The Abstract Semantics

Variable declaration

In the case of a variable declaration, a new variable is created (added in the environment domain). Its initial value is the default value corresponding to the concrete type ***bottom***. In most of the abstract type domains, which are defined as sets of concrete types, it corresponds to an empty set. The environment is modified as follows: at the declared variable, the environment associates the abstract type corresponding to ***bottom***. For the other variables, the new environment associates the same value than the last environment.

$< p, P_a, (e_a, s_a),\ ncl, nm > \longrightarrow < q, P_a, (e_a\ [v / \textbf{\textit{bottom}}], s_a), ncl, nm >$

Where $\{ p \}$ ***var*** t v $\{ q \}$

Assignment

The new abstract type of the variable is simply the abstract type of the assigned value.

$< p, P_a, (e_a, s_a), ncl, nm > \longrightarrow$

$< q, P_a, (e_a\ [v_1 / \text{TypeAbs}\ (e_a, s_a, v_2)], s_a), ncl, nm >$

Where $\{ p \}$ ***affect*** v1 v2 $\{ q \}$

$v_1 \in \text{VarName}$

In the case of an assignment in a field, the instance corresponding to the current class is modified in the same way that the environment in the previous case.

$< p, P_a, (e_a, s_a), ncl, nm > \longrightarrow < q, P_a, (e_a, s_a'), ncl, nm >$

Where $\{ p \}$ ***affect*** $v_1\ v_2$ $\{ q \}$

$v1 \in \text{IFieldName}$

$i_a = s_a(ncl)$

$i_a' = i_a[v_1 / \text{TypeAbs}\ (e_a, s_a, v_2)]$

$s_a' = s_a[ncl / i_a']$

Statement "if"

In the case of the ***if*** statements, the treatment is similar to the concrete case. It is a jump to a certain label of the program, depending on the value of the condition.

Case of an ***instanceOf*** condition evaluated to ***true***

$< p, P_a, (e_a, s_a), ncl, nm > \longrightarrow < q, P_a, (e_a, s_a), ncl, nm >$

Where { p } ***if instanceOf*** v t { q } { r }

$\forall t' \in C_C$ (TypeAbs(e, s, v, ncl)): $t' \leq t$

Case of an ***instanceOf*** condition evaluated to ***false***

$< p, P_a, (e_a, s_a), ncl, nm > \longrightarrow < q, P_a, (e_a, s_a), ncl, nm >$

Where { p } ***if instanceOf*** v t { q } { r }

$\forall t' \in C_C$ (TypeAbs(e, s, v, ncl)): not *SousType* (t', t)

Case of an ***instanceOf*** condition evaluated to ***don't know***

$< p, P_a, (e_a, s_a), ncl, nm > \longrightarrow < q, P_a, (e_a, s_a), ncl, nm >$

where { p } ***if instanceOf*** v t { q } { r }

in the other cases

$x \in \{q, r\}$

Case of a boolean constant evaluated to ***true***

$< p, P_a, (e_a, s_a), ncl, nm > \longrightarrow < q, P_a, (e_a, s_a), ncl, nm >$

Where { p } ***if*** c { q } { r }

C (c, e_a, s_a)

where ***C*** is an evaluation function of the boolean constants

Case of a boolean constant evaluated to ***false***

$< p, P_a, (e_a, s_a), ncl, nm > \longrightarrow < q, P_a, (e_a, s_a), ncl, nm >$

Where { p } ***if*** c { q } { r }

not ***C*** (c, e_a, s_a)

where ***C*** is an evaluation function of the boolean constants

Case of a boolean constant evaluated to *don't know*

$< p, P_a, (e_a, s_a), ncl, nm > \longrightarrow < q, P_a, (e_a, s_a), ncl, nm >$

Where { p } ***if*** c { q } { r }

C (c, e_a, s_a) = *don't know*

where ***C*** is an evaluation function of the boolean constants

Method call

When we encounter a method call, we have to add the information into the abstract stack. The way information is added in this stack has already been explained after its definition.

In the abstract case, there are several following possible states, as a method can be redefined in a sub-class, and we do not have a definite information about the concrete type of the target variable.

There are two cases: the variable that will get the returned value is a 'local variable' or a 'field'. The only difference is that the information added in the stack is (***var***, var_id) or (***field***, field_id).

Case of a local variable

$< p, P_a, (e_a, s_a), ncl, nm > \longrightarrow < r, P_a', (e_a', s_a), ncl', m >$

Where $\{ p \}$ ***proc*** v_ret v m $v_1, v_2, ..., v_n \{ q \}$

$v\_ret \in VarName$

$ncl' \in C_C (TypeAbs (e_a, s_a, v))$

$r = getMeth (m, ncl')$

$e_a' = \perp[\textbf{\textit{this}}/Abs(ncl')),$

$u_1/ TypeAbs (e_a, s_a, v_1), \ldots u_n/ TypeAbs (e_a, s_a, v_n)]$

where the u_i are the formal parameters of the method m

$$P_a' = \begin{cases} P_a[(nm, ncl)/< e_a, \{<q,(\textbf{\textit{var}},v\_ret)>\} >] & \text{if } P_a (nm, ncl) = \textbf{\textit{undef}} \\ P_a[(nm, ncl)/<UnionEnvAbs (e_a, e_a\_init), \{<p_1,tap_1>, \ldots <p_n,tap_n>, <q,(\textbf{\textit{var}}, v\_ret)>\}>] & \\ \quad \text{if } P_a (nm, ncl) = <e_a\_init, \{ <p_1, tap_1>, \ldots <p_n, tap_n> \} > & \end{cases}$$

Case of a field

$< p, P_a, (e_a, s_a), ncl, nm > \longrightarrow < r, P_a', (e_a', s_a), ncl', m >$

Where $\{ p \}$ ***proc*** v_ret v m $v_1, v_2, ..., v_n \{ q \}$

$v\_ret \in IFieldName$

$ncl' \in C_C (TypeAbs(e_a, s_a, v))$

$r = getMeth (m, t)$

$e_a' = \perp[\textbf{\textit{this}}/TypeAbs(e_a, s_a, v)),$

$u_1/ TypeAbs (e_a, s_a, v_1), \ldots u_n/ TypeAbs (e_a, s_a, v_n)]$

where the u_i are the formal parameters of the method m

$$P_a' = \begin{cases} P_a[(nm, ncl)/< e_a,\{<q,(\textbf{\textit{field}},v\_ret)>\}>] & \text{if } P_a (nm, ncl) = \textbf{\textit{undef}} \\ P_a[(nm, ncl)/< UnionEnvAbs (e_a, e_a\_init), \{<p_1,tap_1>,\ldots<p_n,tap_n>, <q,(\textbf{\textit{field}}, v\_ret>)\}>] & \\ \quad \text{if } P_a (nm, ncl) = < e_a\_init, \{ < p_1, tap_1 >, \ldots < p_n, tap_n > \}> & \end{cases}$$

*Constructor call ("**new**")*

The case of a constructor call is similar to the case of a method call. The environment and the stack are modified in the same way. The new label corresponds to the first statement of the called constructor.

Case of a local variable

$$< p, P_a, (e_a, s_a), ncl, nm > \longrightarrow < r, P_a', (e_a', s_a'), t, t >$$

Where $\{ p \}$ ***new*** v_ret t $v_1, v_2, ..., v_n$ $\{ q \}$

v_ret $\in$ VarName

r = getConstr (t)

$e_a' = \perp[$***this***$/Abs(t), u_1/$ TypeAbs $(e_a, s_a, v_1), \ldots u_n/$ TypeAbs $(e_a, s_a, v_n)]$

where the u_i are the formal parameters of the constructor of the class t

$$s_a' = \begin{cases} s_a\,[t/NewAbsInst(s_a, t)] & \text{if } s_a(t) = \textbf{\textit{undef}} \\ s_a & \text{else} \end{cases}$$

$$P_a' = \begin{cases} P_a\,[(ncl, nm)/< e_a, \{< q, (\textbf{\textit{var}}, v\_ret) > \} >] \text{ if } P_a\,(ncl, nm) = \textbf{\textit{undef}} \\ P_a\,[(ncl, nm)/< \text{UnionEnvAbs } (e_a, e_a\_init), \\ \qquad \{<p_1,tap_1>,\ldots<p_n,tap_n>,<q, (\textbf{\textit{var}}, v\_ret)>\}>] \\ \qquad \text{if } P_a\,(ncl, nm) = < e_a\_init, \{ < p_1, tap_1 >, \ldots < p_n, tap_n > \} > \end{cases}$$

Case of a field

$$< p, P_a, (e_a, s_a), ncl, m > \longrightarrow < r, P_a', (e_a', s_a'), t, t >$$

Where $\{ p \}$ ***new*** v_ret t $v_1, v_2, ..., v_n$ $\{ q \}$

v_ret $\in$ IFieldName

r = getConstr (t)

$e_a' = \perp[$***this***$/Abs(t), u_1/$ TypeAbs $(e_a, s_a, v_1), \ldots u_n/$ TypeAbs $(e_a, s_a, v_n)]$

where are u_i are the formal parameters of the constructor of the class t

$$s_a' = \begin{cases} s_a\,[t/NewAbsInst(s_a, t)] & \text{if } s_a(t) = \textbf{\textit{undef}} \\ s_a & \text{else} \end{cases}$$

$$P_a' = \begin{cases} P_a\,[(ncl, nm)/< e_a, \{<q, (\textbf{\textit{field}}, v\_ret)>\} >] \text{ if } P_a\,(ncl, nm) = \textbf{\textit{undef}} \\ P_a\,[(ncl, nm)/<\text{UnionEnvAbs } (e_a, e_a\_init), \\ \qquad \{<p_1,tap_1>,\ldots<p_n,tap_n>,<q,(\textbf{\textit{field}},v\_ret)>\} >] \\ \qquad \text{if } P_a\,(ncl, nm) = <e_a,\{<p_1,tap_1>,\ldots<p_n,tap_n>\}> \end{cases}$$

*"**return**" statement*

The case of the ***return*** statement corresponds to an assignment of the returned value into the "return variable". The environment in this case is modified in the same way than for an assignment.

The problem is to find the "return information" in the stack. We don not have any information about the method we come from. So we have to test all the possible return labels of the stack.
A way to only test a part of them is to test the compatibility (i.e. to see if they have a common archetype in the hierarchy of concrete types) of the abstract type of the returned value and the abstract type of the variable that should receive it. If they are not, we don't consider that "return label".

In the abstract case, we do not have to remove an item of the list (in the concrete case, the first item of the stack is removed). Several successive calls from the same statement of a method (for example in a recursive method) is represented by only one label/"return information" in the stack.

The difference between the cases of the local variable and the field is the same than in the assignment.

Case of a value returned into a local variable

$$< p, P_a, (e_a, s_a), ncl, nm > \longrightarrow < q, P_a', (e_a'', s_a), ncl', nm' >$$

Where $\{ p \}$ ***return*** v

$$<e_a',q> \in \{<env,pts> \mid \exists\, nm' \in IMethName,\ ncl' \in ClassName$$
$$tq\ P_a(nm', ncl') = <env,list>$$
$$\Lambda\ \exists\ v\_ret \in IDes \mid < pts, (\textbf{\textit{var}}, v\_ret) > \in list$$
$$\Lambda\ SousType\ (TypeAbs(e_a,s_a,v), TypeAbs(e_a',s_a,v\_ret)\ \}$$

$$e_a'' = e_a'[v\_ret/\ UnionAbs(TypeAbs\ (e''_a, s_a, v\_ret), TypeAbs\ (e_a, s_a, v))]$$

$$P_a' = \begin{cases} P_a[(nm', cl') / \textbf{\textit{undef}}] & \text{if } P_a(nm', cl') = < e_a, \{< q, tap >\} > \\ P_a[(nm', cl') / < e_a, \{< p_1, tap_1 >, \ldots < p_n, tap_n >\} >] & \\ \quad \text{if } P_a(nm', cl') = < e_a, \{<p_1, tap_1>, \ldots <p_n, tap_n>, <q, (\textbf{\textit{var}}, v\_ret)>\}> & \end{cases}$$

Case of a value returned into a field

$< p, P_a, (e_a, s_a), ncl, nm > \longrightarrow < q, P_a', (e_a'', s_a'), ncl', nm' >$

Where $\{ p \}$ ***return*** v

$<e_a',q> \in \{<env,pts> \mid \exists\ nm' \in IMethName, cl' \in ClassName$

$tq\ P_a\ (nm', cl') = <env, list>$

$\Lambda\ \exists\ v\_ret \in IDes \mid <pts,(\textbf{\textit{field}},v\_ret)> \in list$

Λ SousType (TypeAbs(e_a,s_a,v),TypeAbs(e_a',s_a,v_ret) $\}$

$(typ, i_a) = s_a(ncl')$

$i_a' = i_a[v\_ret/\ UnionAbs(TypeAbs\ (i_a, s_a, v\_ret), TypeAbs\ (e_a, s_a, v))]$

$s_a' = s_a[ncl'/(typ, i_a')]$

$e_a'' = e_a'$

$$P_a' = \begin{cases} P_a[(nm', cl') / \textbf{\textit{undef}}] & \text{if } P_a\ (nm', cl') = < e_a, \{< q, tap >\} > \\ P_a[(nm', cl') / < e_a, \{< p_1, tap_1 >, \ldots < p_n, tap_n >\} >] & \\ \quad \text{if } P_a(nm', cl') = <e_a,\{<p_1,tap_1>,\ldots<p_n,tap_n>, <q,(\textbf{\textit{field}},v\_ret)>\}> & \end{cases}$$

*"**super** constructor call*

The call to the ***super*** constructor can be considered as a simple method call and can be treated in a similar way. It does not work as a constructor call as the instance has already been created (it is the current instance) and does not have to be created anymore.

$< p, P_a, (e_a, s_a), ncl, nm > \longrightarrow < r, P_a', (e_a', s_a), ncl', ncl' >$

Where $\{ p \}$ ***super*** $v_1, v_2, ..., v_n \{ q \}$

internal_v $\in$ VarName

ncl' = archeType(ncl)

r = getConstr (ncl')

$e_a' = e_a\ [u_1/\ TypeAbs\ (e_a, s_a, v_1), \ldots\ u_n/\ TypeAbs\ (e_a, s_a, v_n)]$

where the u_i are the formal parameters of the constructor of the class ncl'

$$P_a' = \begin{cases} P_a[(ncl, nm)/< e_a, \{<q,(\textbf{\textit{var}},internal\_v) >\},TypeCour() >] & \\ \quad \text{if } P_a\ (ncl, nm) = \textbf{\textit{undef}} & \\ P_a[(ncl, nm)/<UnionEnvAbs\ (e_a, e_a\_init), & \\ \quad \{<p_1,tap_1>, \ldots <p_n,tap_n>, <q,(\textbf{\textit{var}},internal\_v)>\}>] & \\ \quad \text{if } P_a\ (ncl, nm) = < e_a\_init, \{<p_1,tap_1>, \ldots<p_n,tap_n>\}> & \end{cases}$$

4.3.3. Correctness Proof of the Rules

Here, we only develop the proof of one rule to give an example. The other rules are proved in a similar way.

4.3.3.1. Reasoning

The proof of the abstract rules use this reasoning:

At a concrete level, the rule characterises the passage from the state "c" to the state "c' ":

$c \longrightarrow c'$

At an abstract level, the passage is from the state "a"

To the state "a' ": $a \longrightarrow a'$

Or to the states $\{a_1 \ldots, a_n\}$: $a \longrightarrow \{a_1', \ldots, a_n'\}$ (case of a ***return***, method call...)

If we take as hypotheses that: $c \in C_C(a)$

I.e. $c = < p, P, (e, s) >$ and $a = < p', P_a, (e_a, s_a) >$

$p = p'$

$\Lambda \quad (e, s) \in C_C(e_a)$

$\Lambda \quad s \in C_C(s_a)$

$\Lambda \quad (P, s) \in C_C(P_a)$

We have to show that: $c' \in C_C(a')$

Or that: $\exists a_i' \in \{a_1' \ldots, a_n'\} \mid c' \in C_C(a_i')$

- During all the proof, we have:

  $c = < p, P, (e, s) >$

  $\Lambda \quad a = < p, P_a, (e_a, s_a) >$

- In our proofs, when it is written that something is proven 'by hypothesis', it means that it is a consequence of the fact that $c \in C_C(a)$.

4.3.3.2. Proof

The case we are developing here is the case of a variable declaration. We chose a basic case as the aim here is not to convince you of the correctness of the rules (this correctness has already been developed in *[ZAP00b]*), but to give you an example of the reasoning we used. This example is quite easy to understand, without additional explanation.

{p} var ***t v {q}***

$c' = < q, P, (e', s) >$ where $e' = e\,[v/\textbf{\textit{undef}}]$
$a' = < q, P_a, (e_a', s_a) >$ where $e_a' = e_a\,[v/\textbf{\textit{bottom}}]$

$c \in C_C(a) \Rightarrow ? \; c' \in C_C(a')$

- $q = q$ ok

- $(e', s) \in ? \; C_C(e_a')$
  $\Leftrightarrow \forall v_i \in dom(e) : Type(e', s, v_i) \in C_C(e_a'(v_i))$
  $\Leftrightarrow \forall v_i \neq v : Type(e', s, v_i) \in C_C(e_a'(v_i))$
  $\Lambda \quad Type(e', s, v) \in C_C(e_a'(v))$
  $\Leftrightarrow \forall v_i \neq v : Type(e, s, v_i) \in C_C(e_a(v_i))$ ok by hypothesis $(c \in C_C(a))$
  $\Rightarrow (e, s) \in C_C(e_a))$
  $\Lambda \quad \textbf{\textit{undef}} \in C_C(\textbf{\textit{bottom}})$ ok by definition of the concretisation function on the types

- $s \in ? \; C_C(s_a)$ ok by hypothesis

- $(P, s) \in ? \; C_C(P_a)$ ok by hypothesis

The others proofs can be found in the annexes of this work. They are enough detailed to be understood.

4.4. Implementation

4.4.1. The Simplified Language

We decide to make the implementation of the static analyser into the CaML language. In order to make some test-programs, it is very important for us to create a translator between the *VSS* and the CaML representation of the abstract structures hidden behind the Java code. Therefore it is also important to find a language easier to translate than the real Java language. We need to invent a language definition that would allow us to translate the *VSS* code directly line by line into CaML functions. We call this intermediate language: the ***S***implified ***L***anguage, witch we will from now on call *SL*. In the *SL* definition we write, we use a little artefact for an easier recognition of the class declarations, the statements and the comments. Therefore we decide to begin every line with a number. This number is made of two figures. That way we can easily recognise the different sort of lines, this is done for a rapid and easy parsing and translating of the ***SL*** into the CaML functional representation. Here we also can see that it is not always necessary to use some pre-made tools like JavaCC, *lex* or *yacc* in order to create a parser. In fact, thanks to the little artefact we use, it is possible to recognise every line by reading the tokens one by one. For the reading of the tokens we created a little function that reads a string until it crosses a white space, so we do not need to use a pre-made lexical analyser either. We easily cut the input file into tokens with this function. The system we use, is the following: Every variable declaration begins with the number "01", every field declaration with the number "02"...

Here is the syntax of the *SL*, every example we give could be one of the lines of the *SL*-program we give to the "mini_parser" as argument.

- ***01: a field declaration***

  Example: 01 String field1

- ***02: a variable declaration***

  Example: 02 1 String variable1 2

  In this example and for the rest of this document we mark the labels in red, this is only done for a didactic reason. Here we have got a "String" typed "variable1".

- ***03: an affectation statement***

  Example: 03 2 c field1 v variable1 3

  We affect « variable1 » into « field1 ».

- ***04: a constructor-call statement***

  Example: 04 3 v variable1 String c field1 v variable2 END 4

  The assigned variable is "variable1", the constructor name is "String" and the string "c field1 v variable2 END" represents a list of effective parameters, in this case a field and a variable. We decide to represent every list as follows: *item1 item2 item3 END*. The keyword END is the representation of the end of the list and has got the same function as the ***::[ ]*** of the CaML language.

- ***05: a procedure-call statement***

  Example: 05 4 v variable1 v variable2 method1 v variable3 v variable4 END 5

  VSS: *variable1 = variable2.methode1 (variable3, variable4);*

- ***06: an if statement***

  Example: 06 5 condition 6 15

  The label that refers to the if part of the statement is the label 6 and the label that refers to the else part of the statement is the label 15.

- ***07: a super-call statement***

  Example: 07 6 v variable1 v variable2 v variable3 END 7

  VSS: *super (variable1, variable2, variable3);*

- ***08: a return statement***

  Example: 08 7 v variable1

  VSS: *return (variable1);*

- ***09: a constructor declaration***

  Example: String Boolean v variable1 Boolean v variable2 END

  VSS: *String (Boolean variable1, Boolean variable2)*

- ***10: a method declaration***

  Example: String Method1 Boolean v variable1 END

  VSS: *String Method1 (Boolean variable1)*

- ***11: a main method declaration***

  Example: 11

  There is, of course, a list of statements in the main method, but this list is created automatically with the lines that are written above this one. Before a line containing the number "11", there must at least be a number of declarations, statements and so on.

- ***12: a class definition***

  Example: Matrix Array

  Here we create the declaration of the class "Matrix" witch extends the class "Array".

- ***13: "Main Class" definition***

  Example: 13

  This class is created automatically.

- ***14: a program declaration***

  Example: 14

  Like for the Main Class, the program declaration is created automatically.

- ***// : a comment line***

  Example: // this is how a comment looks like
  //**********//
  // *** This could be a comment ***//
  //**********//

Once we have the *SL*, it is easier for us to make some test-programs and thus it is easier to test the correctness of the syntactical analyser. Instead of writing test-programs in CaML, we have to write them in SL and use a direct translator. The next step of our work is then writing this direct translator: the *SL-CaML-Translator*. Of course the *SL* is not that intuitive so it is not possible to write a program directly into *SL*. We first have to write them into *VSS* and then we have to make a hand-translation of it into *SL*. Once this translation is done *SL-CaML-Translator* translates the program in CaML.

4.4.2. The SL-CaML-Translator

The *SL-CaML-Translator* takes a text file as argument. This text file is the representation of the *SL*-program. The translator returns a text file of the CaML representation of the abstract structures of the program. The abstract syntax types of the program are defined below. They are used like defined in the sub-section about the ***S**implified **L**anguage*. The numbers 01 to 14 are used for DeclChamp, Instr, DeclConstr, DeclMeth, DefClasse, MethMain, MainClasse and Prog.

01	➳	DeclChamp	= Field declarations.
02 to 08	➳	Instr	= Statements or variable declarations.
09	➳	Dconstr	= Constructor declarations.
10	➳	DeclMeth	= Method declarations.
11	➳	MethMain	= Main methods.
12	➳	DefClasse	= Class definitions.
13	➳	MainClasse	= Main Class definitions.
14	➳	Prog	= Program declarations.

```
type NomClasse = noType | NCl of string;;type NomMeth = NM of string;;

type PtProg = noLabel | Pt of int;;

type Des = this | NV of string | NCh of string | null;;

type Expr = DES of Des;;

type Cond = C of bool | InstOf of (Des * NomClasse);;

type Instr = DVar of (PtProg * NomClasse * Des * PtProg)
      | affect of (PtProg * Des * Expr * PtProg)
      | new of (PtProg * Des * NomClasse * Expr list * PtProg)
      | proc of (PtProg * Des * Des * NomMeth * Expr list * PtProg)
      | ifInstr of (PtProg * Cond * PtProg * PtProg)
      | rien
      | super of (PtProg * Expr list * PtProg)
      | return of (PtProg * Expr);;
```

In the statements, the value 'rien' is used in a constructor, to replace the super statement, if the constructor does not contain one.

```
type DeclConstr = DConstr of (NomClasse * (NomClasse * Des) list
                                                    * Instr list);;

type DeclMeth = DMeth of (NomClasse * NomMeth * (NomClasse * Des) list
                                                    * Instr list);;

type DeclChamp = DChp of (NomClasse * Des);;

type DefClasse = DClass of (NomClasse * NomClasse * DeclChamp list *
                                          DeclConstr * DeclMeth list);;

type MethMain = main of ((NomClasse * Des) list * Instr list);;

type MainClasse = mainClass of MethMain;;

type Prog = Prog of DefClasse list * MainClasse;;
```

4.4.3. Multivariant Algorithm

The implementation of the analyser corresponds to the implementation of an algorithm. This algorithm takes an abstract state of the program and creates the set of the accessible abstract states beginning at the given state. Of course, depending on the algorithm and the definition of the abstract state, you will get a more or less precise information. You choose an algorithm following the degree of precision you want to get for your analysis.

Some algorithms create all possible states, it is more precise than programs that aggregate the states they find (approximations), but the number of states of a program can be very huge. The analysis can become hard. Other algorithms just gather some states together, doing approximations. The information you get is less precise, but the number of created states is less important, you can gain in memory place and in analysis time.

The definition of the abstract objects is important too. If there is a too large approximation in the definition, it is possible that, for some translation rules, to many states are created. It is the case in our work. In the case of return statement, we do not keep enough information in the abstract stack to find the calling method, so we have to test all the possible return points of the stack. A lot of useless states are created. We could add some information in the stack to limit the numbers of bad states.

An important point in the definition of the abstract objects is to compare the gains and the losses adding some information in the states. It can cost a lot in memory place and in treatment time, but you can sometimes gain a lot too.

For the choice of the algorithm, we had to choose between the two algorithms we had studied in the course of 'Interpretation Abstraite' of Mr Le Charlier (*[INFO3105]*): the univariant and the multivariant algorithm. It is possible to find or create other algorithms.

As this work was to test on small programs, we decided to implement the multivariant algorithm. This algorithm returns more precise information, but the number of states created is really more important. If this algorithm is used on large programs, it could be that the number of created states is two large to be handled.

Here is a more detailed explanation of this algorithm:

S: the states to develop
R: the states already developed

Initialisation:
$S := \{< p_0, P_0, (e_0, s_0) >\}$; $R := \{\}$;

While $S \neq \{\}$ do
Choose $< p, P, (e, s) > \in S$;
$S := S \setminus \{< p, P, (e, s) >\}$;
$R := R \cup \{< p, P, (e, s) >\}$;
$S := S \cup (\{< p', P', (e', s') > : < p, P, (e, s) > \longrightarrow < p', P', (e', s') >\} \setminus R)$

The code we have created is a direct translation of this algorithm into the code CaML.

4.4.4. Abstract Domain

Our abstract domain AType can be seen as the set ClassName itself, as there is a one-to-one transformation between the abstract domain and the type domain. At an abstract type corresponds the set containing this concrete type and all the concrete types that inherit from this type.

At a certain point of the program, the abstract type of a program represents its dynamic type.

If a variable has got a certain concrete declaration type, the types of the values that can be assigned to that variable inherit from the declaration type.

Here are some mathematical definitions:

- Type = ClassName + {***null***}
- AType = ClassName + {*bottom*}

Here is a recursive definition of the concretisation function for the types:

C_C : AType → ***P***(Type)

$t \rightsquigarrow \{ t' \text{ tq } t' \leq t \}$ if $t \neq bottom$

null if $t = bottom$

Here is the definition of the union of two abstract types:

$$\forall t_a, t_a' \in A : AbstractUnion(t_a, t_a') = t_a'' \mid t_a'' \in A \ \Lambda \ t_a \leq t_a'' \ \Lambda \ t_a' \leq t_a'' \ \Lambda \ (\forall t_a''' \in A \mid t_a \leq t_a''' \Lambda t_a' \leq t_a''' : t_a'' \leq t_a''')$$

We can here defined the union of two abstract types (*AbstractUnion*):
In the abstract case, the union of two abstract types is the more specialised type that is inherited by the two given abstract types.

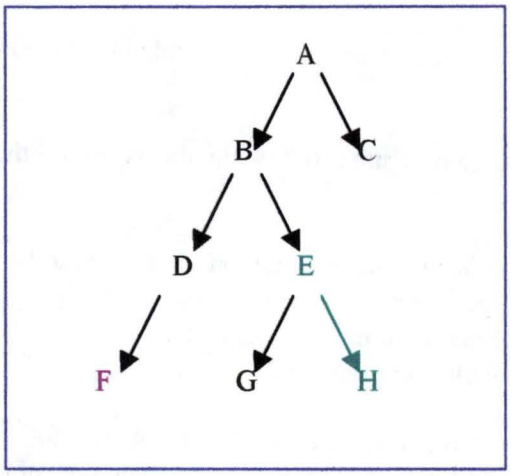
Another function that has got to be defined here is the *SousType* function. The concrete type structure can be seen as a forest of different trees of types. We consider that two types are compatibles if they belong to the same tree. In the abstract domain, it is translated by the fact that there exists an abstract type that is the abstract union of the two given abstract types:
$SousType(t_a, t_a') \Leftrightarrow \exists t_a'' \in A \mid t_a'' = AbstractUnion(t_a, t_a')$

4.4.5. The Analyser

Architecture of the implementation in CaML

- Definition of the types related to the abstract syntax
- Definition of the types related to the abstract states
- General functions on lists / functions (functions *appartient*, *taille*... / *update*...)
- Hierarchical structure of the types:

  Tree structure:
  - Leaf: concrete type
  - Node: concrete type + tree list (the types which extend the current concrete type)



Im 4.4: hierarchical structure of the types

Useful function: addition of a type, a function that get all the *ATypes* from the structure (that get the A domain).

- Structure for the information about the method and the constructor calls: this information is useful to update the new label and the new environment during the treatment of a method call.

  The structure is a n_uplet list with:

  For a method:
  - The method name
  - The label of the first statement of the method
  - The list of the names of the formal parameters
  - The name of the class in which the method is defined. This information is useful to update the "current type" field in the state after the method call.

  For a constructor:
  - Class name (= name of the constructor)
  - Label of the first statement of the constructor
  - List of the names of the formal parameters

Function to add a method or a constructor in the table, to research some information using a method name or a constructor name.

- Structure of the instances: we keep the list of the field names and types of each class.

  Pair list:
  - (field type, field name)

  Function to add a field in the list.

- Function to create all the structures: it takes a program as argument, and it creates the type structure, the instance structure, the table for the method and the constructor calls, and a labelisation function (function that associates the corresponding statement to a label).

  creer_struct : Prog -> Branchement list * TypeArbre list * ClasseStruct * (PtProg -> Instr)

  - Arguments:
    - A program written following the types of the abstract syntax defined before.
  - Results:
    - The table with the information for the method and the constructor calls
    - The type structure of the program
    - The structure of the instances (fields)
    - The labelisation function

- The functions defined in the 3.3.2.2 point (useful functions for the abstract semantics). These functions are used by the functions of the algorithm.

- Implementation of the algorithm:
  - Implementation of the different rules of transition:

    Rule: xxxxx -> AEtat -> AEtat list
    Where AEtat == `PtProg * APile * AEnv * AStore`

    Arguments:
    - Different structures (xxxxx): structures needed, chosen between all the structures created.
    - The initial state

    Results:
    - The list of the final states

    The function that implement the rules are:
    - regle_dvar
    - regle_affect
    - regle_new
    - regle_if
    - regle_super
    - regle_return

- etats_suiv: function that, using all the transition rules, implement a general transition:

```
etats_suiv :
        AEtat -> Branchement list * TypeArbre list * ClasseStruct * (PtProg -> Instr)
                                                    -> AEtat list
```

Arguments:
- The initial state
- The table with the information for the method and the constructor call
- The types structure
- The instances structure
- The labelisation function

Results:
- The list of the different states directly accessible from the initial state.

- algo_mult: implements the multivariant algorithm

```
algo_mult : Branchement list * TypeArbre list * ClasseStruct * (PtProg -> Instr)
                -> AEtat list -> AEtat list -> AEtat list
```

Arguments:
- The table with the information for the method and the constructor call
- The types structure
- The instances structure
- The labelisation function
- The list of the abstract states to treat (the set S in the algorithm)
- The list of the abstract states already treated (the set R in the algorithm)

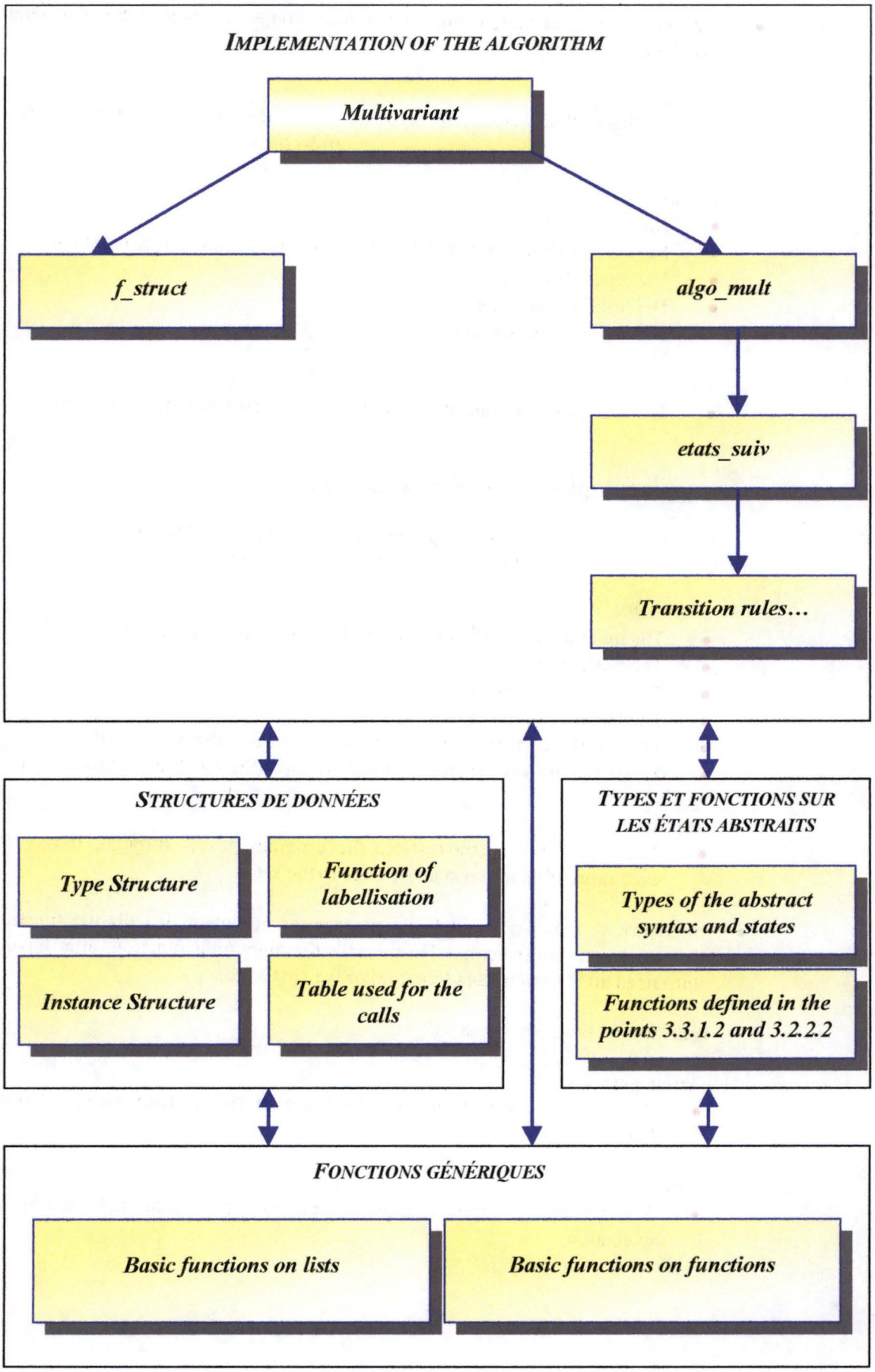
Results:
- The list of the abstract states the program passes through, during an execution, if its initial state belongs to the set S.

- multivariant: that function takes a program as argument. It calls the function that creates all the structure. Then it calls the algo_mult function, after having initialised all the arguments (creation of the first state, ...)

```
multivariant : prog -> AEtat list
```

Arguments:
- A program written following the types of the abstract syntax defined before.

Results:
- The list of the abstract states the program passes through, during an execution.



Im 4.5: Architecture of the implementation in CaML

4.4.6. Left to do

There are still a few bugs we have to correct in the CaML program we have written. We have to test it too. At the moment, it only works on very simple examples. The program has problem to treat ***return*** statements and to come back after a method call. It also returns invalid abstract stores.

Some improvements are scheduled too:

- Our program doesn't allow the redefinition of a method in a sub-class, as a method is only defined by its name in the stack. We have to modify the stack to use, as identify of the methods, their name and the name of the class in which they are defined.
- It could be interesting to create functions that filters the results returned by the *multivariant* function, to help the user in his analysis

4.4.7. Test Programs

4.4.7.1. Translation of a Java program into its *VSS* form

The translation of a Java program into its *VSS* form includes a lot of transformations. This is due to the fact that our *VSS* language has a lot of constraints. Some information of the Java program can be translated to stay in the *VSS* program, but sometimes, it is impossible to translate and we have to delete some information.
Some instances of transformation are the addition of a constructor in all classes and the addition of a ***return*** statement in all methods and constructors, the creation of a empty class ***void*** as this type is not defined in *VSS*.

Here is a simple example of translation:

Java Example

```
public class Test1
{
        public void Test1( )
        {

        }
}

public class ClasseMain
{
 public static void main (String args [])
  {

   var Test1 t1 = new Test1 ( );

   t1 = null;


  }
}
```

VSS Translation of Java Example

```
class Test1
{
        Test1( )
        {
        1 return this; 2
        }
}

class ClasseMain
{
  void main ( )
    {
    init
    2 var Test1 t1; 3
    3 t1 = new Test1 ( ); 4
    4 t1 = null;  noLabel
    end

    }
}
```

As you can see we have removed all the access modifiers of the Java example in order to translate it into the ***VSS***. You can also see that we have added the labels to the statements and the declarations. We have, indeed, also split one declaration combined with an assignment into two statements. And we have grouped all the variable declarations.

This way we have translated the small example into the following text:

```
class Test1
        {
                Test1( )
                {
                1 return this;
                }
        }

class ClasseMain
        {
                void main ( )
                {
                init
                2 var Test1 t1; 3
                3 t1 = new Test1 ( ); 4
                4 t1 = null;        noLabel
                end
                }
        }
```

We can now translate this text into the CaML form representing the abstract syntax. In order to translate this text we first have to translate the *VSS* program into a *SL* equivalent. Once this is done we have got to use the *SL-CaML-Translator* which does the rest.

4.4.7.2. Translation of a *VSS* Program into its CaML Form

First, we have to write the program into its *SL* form. This form is less intuitive than the Java form, but it is still possible to write it by hand.

```
08 1 this this                          // Return statement
09 Test1 END                            // Constructor declaration
12 Test1 notype                         // Class declaration

02 2 Test1 t1 3                         // Variable declaration
04 3 v t1 Test1 END 4                   // Constructor call
03 4 v t1 null null nolabel             // Assignment
                    // be careful, null is always written this way (null null) in SL
11                                      // Main Method declaration
13                                      // Main Class declaration
14                                      // Program declaration
```

Afterwards, we use the translator to translate the ***SL*** program into its CaML form, which is not intuitive at all. The translator returns the following results:

```
let Instr1 = return (Pt(1) , ( DES (this)));;
let Constr1 = DConstr (NCl ("Test1"), [], [Instr1]);;

let DClasse1 = DClass (NCl ("Test1"), NCl ("notype"), [], Constr1, []);;

let Instr2 = DVar ( Pt(2) , NCl ("Test1") , NV ("t1") , Pt(3));;
let Instr3 = new (Pt(3) , NV ("t1"), (NCl ("Test1")), [] , Pt(4));;
let Instr4 = affect ( Pt(4) , NV ("t1") , ( DES (null))  , Pt(nolabel));;
let MethMain = main ( [], [Instr2; Instr3; Instr4]);;

let MainClasse = mainClass ( MethMain );;

let Programme = Prog ([DClasse1], MainClasse );;
```

Once the program is translated into the *SL*, we can use the analyser to make the analysis on the program. We have done this for this little test program.

4.4.7.3. Analyse of a Program

Here are the results of the analysis of a very simple program by our analyser. The analyser returns all the states the program goes through.

The *VSS* program is:

```
class Test1
{
        Test1( )
        {
        1 return this;    noLabel
        }
}

class ClasseMain
{
        void main ( )
        {
        init
        2 var Test1 t1; 3
        3 t1 = new Test1 ( ); 4
        4 t1 = null;      noLabel
        end
        }
}
```

Its CaML form is:

```
PRO: Prog =
 Prog
  ([DClass
    (NCl "Test1", noType, [],
     DConstr (NCl "Test1", [], [rien; return (Pt 1, DES this)]), [])],
   mainClass
    (ClasseMain
     ([],
      [DVar (Pt 2, NCl "Test1", NV "t1", Pt 3);
       new (Pt 3, NV "t1", NCl "Test1", [], Pt 4);
       affect (Pt 4, NV "t1", DES null, noLabel)])))
```

What the *multivariant* function returns is:

```
[Pt 4, ([NM "main"], <fun>), ([NV "t1"], <fun>), <fun>, NCl " ClasseMain ",
  NM "main";
  Pt 1, ([NM "main"], <fun>), ([this], <fun>), <fun>, NCl "Test1", NM "Test1";
  Pt 3, ([], <fun>), ([NV "t1"], <fun>), <fun>, NCl " ClasseMain ", NM "main";
  Pt 2, ([], <fun>), ([], <fun>), <fun>, NCl " ClasseMain ", NM "main"]
```

The store, the environment and the stack functions do not appear clearly in the results. We have to catch these functions and to catch them to know the values they returned.

After a few tests, we can see that what the *multivariant* function returned is this list of states:

- State 1:
  - Label: 4
  - Stack:
    - Domain: NM "main" i.e. the method *main*
    - stack(*main*)=pile_info(([NV "t1"], <fun>), [Pt 4, NV "t1"], NCl "ClasseMain")
      return environment (abstract env of the method): domain {t1}, func
      call labels: Label 4, return variable t1
      class of the method: *ClassMain*
  - Environment:
    - Domain: [NV "t1"] i.e. {t1}
    - Environment (t1) = LST [] i.e. abstract type ***bottom***
  - Store: (the domain of the store is the set of all the defined classes)
    - Store (Test 1) = ***no_inst***
  - Current class: *ClasseMain*
  - Current method: *main*

- State 2:
  - Label: 1
  - Stack:
    - Domain: []
  - Environment:
    - Domain: {***this***}
    - Environment (***this***) = LST [NCl "Test1"] i.e. abstract type *Test1*
  - Store:
    - Store (Test 1) = ***no_inst***
  - Current class: *Test1*
  - Current method: *Test1*

- State 3:
  - Label: 3
  - Stack:
    - Domain: []
  - Environment:
    - Domain: {t1}
    - Environment (t1) = ***bottom***
  - Store:
    - Store (Test 1) = ***no_inst***
  - Current class: *ClasseMain*
  - Current method: *main*

- State 4:
  - Label: 2
  - Stack:
    - Domain: []
  - Environment:
    - Domain: []
  - Store:
    - Store (Test 1) = ***no_inst***
  - Current class: *ClasseMain*
  - Current method: *main*

As this example is very simple, there is nothing to analyse in these results. The aim here was to show you the information returned by out program.

5. CONCLUSION

5.1. Summary

Our thesis consists of the creation of a compiler and an analyser for subsets of the programming language Java.

We have built the compiler in the framework of a larger project, during our internship in Venice – Italy. We have written the compiler in Java. We have created it for a quite large sub-language of Java: the *VTF* (*Vas-T'y-Frotte*). The compiler is composed of two main parts: a parser and a type checker. The compiler creates the abstract representation of a given program. Doing this, it creates the Java objects corresponding to the *LAS*.

We have also written an analyser for a smaller subset of Java. We have implemented this one in CaML. When we want to use this analyser, we suppose that the given program has already passed through a compiler. In fact, we do this compiling work without any compiling tool, and then we use a simple translator to get the accurate CaML structures. The analyser implements the *multivariant* algorithm: it creates all the possible states of the given program.

5.2. Critics

It took us, first, a lot of time to learn the Java language, which was about new for us. We also lost lots of time learning the bases of the abstract interpretation. We would have needed some more time to finish and to test the compiler we created in Venice – Italy.

It is a pity that we attempted the course about the abstract interpretation after our internship. We would have lost less time trying to understand all the documentation we found, as our knowledge in the subject was about null.

5.3. Future work

The analyser we made in the context of our course was meant as a simple example. It is important not to begin directly with a complete analyser. It would have been to difficult to implement a large analyser, with lots of details and subtleties of a complete language. The simple analyser could be a good starting point for latter work. Now, it would be interesting to continue the project in which we created our parser and our type checker. Once the type checker will be finished, we could imagine creating an analyser. This analyser would be implemented in Java, and would be quite more complex than the little one we wrote in CaML.

It is already scheduled that two other students leave next year to a university in the United States to work on the continuation the whole project.

6. BIBLIOGRAPHY

[LC99a]: LE CHARLIER, Baudouin. *Définition du langage Vas-T'y-Frotte*,
Facultés Universitaires Notre-Dame de la Paix, Namur – Belgium, Institut d'informatique,
Notes de cours, Mars 1999

[IPO99]: POLLET, Isabelle.
Sémantiques operationnelles et domaines abstraits pour l'analyse statique de Java,
Facultés Universitaires Notre-Dame de la Paix, Namur – Belgium, Institut d'informatique,
Memoire de DEA (Diplôme d'Etudes Approfondies), Septembre 1999

[ZAP00b]: HAYEZ, Cécile and HENDRICKX, Patrick. *Interprétation Abstraite*,
work related to the course of "Interprétation Abstraite" course of the third maîtrise in information sciences

[INFO3105]: LE CHARLIER, Baudouin. Course: *Interprétation Abstraite*,
Facultés Universitaires Notre-Dame de la Paix, Namur – Belgium, Institut d'informatique,
Academic year 1999- 2000

[JLS96]: GOSLING, James, JOY, Bill and STEELE, Guy. *Java Language Specification*,
Addison – Wesley, Java Series, 1996,
Download this book at: http://www.java.sun.com/docs/books/jls/ -

[InriaJavaccEx]: Inria, Javacc Documentation. (page consulted in July 2000).
URL: http://falconet.inria.fr/~java/tools/JavaCC/examples/

[JavaCC1]: Javacc Documentation. (page consulted in July 2000)
URL: http://www.cs.um.edu.mt/~java/javacc-docs/DOC/index.html

[JavaCC2]: Javacc Documentation. (page consulted in July 2000)
URL: http://www.kluge.net/mqp/report.html

[LexYacc]: Lex - Yacc Documentation. (page consulted in July 2000)
URL: http://www.combo.org/lex_yacc_page/

[SA98a]: AMARASINGHE, Saman. Lecture 2: *Lexical Analysis*, September 1998
URL: http://ceylon.lcs.mit.edu/6035/lecture2/index.htm

[SA98b]: AMARASINGHE, Saman. Lecture 3: *Intro to Syntax Analysis*, September 1998
URL: http://ceylon.lcs.mit.edu/6035/lecture3/index.htm

[ZAP00]: Cécile Hayez: cecile_hayez@yahoo.fr
Patrick Hendrickx: patje@tartopom.com

7. ANNEXE: SUMMARY OF THE *LAS* CLASSES

7.1.1. Package JavAbInt

Abstract class Val

an instance of Val represents a Java value that can be one of those :

a boolean
an integer
a floating point number *(the basic types defined in the LC99a, p.3)*

an undefined value (type bot) *(as defined in the typed and in the labelized abstract syntax in IPO99, parts 1.3 and 1.4)*

an instance of a class

Fields

these are the unique instances of Val, representing the not initialized values of Java :

public static final Ni Null	*uninitialized value of type bot*
public static final Ni undefBOOL	*uninitialized value of type boolean*
public static final Ni undefINT	*uninitialized value of type int*
public static final Ni undefFLOAT	*uninitialized value of type float*

Constructors

protected Val(Type t)

creates a value of type t

Arguments :

t is the type of the new value

Public methods

Val undef(Type t)

returns the uninitialized value corresponding to the type t

Arguments :

t is the type of the uninitialized value

Type getType()

returns the type of the current value

abstract boolean equals(Val v)

returns the truth value of the statement :
"this value is equal to the value v"

Argument :

v is the value to compare with the current one

Abstract class Base

an instance of Base represents a value of a basic type that can be one of those :
a boolean
an integer
a floating point number
(the basic types defined in the LC99a, p.3)

Constructors

protected Base(Type)
creates a value of the basic type t

Arguments :
t is the type of the new value

Class Bool

an instance of Bool represents a Java value of boolean type

Constructors

Bool (boolean v)
creates a boolean of value v

Arguments :
v is the value of the new Bool

Public methods

boolean getVal()
returns the current boolean value

boolean equals(Val w)
Returns the truth value of the statement :
"this boolean value is equal to the boolean value w"

Arguments :
w is the value to compare with the current boolean value

Class Int

an instance of Int represents a Java value of type int

Constructors

public Int(int v)
creates an integer of value v

Arguments :
v is the value of the new Int

Public methods

int getVal()

returns the current integer value

boolean equals(Val w)

Returns the truth value of the statement :
"this integer value is equal to the integer value w"

Argument :
w is the value to compare with the current integer value

Class Ni

an instance of Ni represents a not initialized value (for a basic type) or null (for a non basic type)

Constructors

protected Ni(Type t)

creates an uninitialized value of type t
pre : t should be bot or a class name

Arguments :
t is the type of the new uninitialized value

Public methods

boolean equals(Val v)

Returns the truth value of the statement :
"this uninitialized value is equal to the uninitialized value v"

Arguments :
v is the uninitialized value to compare with the current value

Class Inst

an instance of Inst represents a Java value of a non basic type i.e. an instance of a class

Constructors

public Inst (Nclasse n, Inst s, Val[] c)

creates a value of a non basic type with the following information :

Arguments :
n is the type of the value (a class name)
s is the reference to the super class
c is the array containing the values of the class fields

Public methods

Nclasse getNclasse()

returns the type of the class

Inst getSuper()

returns the reference to the super class

Val getVal(int i)

returns the value of the field of index i

Arguments :

i is the index of the researched field

void setVal(int i, Val v)

gives the value v to the field of index i

Arguments :

i is the index of the modified field
v is the new value of the field

boolean equals(Val v)

Returns the truth value of the statement :
"this instance of class is the same as the instance v"

Arguments :

v is the instance to compare with the current instance of class

Abstract class Type

an instance of Type represents a Java type.
This can be :

boolean, int, float	*(the basic types defined in the LC99a, p.3)*
void	*(as used in LC99a and defined in IPO99, part 1.2.4)*
bot	*(as defined in the typed and in the labelized abstract syntax in IPO99, parts 1.3 and 1.4)*
a class name	*(as defined in the LC99a, part 2.2)*

Fields

these are the unique instances of Type, representing the basic types of Java :

public static final Type BOOL	*basic type boolean*
public static final Type FLOAT	*basic type float*
public static final Type INT	*basic type int*
public static final Type BOT	*type for uninitialized variables of non basic type*
public static final Type VOID	*type void*

Public methods

boolean equals(Type t)

Returns the truth value of the statement :
"this type is equal to the type t"

Arguments :

t is the type to compare with the current type

boolean lowerThan(Type t)

Implements the strict ordering corresponding to <=_{pi}
(defined in IPO99 : Def 1.4, page 21)
returns the truth value of the statement "this <_{pi} t".

Arguments :
t is the type to compare with the current type

boolean lowerOrEqual(Type t)

Implements the ordering <=_{pi} (defined in IPO99 : Def 1.4, page 21)
returns the truth value of the statement "this <=_{pi} t".

Arguments :
t is the type to compare with the current type

Ni undef()

creates a instance of Ni of the current type
Pre : the current type is a basic type different from BOT and from VOID.

String toString()

creates a String representation of the current type, to allow its display

Class simpleType

an instance of simpleType represents a basic type of Java.
This can be : boolean, int, float, void or bot

The type bot is defined in the typed and in the labelized abstract syntax in IPO99, parts 1.3 and 1.4)

Class Nclasse

an instance of Nclasse represents a Java class type.
It contains all the information available for the class.

Constructors

Nclasse (String c, Nclasse p)

Creates a class type, with the given characteristics.
Pre : a class with the characteristics c and p does not exist yet

Arguments :
c is the name of the class type that must be created
p is the reference to the super class

Public methods

void putDefClass(defClass d)

Associates "this" type with its companion class.
Is not public because it should be used only when the companion class is created.

Arguments :
d is the companion class to associate with.

boolean isArcheType()

Says whether "this" type is an "archetype", i.e., a type without ancestor.

Nclasse getSuperType()

returns the reference to the super class

Ni undef()

creates a uninitialized value of "this" type

String nomComplet()

returns the entire name of the class.

public static boolean existe (String c)

tests if the entire name of class given as argument exists.

Arguments :

c is the name of the researched class

public static Nclasse trouver (String c)

returns the reference to the class with the entire name given as argument.
returns null if ensNclasse does not contain a class with the entire name given as argument.

Arguments :

c is the name of the researched class

defClass getDefClass()

Returns the class associated to this type or null if the link has not been established yet.

boolean lowerOrEqual(Nclasse t)

Implements the ordering <=_{pi} (defined in IPO99 : Def 1.4, page 21)
returns the truth value of the statement "this <=_{pi} t".

Arguments :

t is the instance to compare with "this"

boolean lowerThan(Nclasse t)

Implements the ordering <=_{pi} (defined in IPO99 : Def 1.4, page 21)
returns the truth value of the statement "this <_{pi} t".

Arguments :

T is the instance to compare with "this"

Type epsich (String Nchamp)

Implements the function epsilon_ch defined in IPO99, page 21. Definition 1.6
returns the type of the field given as argument.

Arguments :

Nchamp is the name of the field whose type must be searched

Type epsim(String m, listOfTypes lt)

Implements the function epsilon_m defined in IPO99, page 21. Definition 1.6 returns the return type of the method given as argument. If no compatible method has been declared, it returns null.

Arguments :

m is the name of the method
lt is the list of the arguments types of the method

Type epsic(listOfTypes lt)

Implements the function epsilon_c defined in IPO99, page 21. Definition 1.6 returns the class type if a compatible constructor has been defined. Otherwise, it returns null.

Arguments :

lt is the list of the arguments types of the constructor

Class cellOfListOfTypes

an instance of cellOfListOfTypes is a type of a list of types

Fields

Type v	*The current type*
cellOfListOfTypes next	*The next type of the list*

Constructors

cellOfListOfTypes(Type t, cellOfListOfTypes c)

creates a list of types with the following information :

Arguments :

t is the type of the new cell
c is the following cell of the list

Class listOfTypes

Class description:
-Implements a domain "list of types".
-Implements the ordering induced by the ordering on types on lists of types :

By definition,
(T1, ..., Tm) <= (T'1, ..., T'n)
iff
m = n and
Ti <= T'i (for all i: 1<= i<= m(=n)).

Constructors

public listOfTypes()

creates a new empty list

public listOfTypes(Type t)

creates a new list of types containing the type t.

Arguments :

t is the only type contained in the list

public listOfTypes(Type t1, Type t2)

creates a new list of types with two types t1 and t2

Arguments :

t1 is the first type of the list
t2 is the second type of the list

public listOfTypes(Type t1, Type t2, Type t3)

creates a new list of types with two types t1, t2 and t3

Arguments :

t1 is the first type of the list
t2 is the second type of the list
t3 is the third type of the list

Public methods

void addBefore(Type t)

adds a new type in front of the list

Arguments :

t is the type added in front of the list

void addAfter(Type t)

adds a new type at the end of the list

Arguments :

t is the type added at the end of the list

Type getType (int i)

returns the type of index i.
if i is out of the list bounds, it returns null.
pre : i > 0

Arguments :

i is the index of the needed type

boolean equals(listOfTypes lt)

returns the truth value of the statement
""this" list of types is equal to the list of types lt".

Arguments :

lt is the list of types to compare with the current list of types

boolean lowerThan(listOfTypes lt)

return the truth value of the statement
""this" list of types is strictly lower than the list of types lt".

Arguments :

lt is the list of types to compare with the current list of types

boolean lowerOrEqual(listOfTypes lt)

return the truth value of the statement ""this" list of types is lower or equal to the list of types lt".

Arguments :

lt is the list of types to compare with the current list of types

int arity()

Returns the number of types in the list

String toString()

Returns a String representation of the list of types that can be displayed.

static void main(String[] args)

main function that displays a test program

Arguments :

String [] is the given program

Abstract class declProc

an instance of declProc represents a procedure (i.e. a method or a constructor) declaration

Fields

private graphProc mygraph	*graphProc corresponding to the current procedure*
private listOfTypes myListOfTypes	*list of the types of the arguments of the procedure*
private TypePourEnv myTypePourEnv	*the local environment of the procedure*
private Instr firstInstr	*the first statement of the procedure*

Constructors

protected declProc (listOfTypes, graphProc)

creates an instance of declProc, corresponding to the concrete method whose properties are given as arguments

Arguments :

listOfTypes is the list of the argument types of the method
graphProc is the graph associated to the method

protected declProc (listOfTypes, graphProc, TypePourEnv, Instr)

creates an instance of declProc, corresponding to the concrete method whose properties are given as arguments

Arguments :

listOfTypes is the list of the argument types of the method
graphProc is the graph associated to the method
TypePourEnv is the local environment of the method
Instr is the reference to the list of statements of the method

Public methods

void putGraph(graphProc)

gives a value to the graphProc mygraph

Arguments :

graphProc is the new value of mygraph

void putTypePourEnv(TypePourEnv)

gives a value to the local environment

Arguments :

TypePourEnv is the new value to give to the local environment

void putFirstInstr(Instr)

gives a value to the list of statements

Arguments :

Instr is the reference to the list of statements of "this" procedure

listOfTypes listOfTypes()

returns the list of the types of the arguments

TypePourEnv getTypePourEnv()

returns the local environment

Instr getFirstInstr()

returns the list of statements

Class declConstr

an instance of declConstr represents a constructor declaration.

Fields

final public static String PREM	*first constructor*
final public static String SUPER	*constructor based on a constructor of the super class*
final public static String THIS	*constructor based on a constructor of this class*
private String mysort	*sort of the construct: one of the above mentioned*
private listOfExpr myListOfExpr	*list of the expressions used for the call to another constructor (of the super class or of the current class)*
declConstr myTwinOrFather	*the constructor to execute first.*

Constructors

public declConstr(String, listOfTypes, graphProc)

Constructs an instance of the class, of the sort given as String argument

Arguments :

String is the sort of the constructor ("prem", "this" or "super")
listOfTypes is list of the arguments types of the constructor
graphProc is the graph associated to the constructor

public declConstr(listOfTypes, graphProc)

Creates an instance of the class, for an obsolete constructor

Arguments :

listOfTypes is list of the arguments types of the constructor
graphProc is the graph associated to the constructor

public declConstr(String, listOfTypes, graphProc, declConstr, listOfExpr, TypePourEnv, Instr)

Creates an instance of the class, of the sort given as String argument, about which we have all the information

Arguments :

String is the sort of the constructor ("prem", "this" or "super")
listOfTypes is list of the arguments types of the constructor
graphProc is the graph associated to the constructor
declConstr is the super or the twin constructor
listOfExpr is the list of effective parameters used to call the super or the twin constructor
TypePourEnv is the local environment of the constructor
Instr is the list of statements of the constructor

Public methods

void putListOfExpr (listOfExpr)

instanciates the listOfExpr myListOfExpr

Arguments :

listOfExpr is the new value of myListOfExpr

void putTwinOrFather(declConstr)

instanciates the field myTwinOrFather

Arguments :

declConstr is the new value of myTwinOrFather

String getSortOfConstr()

Returns the sort of the construct

listOfExpr getListOfExpr()

Returns the listOfExpr myListOfExpr

declConstr getTwinOrFather()

Returns the constructor to execute first.

static declConstr newPremConstr(listOfTypes, graphProc)

Returns a new constructor of type "prem"

Arguments :

listOfTypes is the list of the arguments types of the constructor
graphProc is the graph associated to the constructor

static declConstr newPremConstr(listOfTypes, graphProc, TypePourEnv, Instr)

Returns a new constructor of type "prem" for which we give all the information

Arguments :

listOfTypes is the list of the arguments types of the constructor
graphProc is the graph associated to the constructor
TypePourEnv is the local environment of the constructor
Instr is the list of statements of the constructor

static declConstr newSuperConstr(listOfTypes, graphProc)
Returns a new constructor of type "super"

Arguments :
listOfTypes is the list of the arguments types of the constructor
graphProc is the graph associated to the constructor

static declConstr newSuperConstr(listOfTypes, graphProc, declConstr, listOfExpr, TypePourEnv, Instr)
Returns a new constructor of type "super" for which we give all the information

Arguments :
listOfTypes is the list of the arguments types of the constructor
graphProc is the graph associated to the constructor
TypePourEnv is the local environment of the constructor
Instr is the list of statements of the constructor

static declConstr newThisConstr(listOfTypes, graphProc)
Returns a new constructor of type "this"

Arguments :
listOfTypes is the list of the arguments types of the constructor
graphProc is the graph associated to the constructor

static declConstr newThisConstr(listOfTypes, graphProc, declConstr, listOfExpr, TypePourEnv, Instr)
Returns a new constructor of type "this" for which we give all the information

Arguments :
listOfTypes is the list of the arguments types of the constructor
graphProc is the graph associated to the constructor
TypePourEnv is the local environment of the constructor
Instr is the list of statements of the constructor

String toString()
Returns a string representation of a declConstr to be displayed

Class declMethode

an instance of declMethode represents a method declaration.

final public static String ABSTRACT	*abstract method*
final public static String CONCRETE	*concrete method*
private String mysort	*the sort of the method : one of the above mentioned*
private String myname	*name of the method*
private Type myResultType	*the result type of the method*

Constructors

public declMethode (String, Type, String, listOfTypes, graphProc)

creates an instance of declMethod, corresponding to the method whose properties are given as arguments

Arguments :

String is the sort of the method
Type is the return type of the method
String is the name of the method
listOfTypes is list of the arguments types of the method
graphProc is the graph associated to the method

public declMethode (String, Type, String, listOfTypes, graphProc, TypePourEnv, Instr)

creates an instance of declMethod, corresponding to the method whose properties are given as arguments

Arguments :

String is the sort of the method
Type is the return type of the method
String is the name of the method
listOfTypes is list of the arguments types of the method
graphProc is the graph associated to the method
TypePourEnv is the local environment of the method
Instr is the list of statements of the method

public declMethode (Type, String, listOfTypes, graphProc)

creates an instance of declMethod, corresponding to the concrete method whose properties are given as arguments

Arguments :

Type is the return type of the method
String is the name of the method
listOfTypes is list of the arguments types of the method
graphProc is the graph associated to the method

public declMethode (Type, String, listOfTypes, graphProc, TypePourEnv, Instr)

creates an instance of declMethod, corresponding to the concrete method whose properties are given as arguments

Arguments :

Type is the return type of the method
String is the name of the method
listOfTypes is list of the arguments types of the method
graphProc is the graph associated to the method
TypePourEnv is the local environment of the method
Instr is the list of statements of the method

Public methods

String getSortOfMethode()

returns the sort of the method

String getName()

returns the name of the method

Type getType()

returns the result type of the method

static declMethode newAbstractMethode(Type, String, listOfTypes, graphProc)

creates an instance of declMethod, corresponding to the abstract method whose properties are given as arguments

Arguments :

Type is the return type of the method
String is the name of the method
listOfTypes is list of the arguments types of the method
graphProc is the graph associated to the method

static declMethode newConcreteMethode(Type, String, listOfTypes, graphProc)

creates an instance of declMethod, corresponding to the concrete method whose properties are given as arguments

Arguments :

Type is the return type of the method
String is the name of the method
listOfTypes is list of the arguments types of the method
graphProc is the graph associated to the method

static declMethode newConcreteMethode(Type, String, listOfTypes, graphProc, TypePourEnv, Instr)

creates an instance of declMethod, corresponding to the concrete method whose properties are given as arguments

Arguments :

Type is the return type of the method
String is the name of the method
listOfTypes is list of the arguments types of the method
graphProc is the graph associated to the method
TypePourEnv is the local environment of the method
Instr is the list of statements of the method

String toString()

creates a string representation of the declMethod, to allow its display

Class defClass

an instance of defClass represents a class, with all its proprieties.

Fields

private Hashtable ensOfNames

set of simple names of all fields and methods for this class. A method name is completed with "()" to make it different from the corresponding field name

private Nclasse myType	*The type associated to this class*
private int nbrOfChamps	*number of fields*
private int nextIchamp	*counter of fields*
private String [] Nchamp	*array with the names of the fields*
private Type [] Tchamp	*array with the types of the fields*

private int nbrOfMethodeNames — *number of methods*
private int nextImethodeName — *counter of methods*
private String [] MethodeName — *array with the different method names*
private Type[] MethodeType — *array with the result types of the methods*
private graphProc [] graphOfMethodes — *array with the graphProc of the methods*
private int nbrOfConstr — *number of constructors*
private graphProc graphOfConstr — *graphProc associated with the constructors*

Constructors

public defClass(Nclasse, int, int)

Creates a new defClass and links it to the type t. No field, method, or constructor is created.

Argument :

Nclasse is the Type associated to the class
int is the number of fields declared in this class
int is the number of different method names

Public methods

int addChamp(String, Type)

If the class does not contain a field named by the String in argument, a new field is added to the class. Moreover, the index of this field is returned. If such a field already exists, the value -1 is returned.

Argument :

String is the name of the new field
Type is the type of the new field

int getIchamp(String)

Check whether a field already exists. Return the index of the field s or -1 if it doesn't exist (yet). Can be used to "convert" the name of a field to its corresponding index.

Argument :

String is the researched field

int addMethodeName(String, Type)

If the class does not contain a "MethodeName" as the given String, a new MethodeName is added to the class. Moreover, the index of this MethodeName is returned. If such a "MethodeName" already exists, the value -1 is returned.

Argument :

String is the name of the new method
Type is the return type of the new method

int getMethodeName(String)

Checks whether a "MethodeName" already exists. Returns the index of the "MethodeName" or -1 if it doesn't exist (yet). Can be used to "convert" the MethodeName to its corresponding index.

Argument :

String is the name of the to get method

declMethode addMethode(String, Type, listOfTypes)

Roughly speaking, this method adds a new method to "this" class.
The following precondition are checked:
1) another method with the same signature doesn't exist.
2) other existing methods with the same name have the same type t.
If any precondition is violated the null value is returned.
Otherwise the reference to the declaration of the new method is returned

Argument :

String is the name of the new method
Type is the return type of the new method
listOfTypes is the list of arguments types of the method

declMethode addMethode(declMethode)

Roughly speaking, this method adds a new method to "this" class. It is assumed that the method is "sufficiently" initialized.
The following precondition are checked:
1) another method with the same signature doesn't exist.
2) other existing methods with the same name have the same type.
If any precondition is violated the null value is returned.
Otherwise the reference to the declaration of the new method is returned.

Argument :

declMethode is the object corresponding to the to add method

declConstr addConstr(listOfTypes)

Roughly speaking, this method adds a new constructor to "this" class.
This precondition is checked: another constructor with the same signature doesn't exist. If this precondition is violated the null value is returned.
Otherwise the reference to the declaration of the new constructor is returned.

Argument :

listOfTypes is the list of arguments types of the constructor

declConstr addConstr(declConstr)

Roughly speaking, this method adds the constructor c to "this" class.
This precondition is checked: another constructor with the same signature doesn't exist.
If this precondition is violated the null value is returned.
Otherwise the reference to the declaration of the new constructor is returned.

Argument :

declConstr is the object corresponding to the to add constructor

Type epsi0ch (String)

returns the type of the field given as argument, this field must have been declared in the class itself.

Argument :

String is the name of the to get field

Type epsich (String)

returns the type of the field given as argument, this field can have been declared in the class itself or in a super class.

Argument :

String is the name of the to get field

Type epsim(String, listOfTypes)

returns the result type of the method given as argument, if no compatible method has been declared, it returns null.

Argument :

String is the name of the to get method
listOfTypes is the list of the arguments types of the to get method

Type epsic(listOfTypes)

returns the class type if a compatible constructor has been defined. Otherwise, it returns null.

Argument :

listOfTypes is the list of the arguments types of the to get constructor

declMethode mostSpecificMethode(String, listOfTypes)

This is a "more informative" version of epsim. If there is a (unique of course!) most specific method as given whose signature is more general or equal to the given one, the reference to this method is returned. The null reference is returned, otherwise.

Argument :

String is the name of the to get method
listOfTypes is the list of the arguments types of the to get method

declConstr mostSpecificConstr(listOfTypes)

This is a "more informative" version of epsic. If there is a (unique of course!) most specific constructor Nameofthisclass(listOfTypes) whose signature is more general or equal to the given one, the reference to this constructor is returned. The null reference is returned, otherwise.

Argument :

listOfTypes is the list of the arguments types of the to get constructor

Nclasse getType()

Returns the type associated to this class.

int getNumberOfFields()

Returns the number of fields

String getNameOfField(int)

Returns the name of the field of the given index

Argument :

int is the index of the to get field

Type getTypeOfField(int)
returns the type of the field of the given index

Argument :
int is the index of the to get field

String toString()
Returns an external (printable) representation of this defClass

Class Env

An instance of Env represents a local semantic environment.

Fields

private TypePourEnv infostat *contains the "static" information of the environment*
private Val[] v *variables values*
private Inst thisRef *the current instance (alias this).*

Constructors

public Env(TypePourEnv, Inst)
Creates a new instance of Env, with the given information
All the variables are set to "undef".

Argument :
TypePourEnv is the information about the "static" environment
Inst is the reference to the instance of the current class

Public methods

Val getVal (int)
returns the value of the variable of the given index

Argument :
int is the index of the to get variable

void setVal(int, Val)
gives a new value to the variable of the given index

Argument :
int is the index of the to modify variable
Val is the new value of the variable

Type getType(int)
returns the type of the variable of the given index

Argument :
int is the index of the to get variable

String getNom(int)
returns the name of the variable of the given index

Argument :
int is the index of the to get variable

int getNbrVar()
returns the number of variables of the environment

Type getTypeOfThis()
returns the type of the current class

Class TypePourEnv

Contains the static information relative to an environment

Fields

private int nombreDeVariables *number of the variables*
private Type[] type *type of the variables*
private String[] nom *name of the variables*

private Nclasse typeOfThis *type of the current instance (alias this).*

Constructors

public TypePourEnv(Nclasse, int, Type[], String[])
Creates a new instance of TypePourEnv with the given information

Argument :
Nclasse is the Type associated to the current class
int is the number of variables
Type[] is the array with the types of the class variables
String[] is the array with the names of the class variables

Public methods

Type getType(int)
Returns the type of the variable of index i

Argument :
int is the index of the to get variable

String getNom(int)
Returns the name of the variable of index i

Argument :
int is the index of the to get variable

int getNbrVar()
Returns the number of variables

Type getTypeOfThis()
Returns the type of the current instance

public String toString()
Returns an external (printable) representation of this TypePourEnv.

Class cellOfGraphProc

class that represents a cell in the list graphProc, corresponding to one procedure

Fields

declProc v	*information on the procedure*
cellOfGraphProc next	*next cell on the list*

Constructors

cellOfGraphProc(declProc, cellOfGraphProc)

creates an instance of cellOfGraphProc

Arguments :

declProc is the value of the cell (the information on a procedure)
cellOfGraphProc is the next cell in the list

Class graphProc

Class description:

-implements a graph of procedure of "same kind" (constructors or same name and type).
-allows one to find a procedure with a given signature.
-allows one to find the list of procedures whose list of types is minimaly greater that a given list of types.

Fields

static String METHODE	*procedure of type method*
static String CONSTR	*procedure of type constructor*
private final String sortOfGraph	*One of the above's.*
private final Nclasse classOfGraph	*class associated with the graph*
private String methodName	*common name of the methods of the graph*
private Type methodType	*common result type of the methods of the graph*
private cellOfGraphProc first	*first cell of the list of cellOfGraphProc*

Constructors

public graphProc(Nclasse)

creates an instance of a graphProc of type constructor, containing the information given as argument

Arguments :

Nclasse is the name of the class

public graphProc(Nclasse, String, Type)

creates an instance of a graphProc of type method, containing the information given as argument

Arguments :

Nclasse is the name of the class
String is the name of the method
Type is the return type of the method

Public methods

cellOfGraphProc listOfMostSpecificProcs (listOfTypes, cellOfGraphProc)

Pre: oldlist is a list of Procs with the same arity as the arguments listOfTypes all those methods have a list of types greater or equal to the arguments listOfTypes the list of types are not comparable two by two.

This method merges the list of Procs of this graph, whose list of arguments types is greater or equal to the arguments listOfTypes, to the arguments cellOfGraphProc Only most specific Procs are returned.
Thus if a Proc is less specific than another, only the latter is kept in the list.
If a Proc in oldlist has the same list of types as a Proc in the graph, the latter is not added to the returned list.

Arguments :

listOfTypes is the list of arguments types
cellOfGraphProc is the old list of types that must be merged with the new one.

declMethode addMethode(listOfTypes)

if a method, whose list of types corresponds to the arguments listOfTypes, already exists in the graph, null is returned.
*Otherwise, such a method is *created* and then added to the graph.*
the reference to the method declaration is returned.

Arguments :

listOfTypes is the list of arguments types of the to add method.

declMethode addMethode(declMethode)

if a method, whose list of types is the same as the to add method, already exists in the graph, null is returned. Otherwise, this method is added to the graph. The reference to the added method is returned.

Arguments :

declMethode is the to add method.

declConstr addConstr(listOfTypes)

if a constructor, whose list of types corresponds to the arguments listOfTypes, already exists in the graph, null is returned.
*Otherwise, such a constructor is *created* and then added to the graph.*
the reference to the constructor declaration is returned.

Arguments :

listOfTypes is the list of arguments types of the to add constructor.

declConstr addConstr(declConstr)

if a constructor, whose list of types is the same as the to add constructor, already exists in the graph, null is returned. Otherwise, this constructor is added to the graph.
The reference to the added constructor is returned.

Arguments :

declConstr is the to add constructor.

boolean existProc(listOfTypes)

Checks whether a Proc declaration with specific list of arguments types already exists in this graph.

Arguments :

listOfTypes is the to check list of arguments types

String toString()

Returns an external (printable) representation of this graphProc.

7.1.2. Package JavAbInt.concreteSyntax

Class lexeme

This class implements the set of lexical items that are relevant for the intermediate internal representation of VTF programs. These are

- *Identifiers (3.8)*
- *Keywords (3.9)*
- *Literals (3.10)*
- *Operators (3.12)*

Here we use the classification of Chapter 3 of JLSpec, from which we eliminate irrelevant symbols. Moreover, only the symbols defined in VTF are recognized. Every lexical item is uniquely represented.

Fields

static Hashtable ensLexeme	*set of all the lexemes*
The sorts of lexeme :	
public final static String IDENTIFIER	*an identifier*
public final static String LITERAL	*a literal*
public final static String OPERATOR	*an operator*
public final static String KEY_TYPE	*a type name*
public final static String KEY_ACCES	*an access modifier*
public final static String KEY_COMMANDE	*a statement keyword*
public final static String KEYWORD	*a general keyword*
specific lexemes :	
public final static lexeme ALL_CLASS	*the symbol "*" in a package declaration*
public final static lexeme CLASS	*the keyword "class"*
public final static lexeme EXTENDS	*the keyword "extends"*
public final static lexeme IMPORT	*the keyword "import"*
public final static lexeme PACKAGE	*the keyword "package"*
public final static lexeme SUPER	*the keyword "super"*
public final static lexeme THIS	*the keyword "this"*
public final static lexeme BOOLEAN	*the keyword "boolean"*
public final static lexeme FLOAT	*the keyword "float"*
public final static lexeme INT	*the keyword "int"*
public final static lexeme VOID	*the keyword "void"*
public final static lexeme STATIC	*the keyword "static"*

public final static lexeme ABSTRACT — *the keyword "abstract"*
public final static lexeme FINAL — *the keyword "final"*
public final static lexeme PRIVATE — *the keyword "private"*
public final static lexeme PACKAGE_MOD — *when there is no specific access modifier*

public final static lexeme PROTECTED — *the keyword "protected"*
public final static lexeme PUBLIC — *the keyword "public"*

public final static lexeme AFFECT — *the assignment*
public final static lexeme WHILE — *the statement "while"*
public final static lexeme RETURN — *the statement "return"*
public final static lexeme NEW — *the keyword "new"*
public final static lexeme IF — *the keyword "if"*
public final static lexeme ELSE — *the keyword "else"*

public final static lexeme LTH — *operator("<")*
public final static lexeme LEQ — *operator("<=")*
public final static lexeme GTH — *operator(">")*
public final static lexeme GEQ — *operator(">=")*
public final static lexeme EQ — *operator("==")*
public final static lexeme NEQ — *operator("!=")*
public final static lexeme MOD — *operator("%")*
public final static lexeme PLUS — *operator("+")*
public final static lexeme MINUS — *operator("-")*
public final static lexeme MULT — *operator("*")*
public final static lexeme DIV — *operator("/")*
public final static lexeme AND — *operator("&")*
public final static lexeme OR — *operator("|")*

public final static lexeme FALSE — *the keyword "false"*
public final static lexeme NULL — *the keyword "null"*
public final static lexeme TRUE — *the keyword "true"*

private String sortOfLexeme — *the sort of the lexeme*
private String valueOfLexeme — *the value of the lexeme*

Constructors

private lexeme(String, String)

Creates a new lexeme with the given information

Arguments :

String is the sort of the lexeme
String is the value of the lexeme

Public methods

String getSort()

returns the sort of the lexeme

String getValue()

returns the value of the lexeme

static lexeme lexemeOf(String, String)

Converts a string v of sort s to a lexeme.
Precondition: v actually is of sort s. (No checking!)

Arguments :
String is the sort of the lexeme
String is the value of the lexeme

static lexeme identifierOf(String)

Converts an identifier v to a lexeme.
Precondition: v actually is an identifier. (No checking!)

Arguments :
String is the value of the lexeme

static lexeme literalOf(String)

Converts a literal v to a lexeme.
Precondition: v actually is a literal. (No checking!)

Arguments :
String is the value of the lexeme

static lexeme keywordOf(String)

Converts a keyword v to a lexeme.
Precondition: v actually is a keyword. (No checking!)

Arguments :
String is the value of the lexeme

static lexeme operatorOf(String)

Converts an operator v to a lexeme.
Precondition: v actually is an operator. (No checking!)

Arguments :
String is the value of the lexeme

Class construct

A construct either is a terminal (lexeme) or a sentence. In the latter case, it is in fact an instance of a non terminal, i.e., a data structure exhibiting the value and structure of this non terminal instance.

Fields

public static String TERMINAL	*a construct of type "terminal"*
public static String SENTENCE	*a construct of type "sentence"*
private construct next	*The next construct in a sentence*
private String sortOfConstruct	*either TERMINAL or SENTENCE*

Constructors

protected construct(String, construct)

Creates an instance of construct with the given information

Arguments :
String is the sort of the construct

construct is the next construct in the sentence

protected construct (String)

Creates an instance of construct with no following construct

Arguments :

String is the sort of the construct

Public methods

construct getNext()

returns the next construct in a sentence

void setNext(construct)

set a construct as the next of the current one

Arguments :

construct is the new next construct

String getSort()

returns the type of the construct

Class sentence

A sentence consists of
- a "main cell" containing
+ the sort of the sentence (statement, expression, etc.)
+ the reference to the first construct of the sentence
+ the reference to the last construct of the sentence
both pointer are null if the sentence is empty;
- a sequence of constructs representing in a structured way the sentence.

Fields

final public static String ABSTR_METHOD_DECL	*an abstract method declaration*
final public static String CLASS_DECL	*a class declaration*
final public static String CONCR_METHOD_DECL	*a concrete method declaration*
final public static String CONTRUC_DECL	*a constructor declaration*
final public static String FIELD_DECL	*a field declaration*
final public static String IMPORT_DECL	*an import declaration*
final public static String PACKAGE_DECL	*a package declaration*
final public static String VAR_DESIGN	*a variable name*
final public static String SUPER_DESIGN	*the "super" designator*
final public static String THIS_DESIGN	*the "this" designator*
final public static String PROGRAM	*a program*
final public static String NOM	*a name*
final public static String LIST_PARAM_EFF	*a list of effective parameters*
final public static String LIST_PARAM_FORM	*a list of formal parameters*
final public static String PARAM_FORM	*a formal parameter*
final public static String STATEMENT	*a statement*
final public static String LIST_STATEMENT	*a statement list*
final public static String EXPRESSION	*an expression*
final public static String APPEL_PROC	*a method call*
final public static String APPEL_NEW	*a "new" call*
final public static String APPEL_CONSTR	*a constructor call*

final public static String CAST — *a cast*

private construct first — *The first construct in a sentence.*
private construct last — *The last construct in a sentence.*
private String sortOfSentence — *either STATEMENT or EXPRESSION or ...*

Constructors

public sentence(String, construct)

Creates an instance of sentence with the given information

Arguments :

String is the sort of the sentence
construct is the next construct in a sentence

public sentence(String)

Creates an instance of sentence with the given information

Arguments :

String is the sort of the sentence

Public methods

construct getFirstConstruct()

returns the first construct of the sentence

construct getLastConstruct()

returns the last construct of the sentence

void addFirstConstruct(construct)

adds a new construct at the beginning of the sentence

Arguments :

construct is the to add construct

void addLastConstruct(construct)

adds a new construct at the end of the sentence

Arguments :

construct is the to add construct

String getSort()

returns the sort of the sentence

Class terminal

A terminal (lexeme) is an occurrence of a lexeme in a sentence.

Constructors

public terminal(lexeme l, construct c)

Creates an instance of terminal with the given information

Arguments :

lexeme is the lexeme of the new terminal
construct is the next construct in a sentence

public terminal (lexeme l)

Creates an instance of terminal with the given information

Arguments :

lexeme is the lexeme of the new terminal
construct is the next construct in a sentence

Public methods

lexeme getLexeme()

returns the lexeme of "this" terminal

7.1.3. Package JavAbInt.concreteSyntax.Parser

class Nlook1

An instance of the class Nlook1 is a special object, containing : the stream of characters with the code of the to parse program given as input, and different information on the tokens and on the options of the parser.

Public methods

sentence IIRParser (InputStream)

this method returns the syntactic tree of the program given as argument, corresponding to the IIR definition. The return value is a sentence.

Arguments :

InputStream is the text file of a Java program, that is supposed to be syntactically correct, following the definition of "syntactical correctness".

In the IIRParser precondition, we use the notion of "syntactical correctness". We will define here what this notion exactly means in this method. The parser methods check that the construction given in entrance follow the rules of the VTF syntax. But they don't check the following rules :

- *it doesn't check that the declarations (after the "package" and the "import" declarations) are class declarations.*
- *it doesn't check that the declaration in entrance has got allowed access modifiers. ex : the method would accept a class defined as "protected" and "static"*
- *it doesn't check that the declarations in the body of a class are allowed declarations (not a class declaration)*
- *it doesn't check that a field is not declared with the type "void"*
- *it doesn't check that a field or a method is declared with an identifier. ex : you can declare : int = 1;*
- *it doesn't check that the statements defined in a simple method (not a constructor) are different of a constructor call.*

- *it accepts that a designator is only followed by a semicolon but it shouldn't be accepted.*

- *it doesn't check that a designator has got two consecutive lists of parameters.*
  ex : you can have : toto.add(x,y)(z)

A program in entrance of this method is "syntactically correct" in our terms if it respects the rules that are not checked by the parser.
That parser could be improved by adding code that would test all these constraints.At the moment, the main method of the class CheckType checks all these unchecked rules.

7.1.4. Package JavAbInt.concreteSyntax.Display

Class IIRDisplay

Public methods

static void ConstructDisplay (construct)
this method displays a object of type construct (sentence or lexeme) by calling the corresponding method and initializing the level

Arguments :
construct is the to display construct

7.1.5. Package JavAbInt.concreteSyntax.Tools

public class CheckType

An instance of CheckType represents a special object, containing a main method which cerates objects corresponding to the SAP grammar and checks the types of a program given in entrance.

Fields

private static String CurrentClass *the class which we deal with*

Public methods

static void main(String args[]) throws Exception
this method calls the paser Nlook1, which creates an IIR tree.
Creating the SAP objects, it checks the syntax rules (not checked by the parser) and all the type checking rules defined in the thesis "semantiques operationnelles et domaines abstraits pour l'analyse statique de java" from Isabelle Pollet (from p.25)

Arguments :
String is the file with the entrance program

class CheckTypeException

an instance of CheckTypeException represents a particular exception thrown by the CheckType program.

Constructors

CheckTypeException (String)

This method creates an instance of CheckTypeException, reporting to the given error message

Arguments :

String is the Exception message to throw

Public methods

static void ThrowException (int, String, String) throws CheckTypeException

The methods creates a CheckTypeException, with the exception messages reporting to the different errors :

Arguments :

int is the number corresponding to the error message
String is the information needed to create an explicite message (method name, invalid access modifier...)
String is the name of the class where the error occured

7.1.6. Package JavabInt.SAP

class cellOfListOfExpr

an instance of cellOfListOfExpr represents a cell of a list of expressions used in a method or a constructor call

Fields

Expr v	*the expression of the current cell*
cellOfListOfExpr next	*the following cell*

Constructors

cellOfListOfExpr(Expr, cellOfListOfExpr)

creates an instance of cellOfListOfExpr with the given values

Arguments :

Expr is the expression of the to create cell
cellOfListOfExpr refers to the next cell of the list

Public methods

Type getType()

returns the type of the expression of "this" cell or null if the expression has not been initialized

Abstract class Expr

This class implements the expressions of the VTF grammar. It's useful to have a type that gather all the expression types...

Fields

final public static String NULL	*the "null" value*
final public static String LITT	*the expression is a litteral (int, float or boolean)*
final public static String TERME	*an predefined operator applied on expressions*
final public static String EXPRDES	*a designator*
private String sortOfExpr	*the sort of the expression (one of the above-mentioned)*

Constructors

Expr(String)

creates an instance of Expr, of the given sort

Arguments :

String is the sort of the to create Expr

Public methods

String sortOfExpr()

returns the sort of the expression

abstract void putType(Type)

gives a value to the type of the expression

Arguments :

Type is the type of the expression

abstract Type getType()

returns the type of the expression

String prefToString()

allows the display of "this"

String suffToString()

allows the display of "this"

public class Null extends Expr

an instance of Null represents the "null" expression in Java

Fields

private Type type *the type of the expression*

Constructors

public Null()

creates a new instance of Null

public Null(Type t) { super(NULL); putType(t);}

creates an instance of Null, whose type is known

Arguments :

Type is the type of the to create "null" expression

Public methods

void putType(Type)

gives a value to the current null expression

Arguments :

Type is the type of "this" expression

Type getType()

returns the type of "this" expression

void fulls()

gives to "this" expression the type of the "null" value

String toString()

allows the display of the expression

Abstract class Instr

This class implements the set of statements accepted by the VTF grammar. Every statement is uniquely represented.

Fields

	the sorts of statements
final public static String AFFECT	*an assignment*
final public static String IF	*the statement "if"*
final public static String WHILE	*the statement "while"*
final public static String SKIP	*the statement "skip"*
final public static String PROC	*a procedure call*
final public static String RETURN	*the statement "return"*
final public static String FONC	*a method call*
final public static String CONSTR	*a constructor call*
private String sortOfInstr	*the sort of the statement*
private Instr lab	*The next Instr*

Constructors

Instr(String s)

Instr(String s, Instr l)

Public methods

String sortOfInstr()

Instr getNext()

void putNext(Instr l)

Class Affect

This class implements the statement "assignment"

Fields

private Des des	*the designator of the current assignment*
private Expr expr	*the expression of the current assignment*

Constructors

Affect()	*creates an Instr of type AFFECT*
public Affect(Instr l)	*creates an Instr of type AFFECT with "l" as following statement*
public Affect(Des d, Expr e, Instr l)	*creates an Instr of type AFFECT with "l" as following statement, "e" as expression and "d" as designator*

Public methods

Des getDes()	*returns the designator of the current assignment*
void putDes(Des d)	*puts a designator into the current assignment*
Expr getExpr()	*returns the expression of the current assignment*
void putExpr(Expr e)	*puts an expression into the current assignment*
String toString()	*creates a string to allow the display of the current assignment*

Class tableOfNvars

This class is used when translating a declaration of procedure from IRR to SAP.
- All parameters must be added before the first local variable is (added).

Fields

private int nbrOfParams = 0	*number of parameters*
private int nbrOfVars = 0	*total number of parameters and local variables*
private int nbrOfPVIs = 0	*total number including "internals"*
private String [] arrayOfNames	*array with the name of the variables*
private Type [] arrayOfTypes	*array with the type of the variables*
private listOfTypes l	*"listOfTypes" corresponding to the parameters*
private Hashtable ensOfNvars	*set of variables*

Public methods

void makeDataStructures()	*This method creates the arrays with the types and the names of the variables, using the numbers of variables and the information in the Hashtable.*
int addParam (String p, Type tp)	*This method adds a parameter in the table. It also modifies the counters of variables and returns the index of the variable in. It returns –1 if the variable has already been defined.*
int addLocalVar(String v, Type tv)	*This method adds a local variable in the table. It also modifies the counters of variables and returns the index of the variable in. It returns –1 if the variable has already been defined.*
int addInternalVar(Type ti)	*This method adds an internal variable in the table. It also modifies the counters of variables and returns the index of the added variable It returns –1 if the variable has already been defined.*
int getVarIndex(String s)	*This method returns the index of a variable in the table. It returns –1 if the variable has not been defined yet.*
Type getVarType(String s)	*This method returns the type of a variable. If the variable has not been defined yet, it returns "null"*

static String internalName(int i)

This method transforms an integer into an internal variable name.

String [] getArrayOfNames()

This method returns the array containing the names of the variables of the current tableOfNvars.

Type [] getArrayOfTypes()

This method returns the array containing the types of the variables of the current tableOfNvars.

listOfTypes getListOfTypes()

This method returns a "listOfTypes" containing the types of the variables given as parameters.

int getNbrOfParams()

This method returns the number of parameters

int getNbrOfLocalVars()

This method returns the number of local variables

int getNbrOfInternalVars()

This method returns the number of internal variables

int getNbrOfNvars()

This method returns the total number of variables

Class elementOfEnsOfNvars

this class implements a element of the set "ensOfNvar"

Fields

String n	*name of the variable*
Type t	*type of the variable*
int i	*index of the variable*

Constructors

elementOfEnsOfNvars(String n1, Type t1, int i1)