Vrije Universiteit Brussel - Belgium

Faculty of Sciences

In Collaboration with Ecole des Mines de Nantes - France

2002

A Binding-Time Analysis for petitCafé

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

By: María Laura Ponisio

Promotor: Prof. Theo D'Hondt (Vrije Universiteit Brussel)
Co-Promotor: Jacques Noyé (Ecole des Mines de Nantes)

To my parents, Cristina y Jorge.

To my grandparents, Emilio, Cristina, Angélica y Héctor.

To Patricio.

Abstract

One of the major advantages of object-oriented languages is that they allow the easy construction of general software components, but this is achieved at high cost in terms of inefficiency.

We can attenuate this inefficiency by adapting programs automatically to the environment where they are inserted. Partial evaluation is a program specialization technique to achieve this task, transforming a program into a more specialized version of it. Off-line partial evaluators perform as a first step a binding-time analysis, BTA, that determines which parts of the program can be executed in advance and annotates them accordingly. The BTA only works with the knowledge that some input values are known and some are not. Then, the analysis propagates this knowledge through the program and annotates each program construct, indicating whether the knowledge of the input value will make it possible to compute this program construct or not. Consequently, the design of a correct and precise BTA is of primary importance to perform efficient off-line partial evaluation.

The mission of this work was to lay the foundations for that design. Thus, we have modeled a simple BTA for a subset of Java that relies on constraint-based program analysis.

We have achieved the mission by designing petitCafé, a subset of Java with specific object-oriented features, modeling a simple BTA for petitCafé using constraints and implementing a prototype of the analysis. Our model adds precision to the analysis by considering objects as complex structures and it keeps the analysis simple by classifying them into equivalence classes. It is simple and flexible. It provides the basis for implementation, facilitates the reasoning that would be no longer possible using a too complicated model and lays the foundations for the design of correct and precise BTAs for Java.

Acknowledgements

Thanks to my parents Cristina Noval and Jorge Ponisio.

Thanks to Patricio Armesto for his support.

Thanks to my grandfather, Emilio H. Noval.

I would like to thank Jacques Noyé, my supervisor, for his interest in my work, guidance, understanding, patience and constant support during this research. He was always there with a smile.

Thanks to Prof. Theo D'Hondt for his teaching and for being the alma mater of the EMOOSE.

I would like to thank Ulrik Pagh Schultz for reviewing this work, for his advice and time.

Thanks to Gustavo Rossi for his advice, support, guidance and trust. Thanks to the LIFIA. Thanks to Gabi Arevalo.

Special thanks to Luis Ernesto Cleve, Paul Asman, José Luis Villa, and Martín Mantovanni.

I am very fortunate that our paths have met.

Nantes, France — María Laura Ponisio

August 21, 2001

Table of Contents

Chapter 1

Introduction

One of the major advantages of object-oriented languages is that they allow the easy construction of general software components. But this generality usually implies inefficiency. This can be solved in part using partial evaluation[14].

Partial evaluation is a technique that consists in transforming a program into a more specialized version of it. This new version is more efficient in terms of memory and performance and the transformation is done using part of the program's input.

One of the strategies to perform partial evaluation is the off-line strategy that consists of splitting the partial evaluation process into two phases: *binding-time analysis* that determines which parts of the program can be executed in advance and to annotate them accordingly; and *specialization* which processes a program driven by the binding time information and the concrete values.

Binding-time analysis determines at what time (basically specialization time or execution time) the value of a variable can be computed, that is, the time when the value can be bound to the variable[14][13]. Thus, the binding time information of a program can be used for specialization as long as the input values match the known/unknown pattern given for binding-time analysis[7].

Since binding-time analysis determines how an off-line partial evaluator will specialize a program, the accuracy of the binding-time information directly determines the degree of specialization[11]. It becomes important, then, to have a model that facilitates the reasoning to well understand and develop ways to build correct and efficient binding-time analyses.

1.1 Thesis

This dissertation deals with binding-time analysis for a subset of the Java language called petitCafé. The state of the art of binding-time analysis for object-oriented languages is studied.

The focus is in binding-time analysis, classifying variables into *static* (if its values can be determined at specialization time) and *dynamic* (if not static) of a program given a known/unknown division of its inputs. The analysis performed in this work is Constraint-Based[16].

The goal is to define a model on which it is easy to reason about binding-time analysis in object-oriented languages, but also that allows precise annotations. In order to do this, the model makes it possible to specify and reason about partially static objects.

This work defines a model to express and produce using constraints the binding-time of expressions of an object-oriented language. It includes the definition of constraints to produce the binding-time analysis in an abstract as well as in a practical way, that is, explaining how to write them in a way understandable by an equation solver, in this case REQS, a program that solves a recursive equation system. This work shows clearly and unambiguously how to produce the classification of the binding time for every variable in the program.

The model presented here sets the basis for developing binding-time analyses that certainly will improve the efficiency of real object-oriented specialized programs.

Based on this model, a prototype of a BTA for petitCafé has been implemented.

The work is organized as follows: chapter 2 presents program specialization, binding-time analysis and the components related to them, introducing the background and discussing the basic properties. Chapter 3 presents the model, gives specifications and explains the basics characteristics of the approach taken. Next, chapter 4, presents the abstract constraints defining a system that indicates how to annotate the program parts. Afterwards, chapter 5 shows how to solve the equation system, explains how to perform the binding-time analysis and gives examples. Finally, chapter 6 summarizes the work and discusses further research.

Chapter 2

Partial Evaluation

2.1 Introduction

One of the major advantages of object-oriented languages is that their abstraction mechanisms encourage building generic components that can be adapted to be used in the context where they are deployed. General software components can be easily constructed. But this generality is achieved at the expense of efficiency.

The next sections show the conflict between generality and efficiency and present program specialization and more specifically program evaluation as one appropriate technique to solve this problem.

Then binding-time analysis is introduced an its place in partial evaluation shown, together with the definition of static and dynamic expressions.

Finally we present an insight of constraint-based analysis, on which relies the analysis performed throughout this work.

2.2 Partial Evaluation

2.2.1 Generality versus Efficiency

Encapsulation and message passing allow objects to become autonomous entities that interact between them to define a program. Once the object's data and implementation is hidden, we can parameterize the object and use it in several programs.

A consequence of the generality is that several features of the object will not be necessary in some of its contexts of use. Instead, at execution time, the object has to adapt its behavior to the context where it is.

To do that the context gives the object data in the form of parameters. Then,

depending on the context of use, the object may have parts of its data and implementation that are *fixed* since the context will provide it always the same data. At this point there can be computations and data of the object that depend only on that fixed data, which makes them also fixed. This leads to a situation where superfluous computations are performed over and over and thus to a loss of performance.

To summarize, object-oriented programming facilitates generality, but with the disadvantage of inefficiency. This is in part because a highly parameterized program can spend most of its time testing and interpreting parameters, and relatively little in carrying out the computations it is intended to do[14].

2.2.2 Program Specialization

The efficiency versus modularity and generality conflict can be solved using *program specialization*.

The general idea of program specialization is to automatically optimize a generic program to a specific execution context described by program invariants.

Program specialization is then a technique for mapping generic programs into specific implementations dedicated to a specific purpose [23].

2.2.3 Partial Evaluation

Given a program and its input data, an interpreter can execute the program producing a final answer. Given a program and only part of this program's input data, a *program specializer* will attempt to execute the given program as far as possible yielding as a result a *residual program* that will perform the rest of the computation when the rest of the input data is supplied[14]. It is a source-to-source staging transformation.

The effect of running the original program on full input data is identical to that of running the specialized version of the program on the dynamic input data part.

Partial evaluation is a special form of program specification where the invariants are actual program input values (in a more general sense, this can be properties on program input [7]).

A partial evaluator is given a *subject* program together with part of its input data, `in1`. Its effect is to construct a new program `p_in1` which, when given `p`'s remaining input `in2`, will yield the same result that `p` would have been produced given both inputs. In Figure 2.1 there is a sketch of the process. Data values are in ovals and programs are in boxes. The specialized program `p_in1` is first considered as data and then considered as code, whence it is enclosed in both. In addition single arrows are indicate program input data, and double arrows indicate outputs. The partial evaluator has two inputs while p_in1 has only one and p_in1 is the output of the partial evaluator[14].

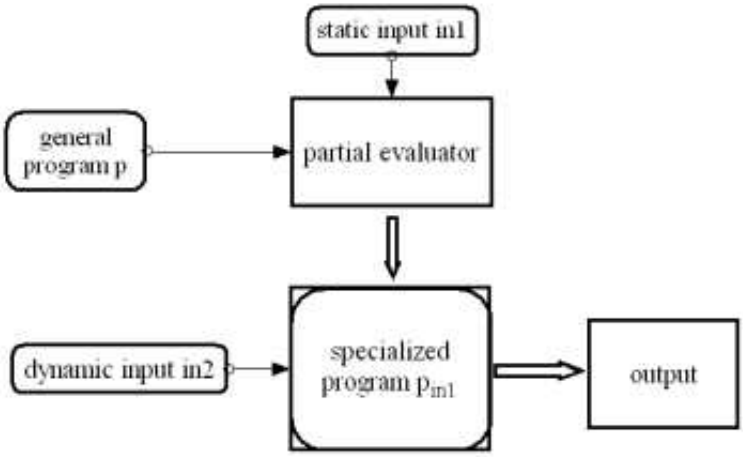


Figure 2.1: A partial evaluator

Using partial evaluation, it is possible to write a highly parameterized but maybe inefficient program, and use a *partial evaluator*, another program, to specialize it to each interesting setting of parameters, automatically obtaining as many customized versions as desired[14].

There are two approaches to achieve partial evaluation: *off-line* and *on-line*.[1] In off-line specialization, the partial evaluator has two stages. In the first one, binding-time analysis, treated in the section 2.2.4, it analyzes the program to determine which parts of it can be executed in advance and annotate them. The partial evaluator classifies and annotates the expressions into the ones that can be computed at program specialization time, which are called *static* and the others, the ones whose values are unknown at specialization time, called *dynamic*. In the second stage of off-line partial evaluation, *specialization*, the partial evaluator takes the analyzed and annotated program and the known input and produces a specialized version for that input.

On-line and off-line partial evaluation

Above we have described partial evaluation as a process that has two stages.

The first one is to compute classification between expressions that are known at specialization time, so-called static, from the ones whose values cannot be determined at specialization time, the so-called dynamic expressions. This classification is based of the knowledge that some input will be known at specialization time or not.

[1] In this work we assume off-line partial evaluation. For explanations on on-line partial evaluation, see [14].

This implies a certain *division*, a classification between static (known) and dynamic (unknown) input variables. According to [14], an essential requirement in program specialization is that the division is *(uniformly) congruent*, which means that "*any variable that depends on a dynamic variable must itself be dynamic*". This is also called *congruence condition.*[14]

Congruence. A division is *congruent* if the value of every static variable is determined by the values of other static variable (and thus ultimately by the available input). Equivalently, a variable whose value depends on a dynamic variable must itself be dynamic. This means that the static parameter cannot be bound to a residual expression during specialization.

An expression exclusively built from constants and static variables is also called static, while it is called dynamic if it contains a dynamic variable. For example suppose that the program to specialize contains the assignment

```
X := exp
```

If `exp` is dynamic then by the congruence condition `X` must also be dynamic.

So the partial evaluation is a process with two stages. *First* compute a division B from the program and the initial division$\overline{B}$, *without* making use of the concrete values of the static input variables. *Then* the actual program specialization takes place, making use of the static inputs to the extent determined *by the division*, not by the concrete values computed during specialization. This approach is called *off-line partial evaluation*, as opposed to *on-line partial evaluation.*

A partial evaluator decides which available values should be used for specialization. If the concrete values computed during program specialization can affect the choice of the action taken (by the partial evaluator) then the partial evaluation is *on-line.* Otherwise it is *off-line*[14].

Many partial evaluators mix on-line and off-line methods, since both have advantages. The main advantage of on-line partial evaluation is that it can sometimes exploit more static information during specialization than off-line, thus yielding better residual programs. Offline techniques make generation of compilers, etc., by self-application feasible and yield faster systems using a simpler specialization algorithm. In [14] the authors argue that "only off-line specializers have been successfully self-applied." By 'successfully' they mean that a resulting program, typically compilers, have been reasonably small and efficient. They make a case study of compilation and compiler generation by on-line partial evaluation. They find that the result of compiler generation is not satisfactory and show how off-line evaluation provides a simple solution.

It should be noted the importance of binding-time analysis given the fact that almost all off-line partial evaluators base their decisions on the result of a preprocess,

the binding-time analysis.

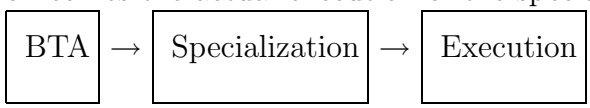
Partial Evaluation is automatic

The process of partial evaluation is done automatically. A program together with part of its input data are transformed into a more efficient specialized version by pre-computing parts of the program that depend only on one given input.
This process has three main stages: analysis, specialization and execution.

First, a binding-time analysis determines which parts of the program can potentially be executed using only some of the program runtime input (so-called *static* data).

Once values for the static data have explicitly been given, the program is transformed by pre-computing parts depending only on the program input into the specialized version (which can then be compiled). This is the specialization time. At this time if a variable was catalogued as static during the binding-time analysis time and its value has a textual representation, i.e. it is an integer, and is in a context known as dynamic, then it is *lifted* into a residual representation. That means that the textual representation of the static variable is computed at specialization time.

After the specialization comes the actual execution of the specialized program.



Although binding-time analysis is fully automatic, the program to specialize may need some 'tuning,' i.e. modification by hand, to get better binding-time separation and so better speedup. After binding-time analysis, specialization can be done automatically and completely without human intervention as often as wished, e.g. every time the static data changes. A well-known example is compiling by specializing an interpreter.

Why partial evaluation

Partial evaluation is applicable to a wide range of problems. The essence of partial evaluation is to solve a problem indirectly, using some relatively static information to generate a faster special purpose program, which is then run on various data supplied more dynamically[14]. The specialized program can be much faster than the original one (speedups exceeding 10 have been seen in practice). The speedup can be either measured, or predicted from the binding-time information.

Problems that can be solved faster using specialization are for instance: highly parameterised computations that use much time consulting parameters, but are often run using the same parameter settings; programs with many similar subcomputations;

programs of a highly interpretive nature, e.g. circuit and other simulators, where specialization removes the time to scan the object being simulated; database query search algorithms; metaprogramming, where a problem is solved by designing a user-oriented language and an interpreter for it.

Partial evaluation has been used to solve problems in the areas of computer graphics, database queries, neural networks and scientific computing between others. But the most developed area is programming language processors and specially compiling.

State of the art

According to the Computer Science Department of the University of Copenhagen, partial evaluators exist for Scheme, C, SML, Fortran77 and logic programming. For Scheme there are two systems, Similix at Copenhagen and Schism at Rennes; the same for C, C-mix at Copenhagen and the TEMPO system at Rennes; whereas for SML there is SML-mix, a prototype at Copenhagen; and for Fortran 77 there is a prototype developed at Vienna and Copenhagen; finally for logic programming there are systems developed by SICS (Swedish Institute of Computer Science), K.U. Leuven (Belgium) and University of Bristol.

All are academic projects, with promising results so far. The Scheme systems are the most mature. The C and Fortran 77 systems are most promising for industrial use. However both need more development and experimentation to bring them up to industrial strength.

An important work is JSpec [21][1], an off-line partial evaluator that treats the entire Java language excluding exception handles. JSpec takes Java source code or Java bytecode as input, and can produce Java source code or C code as output. Specialized C code is executed in the Harissa environment, and can incorporate a number of low-level optimizations, such as array bounds check elimination. JSpec exploits object-oriented opportunities for specialization, such as global type information propagation, elimination of virtual dispatching and (safe) global propagation of known encapsulated values.

2.2.4 Binding-Time Analysis

Binding-time analysis is the process of determining at what time the value of a variable can be computed, that is the time when the value can be bound to the variable[14]. This time can be program specialization time, in this case the variable is classified as *static* or later, in this case the variable is classified as *dynamic*. The output of a binding-time analysis is basically a division: a classification of each program variable as static or dynamic. This division makes it then possible to classify each program construct as static or dynamic.

In off-line partial evaluation, the program specializer uses this division to determine in advance which are the static parts of the program being analyzed. This is essential to the success of self-application [14] because the static parts of the program to analyze are determined prior to the specialization phase, and the program specializer uses this information when it is run. For example, supplying the division of the subject program's variable is the simplest way "of communicating the necessary insight to mix", the compiler generator described in [14]. Moreover, if all the assignments and conditionals are marked as either *eliminable* or *residual*, the specializer can determine its actions at a very low cost, making specialization even more efficient.

Binding-time analysis is important because it determines how an off-line partial evaluator will specialize a program. In consequence, the accuracy of the binding-time information directly determines the degree of specialization. There are many techniques to perform binding-time analysis (i.e. dataflow analysis and abstract interpretation). In this work we are going to use constraint-based analysis.

2.3 Divisions

So far we have assumed that the task of binding-time is to compute only one division (between static and dynamic) and that this division is valid at all program points. This simplified view of binding-time analysis can be enriched with other ways to calculate the division aimed to obtain a more precise binding-time analysis, that is to let the specializer find as many static expressions as possible so that it can specialize as much as possible.

2.3.1 Flow Sensitivity

The binding-time analysis can be *flow sensitive* opposed to *flow insensitive*. In a flow sensitive analysis, different binding-times can be associated to a variable at different program points. Whereas if the analysis is flow insensitive, the variable abstraction is independent of the program flow.

In a flow sensitive binding-time analysis, a different binding-time description is computed for each program point, allowing the same variable to be considered static at one program point and dynamic at another.[11]

The analysis used in this work is *flow insensitive*.[2]

[2]By using *SSA* (Static Single Assignment) whereby each variable definition is used just once, we make a flow insensitive analysis, but in fact obtain flow sensitive results. With SSA if we make flow insensitive analysis over a a program P1 that is the Static Single Assignment version of a program P, it is like if we had done the flow sensitive analysis of P. For a detailed explanation of SSA, see [8].

2.3.2 Use Sensitivity

There is also a *use sensitive* binding-time analysis, opposed to *use insensitive*. If it is use sensitive, it allows both static and dynamic *uses* of a given variable. The variable definition must then be annotated as static *and* dynamic. The definition will be both executed at specialization-time and residualized.

An insight is that at specialization time the value of a variable is allowed to be computed in certain contexts even if the variable identifier is residualized in others. In this case, it is important to handle pointers and structures to get as much specialization and as few residualized constructs as possible. How to handle these matters is described in [12].

To understand why use sensitivity is needed in the cases where it is important to keep static as many constructs as possible, it is necessary to understand how the context of variable use affects its binding time. Here is an explanation taken from [10].

A static variable is a dynamic context is evaluated during specialization and the resulting value is converted into its textual representation.When this happens, it is said that the variable is *lifted*. Values for which there exists a corresponding textual representation, such as integers, can be lifted, but values which do not have a textual representation, such as pointers, structures and arrays, cannot be lifted, so any time a non-liftable static variable is used in a dynamic context, its binding-time is dynamic. If the binding-time is use insensitive, all the uses of a variable must have the same binding time. And thus forcing the uses that appear in static context to be considered dynamic.

For example, consider the case where a variable is assigned a static value and then it is used multiple times. If *any* of the contexts are dynamic, then *all* of the uses become dynamic.

Use sensitivity keeps a variable that *must* be residualized from interfering with other uses of the variable which *could* be evaluated[12]. Even if a variable becomes dynamic due to its dynamic context, the other variable uses in static contexts remain static.

Use sensitivity is more precise than flow sensitivity. Flow sensitivity associates a different binding-time value to a variable each time it is assigned. For each assignment, however, a variable may be used used multiple times. Use sensitivity associates a different binding time to each of these uses[10].

In this work it is performed a use insensitive analysis to favor the simplicity of the model in order to facilitate reasoning.

2.3.3 Polyvariant Analysis

The binding-time analysis can be *monovariant*, also said to be *context insensitive*. It is a kind of binding-time annotation where only one annotation is permitted per program point. It allows only one binding-time description for each function. To illustrate this point, consider the example of a program in a flow language taken from [14]. Assume an initial division where variable `X` is known, ergo it is considered static and `Y` unknown, considered dynamic.

```
read (X Y);
init: if Y > 42 goto xsd else dyn
dyn: X := Y;
    goto xsd;
xsd: X := X + 17;
    ...
```

In a monovariant division, `xsd` would have a division indicating that `X` as well as `Y` are dynamic.

The opposite of monovariant is *polyvariant*, when a function can have more that one binding-time description depending "not only of the program point but also on *how* the program point was reached"[14]. In the previous example, a polyvariant division assigns to each label a *set* of divisions.

For the above program, a polyvariant division would be:

- at the program point `init`, `X` static and `Y` are dynamic, this could be written as the pair (S, D);
- at the program point `dyn`, both `X` and `Y` dynamic, shown by the pair (D, D)

and finally `xsd`, has a set of divisions, that is $(S, D), (D, D)$. This set has two elements, one indicates that `X` is static whereas `Y` is dynamic and the other that both of them are dynamic.

One of the software engineering advantages of the Object-Oriented Paradigm is code reuse. In an object-oriented language, individual classes can be created and then joined to form different programs. A class can be developed as an encapsulated program unit that implements some behavior. The abstraction permits the class to be reused. The partial evaluator could take advantage of this style of code reuse and know that two objects of the same class might require different treatment. There are several polyvariant analysis that intend to produce a more precise binding-time analysis. Examples of those are type-polyvariant and method polyvariant analysis. A detailed presentation of them can be found in [23].

However, polyvariance can produce too much detailed information that in turn could lead to overspecialization, and thus to a need to control it.[23] In order to

keep the model simple with the final objective of allowing an easy reasoning about binding-time analysis of object-oriented programs, we stick to monovariant binding-time analysis.

2.4 The Constraints

Constraint-Based Analysis(CBA) is one of the four main approaches to *program analysis*[16][17]. We explain program analysis in the following section and next describe constraint based analysis. The ideas of this section will be needed in chapter 4.

2.4.1 Program Analysis

Program analysis offers static compile-time techniques for predicting safe and computable approximations to the set of values or behaviors arising dynamically at runtime when executing a program on a computer. A typical application is to allow compilers to generate code avoiding *redundant* computations, e.g. by reusing available results or by moving loop invariant computations out of loops, or avoiding *superfluous* computations, e.g. of results known to be not needed or of results known already at compile-time [16].

In order to remain computable, in program analysis one can only provide approximate answers like "the only values for variable y at 'this' program point are 1 or 2". Refer to [16] for detailed examples. In general, we expect the program analysis to produce a possible *larger set of possibilities* that what will ever happen during execution of the program. Nevertheless, although an analysis does not give precise information, it may still give useful information. For example, knowing that the value of y will be 1 or 2, still tells that y will be positive, and an integer.

2.4.2 The Constraint-Based Approach

The idea is to extract a number of inequations or *constraints* out of the program.

The syntactic structure of the program gives rise to a set of constraints whose least (in a certain imposed order) solution is desired.

In other words, out of a program we obtain a set of constraints for expressing the effect of *elementary blocks* (i.e. assignments, tests).

A *constraint* is an inequality on properties of program parts.
In general we have a constraint of the form:

$PROPERTY_1(e_1) \supseteq PROPERTY_2(e_2)$

where $PROPERTY_1$ and $PROPERTY_2$

are *properties* of the expressions e_1 and e_2.

This constraint states that if a property holds in $PROPERTY_2(e_2)$, then it must hold in $PROPERTY_1(e_1)$.
In a more straightforward explanation: the value of the property $PROPERTY_1(e_1)$ is greater than or equal to the property $PROPERTY_2(e_2)$.

For example a *property* of a variable could be that it has dynamic BT. In this case the constraint would be satisfied when it holds for both e_1 and e_2, or when it does not hold for neither e_1 nor e_2, or when it holds for e_1 and not for e_2 (We assume here that static is smaller than dynamic). In other words the constraint is satisfied in any case except that $PROPERTY_1(e_2)$ holds that property and $PROPERTY_1(e_1)$ does not.

2.5 Summary

This chapter presents the fundamental concepts that will be used in further sections.

Among the most basic of them are the ideas of partial evaluation, binding-time analysis and division among static and dynamic binding-time states, that will be used in 3.

As well as that we have explained the different aspects related to divisions such as flow sensitivity, use sensitivity and polyvariant analysis.

Finally we have presented an insight of constraint based analysis, on which the analysis defined in this work relies. This constraint-based analysis explanation will support the design of correct and precise binding-time analysis for petitCafé presented in chapter 4.

Chapter 3

Binding-Time Analysis for Object-Oriented Languages

3.1 Introduction

Binding-time analysis is used to discover which parts of a program are static and which are dynamic. There are several techniques to perform binding-time analysis (i.e. data flow analysis, constraint based, abstract interpretation and type and effect systems [16]). In this work we are going to use constraint-based analysis.

Our binding-time analysis is performed by writing constraints on parts of the program. These constraints define a system of inequations whose solution is the lowest possible binding time for each part. By lowest we mean that parts should be assigned static binding time whenever possible.
We present here a binding-time analysis for an object-oriented language subset of Java. We define the binding time of an object in terms of the binding time of its instance variables. Instead of considering and object dynamic because at least one of its instance variables is dynamic, we regard the binding time state of an object as the *tuple* of binding time states of its instance variables. Taking into account the imperative features of object-oriented languages, this analysis includes references to objects. To keep the balance between performance and binding time expressiveness, the binding time of an object is the binding time of its class.

We define how to generate the constraints that apply to a program and, once the equation system is solved, determine the binding time of the program constructs.

This work differs from previous works in that the problem of producing the binding-time analysis is solved by means of Constraint-Based Analysis as in C-mix [2] whereas other previous work solved this using Data Flow Analysis as in JSpec [20], whose implementation is based on [12]. Secondly, to express the binding time of complex structures such as objects we use a tuple of binding times where each

element is the binding time of an instance variable, whereas previous works treat it conservatively annotating as dynamic the whole object [23] , even though parts of it could be static.

There are two levels addressed in this work: syntactic and semantic. The syntactic level is required to write the constraints, since the analysis is Constraint-Based and constraint generation is syntax-directed. The semantic level helps to reason in terms of execution and objects.

How it is organized

In the following sections, we present the heart of the model: petitCafé and the definition of binding times for objects, primitive types and references.

Section 3.2 shows the syntax for our language, petitCafé.

Then, section 3.3 defines the domain necessary to further present the Constraint Based Analysis, expressing the model at an abstract level as well as at a syntactic level. Objects are treated as abstractions of memory parts that can be seen in a more concrete way or in a more abstract way.

Afterwards, section 3.5 defines the binding-time of objects and primitive types.

Finally section 3.6 presents a summary.

3.2 petitCafé

To provide a concise description of partial evaluation of class-based object-oriented languages, we use a small class-based object-oriented language with inheritance based on Java [9][15]. Our language is called petitCafé. Intuitively, petitCafé is a subset of Java and any petitCafé program behaves like the syntactically equivalent Java program. Nevertheless it has a reduced set of features compared to that of Java. As in [23], the choice of what programming language features to include in petitCafé is influenced by what features are found in object-oriented languages and what features are interesting from the point of view of partial evaluation.

Object-oriented languages have as essential features encapsulation of data and methods[3], so petitCafé has them. It also includes standard language features such as conditional, operators and boolean and integer constants.

Compared to the EFJ, the Extended Featherweight Java language of [23], petitCafé extends EFJ with imperative features.

The following section provides the syntax of petitCafé. For a detailed Java syntax see [6].

3.2.1 Abstract Syntax

This section presents the grammar for the petitCafé programming language. The grammar below uses the style conventions:

- $x|y$ means either x or y
- x^* denotes zero or more occurrences of x
- $[x]$ means zero or one occurrences of x

A symbol in **bold** is a keyword or a program separator, i.e. **(**, **)**, {, } **return** and **if**.

An assignment has the form:

$$assignment ::= \text{location} = assignment\_exp$$

where the left side is restricted to be a location, an obvious limitation imposed to petitCafé with respect to Java. This is on purpose to limit the model in order to keep it manageable. At the point of defining the left-hand side of an assignment there were two options: to consider it a variable that was itself an Identifier that could be qualified, i.e. Identifier {.Identifier *} as in [9], with another rule for field assignment, or to show explicitly in the syntax that a left-hand side of an assignment can be either the identifier of a variable or an object field.

The first option is closer to the syntax of [9], but the second one enforces the idea that the left-hand side of an assignment can be a variable present in a program or an object field. Variables and object fields are *memory locations* (see 3.3.1). This approach makes it possible to express clearly the constraints for object fields in 4.2.1. In section 3.3.1, the Syntactic Domain Loc, which corresponds to the set of memory locations that can be defined in a program clearly expresses the role of the variables present in a program and of the object fields present in a program. Part of this idea was inspired by the subset of C syntax from [11]. In order to be consistent, the syntax of petitCafé includes the fact that a location can be an variable or can be an object field (named field in the syntax). But the object field is in essence the Qualified_Identifier of [9], which can be an Identifer or a qualified identifier, (Qualified_Identifier ::= Identifier {.Identifier*}).

3.2.2 Transformations

We perform the binding-time analysis under certain conditions. We assume the originally the Java program is transformed into a program that one can write using the grammar we present in section 3.2.1, which is a subset of the Java grammar. We

Domains:

$literal \in Integer \bigcup Boolean$

$bop \in BinaryOperator$

$uop \in UnaryOperator$

$identifier \in Identifier$

Abstract syntax:

```
program ::= class_def
class_def ::= class class_name [extends class_name] {
              instance_variable_def *
              [ constructor_def]
              method_def * }
instance_variable_def ::= type variable
constructor_def       ::= class_name () block
method_def            ::= type method_name (args) block
block                 ::= {stmt *}
method_name           ::= identifier
args                  ::= variable *
stmt                  ::= assignment
                      |   return  variable
                      |   if variable  block block
                      |   while variable block
                      |   variable.method_name args
assignment            ::= location = variable_assignment_exp
location              ::= variable | field
variable              ::= identifier
field                 ::= variable.field_name
field_name            ::= identifier
variable_assignment_exp ::= variable bop variable
                        |   uop variable
                        |   variable
                        |   this
                        |   literal
                        |   method_invocation
                        |   class_instance_creation_exp
method_invocation       ::= location.method_name (args)
                        |   field_access.method_name (args)
                        |   this.method_name (args)
class_instance_creation_exp ::= new class_name (args)
primitive_type              ::= boolean | integer
type                        ::= class_name | primitive_type
class_name                  ::= identifier
```

Figure 3.1: Syntax of Java subset

analyze this program written using the subset grammar. As a consequence of the analysis we perform, we obtain a program that is written following the grammar subset of Java and finally, since our grammar is a subset of the one of Java, this program can be transformed to comply with the full grammar of Java. Using a subset of Java is actually not a limitation although the fact that petitCafé is proper subset in this respect still need to be checked.

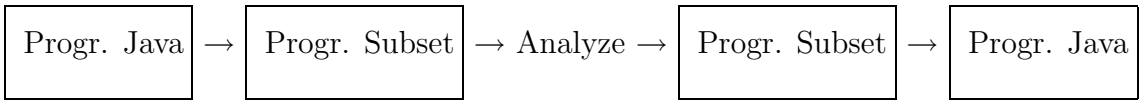


Figure 3.2: Transformations of the source program

To perform the binding-time analysis, we assume that all identifiers have been renamed in order to be unique. Thus, there should be a transformation that renames variables that have the same names and are in nesting blocks. This transformation could be achieved by changing the name of the 'repeated name variables' adding a suffix corresponding to the number of the block where they appear. All this provided that we assign virtual numbers to the block.

3.3 Domains

This model is intended to facilitate the easy reasoning about binding times of object-oriented languages. It intends to lay the foundations for the design of correct and precise binding-time analysis for Java. With such an aim, a clear and precise definition of the binding-time of objects, primitive types and references based on language subset of Java defined above is essential.

The basic features of the presented model are:

- This model is based on petitCafé, a subset of Java that this work defines.
- The binding-time analysis is monovariant, flow-insensitive and use-insensitive.
- For any instance of a given class, the binding time corresponds to that of the class. References have the same binding time as objects.
- The binding time of objects is a set of binding-times of the object's instance variables.
- Primitive types can have static or dynamic binding time.
- The binding time analysis is performed using constraints. The constraint generation is syntax-directed.

3.3.1 Concrete and Abstract Domains

In this section we explain the domains on which this work relies.

It comprises the different binding-time types and values used to indicate the binding-time state according to the part of the program to annotate (i.e. objects or variables of basic types).

At the semantic level, we work at a concrete and at an abstract level. At the concrete level there are the values, objects and references, but at an abstract level there is a certain binding-time, object binding-time and reference binding time respectively.

Syntactic Domain

To facilitate the explanation of the constraints applied to *elementary constructs* (i.e. assignments, field access, etc.) of programs complying with petitCafé, a syntactic domain of locations is introduced. This domain, called *Loc*, corresponds to the set of memory locations that can be defined by a program.

$$Loc = Var \cup Field$$

where Var is the set of variables present in a program and $Field$ is the set of object fields present in a program. An element of $Field$ is of the form $v.f$, where $v \in Var$ and f is a field name of the declared class of v. $\in$ is the membership operator.

In conclusion the basic objective of the analysis is the computation of the function *bt* defined as follows:

$$bt : Loc \rightarrow BT$$

We use the function *bt* to express the binding-time BT (i.e. S) of the location *loc*. Intuitively, a binding-time is a division, a classification of locations into static or dynamic as was defined in section 2.2.4. The values of the binding-time type BT are defined in 3.4.2.

Locations can be atomic or structured. If atomic, they correspond to variables of primitive_type. Structured locations correspond to objects and they are structured because objects have fields (i.e. each field is an instance variable).

Fields, dereferencing and addressing

The mapping between the syntactic level and the semantic level is as follows: a location is the means we use in this model to access memory. Memory access can then be modeled by a function

$$value : Loc \rightarrow Value$$

which returns the value associated to a given location. Note that there are actually different kind of memory accesses depending on the location, which can be a local variable, an object reference or an object field (above, Var could be further refined into $Var = LocalVar \cup Reference$).

More basic functions can be defined, such as

$$fd_i : Object \rightarrow Value$$

which takes an object and an object and returns the value of the i_{th} instance variable (this is intended to express at a syntactic level how to access object fields), and

$$deref : Reference \rightarrow Object$$

which takes a location that is a reference and returns the object pointed to by the reference. The function $deref$ is actually a restriction of the more general function $value$ to references.

One can see then that, in case of a reference, the function value is the composition of the functions $deref$ and fd_i.

Note that we have talked here about values in a very general manner. These values can actually be either concrete or abstract. When talking about abstract values, we actually have

$$value = bt \text{ and } Value = BT$$

Since in object-oriented languages objects are accessed via references, in this model locations represent the syntactic element to access objects. This access is performed in such a way that if `o` is an object of class `O` an `l` is a location to that object, then the two following equations hold:

$$deref(l) = \texttt{o}$$

and

$$l = address(\texttt{o})$$

We can think at

$$deref : Loc \rightarrow Object$$

as function that takes a location and returns the object, the value pointed to by the location. It receives a reference to an object and returns the value pointed to by the reference. It represents the access to the object through the location, which is the parameter. Here the object is seen as a tuple of instance values and it is to this tuple

that the location points to. We suppose that the binding time of the object pointed to by a given location l is equal to $bt(\mathtt{o})$.

Furthermore, *address* is another function that takes an abstract object and returns its location.

In petitCafé, which is a subset of Java, to symbolize that a reference is pointing to an instance variable, in a program we write the location followed by a point and the object field:

$$location.objectField$$

To summarize:

Syntactic level	location	
Semantic level	concrete	values, objects and references
	abstract	BT, objectBT and referenceBT

Example of Using fd_i

Let `p` be an object of class Point, having this class two instance variables named `x` and `y`. In this context x is what we will call the first instance variable and y the second one. In fact in object-oriented languages there is no notion of first instance variable, but we produced it here there to help in the understanding of how we use the function *fdi*. According to this example the following code

```
Point p;
...
p = new Point();
p.x = 1;       // this assignment
```

can be translated into the function notation for references and objects as shown. The marked assignment is represented as fd1(p) = 1; where fd1 is a function that returns the value of the 'first' instance variable of object p (strictly speaking the object pointed to by p).

Syntactic and Abstract locations

Expressing constraints 4.2 is done in a syntax-directed manner, i.e. the program is traversed and, considering the intra-procedural analysis, a set of constraints is generated for each encountered statement, depending on the grammar. The constraints are written using locations, as they were defined in the previous paragraphs.

The *syntactic* locations are those that can be extracted from the program through its syntax. For instance `p.x` and `q.x` are two syntactic locations.

On the contrary, *abstract* locations are those that represent a variable of the program at an abstract level.

Note that there is not a one-to-one mapping between the syntactic locations and the abstract locations.

For instance `p.x` and `q.x` are two different syntactic locations, but if `p` and `q` have the same declared class A, it could be that they refer to the same abstract location denoted, for instance A.`x`.

Abstract locations define a relationship between the syntactic locations in the sense that two syntactic locations that refer to the same abstract location belong to the same equivalence class.

This concept can be used to tune the precision of the analysis. Nevertheless, this idea will not be explored in this dissertation, as show in section 3.5.2, where the binding time of an object is that of its class.

3.4 Binding Time Domains

In the following sections we give the definition of binding-times of primitive values, objects and references.

We begin by presenting the abstract binding-time type BT. Its role is essential in the model, since it serves to define the binding-time of all the components.

We will observe that in the first place we have π, that corresponds to the binding time of simple values such as constants and liftable variables. In the second place we have Γ, the binding-time type for objects. This last type is more complex, since its elements are tuples of binding times. The final binding-time type used in the model is the union of the ones aforementioned.

3.4.1 The Basic Binding Times *S* and *D*

There are two binding-time values S for static and D for dynamic[4].

We call π the type of binding time representing the set of values $\{S, D\}$

$$\pi = \{S, D\}$$

Origins of binding time

Objects and variables have different structures to define their binding time will be shown in section 3.5.2. A consequence of this is that there are two types of binding times.

On the one hand there is the binding-time type corresponding to the binding time of primitive values that belong to primitive types such as constants and liftable variables (see an explanation of liftable variable in section 2.3.2).

On the other hand, there is the binding-time type of objects. This binding-time type reflects the fact that an object has instance variables, each one of these having its own binding time. It is then more complex, since its elements are tuples of binding-times.

We will call π the binding-time type for basic types and Γ the binding-time type for objects. The binding-time type for objects is a tuple of binding-time elements. Each binding-time of this tuple corresponds to the binding-time of an object's instance variable. The first binding-time corresponds to the binding-time of the 'first' instance variable according to some predefined order. An instance variable i can be of a basic type or an object. If it is of a basic type, then the corresponding i element of the tuple will be S or D, indicating the binding-time of that instance variable. On the contrary, if the instance variable i is an object, then the binding-time indicated in the i^{th} position of the tuple will be another tuple, reflecting the binding-time of the object that has the instance variable i. Observe that this recurses downwards, ending always in the binding-time of a basic type.

Binding time of locations

Atomic locations have binding-time values S and D. But as specified in the previous section, locations can be atomic or structured. A structured location correspond to objects. We can think of each instance variable as a sub-structure of a structured location. Objects can have several instance variables, each one having its own binding-time.[1] To reflect this, the binding-time of structured locations is a tuple of binding-times, one binding-time per sub-structure of the location. For example each element (i.e. object field) of the structured location has its binding time. We call this binding-time type Γ.

$\Gamma = \{t | t \text{ is } n\_\text{tuple whose } i_{th} \text{ element is a binding-time, } 1 \leq i \leq n\}$

3.4.2 The Binding Time BT

In general, the binding-time BT is the union of the binding-time values S and D (the values of the type π) and the binding-time values of structured locations, (the values of Γ) which are themselves tuples.

$$BT = \pi \bigcup \Gamma$$

[1] Observe that either a variable or an object field can be structured locations.

In general we map elements of the concrete domain to elements of the abstract domain in the following way:

Concrete domain	Abstract domain
BasicType	PrimitiveTypeBT ($= \pi$)
Value	BT ($= \Gamma$)
Object	ObjectBT
Reference	ReferenceBT

Where ObjectBT and ReferenceBT are of type BT

$$BT = PrimitiveTypeBT \bigcup ReferenceBT \bigcup ObjectBT$$

3.5 Binding Times

3.5.1 The Binding Time of Primitive Values

Primitive types are integer and boolean, any variable that is an integer or boolean can have as its binding time the value static or dynamic.

3.5.2 The Binding Time of Objects

There are two main ideas that help define the binding time of objects. The first one is that the binding time of objects is regarded as a tuple of binding times and the second one is that the binding time of an instance is the binding time of its concrete type.

Binding-time of objects regarded as a tuple of binding times

Objects have instance variables which are elements of base type or objects themselves. Since they are composed of instance variables, their binding-time can be defined by a tuple of length equal to their number of instance variables. Each element of the tuple is the binding time of a specific instance variable.

The binding time we are interested in for objects takes into account and maintains the binding time of each instance variable of an object. In fact, we define **the binding-time of an object as a tuple of binding-times where each position in the tuple corresponds to the binding time of an instance variable**.

Thus the binding-time of an object is defined as follows:

Let us assume

o is an object with n instance variables,

iv_i is an instance variable of object o such that $1 \leq i \leq n$

where n is the number of instance variables.

$bt(o)$ is the binding time of object o,

$$bt(o) = (bt(instanceVariable_1), ..., bt(instanceVariable_n))$$

$$\forall instanceVariable_i \text{ instance variable of } o, \quad i \text{ belonging to } [1..n]$$

where $bt(instanceVariable_i)$ is the binding time of the i^{th} instance variable.

An alternative approach would be to merge the instance binding times into one to produce the binding time of the object. This approach associates the *dynamic* binding-time value to an object if one or more of its instance variables is dynamic. For instance, consider a point p with two instance variables, x and y. Let us say that x has a static binding-time whereas y has dynamic binding time. Then with such an approach, the object would be considered dynamic, since its instance variable y is dynamic. This would make the model lose precision. Note the difference with our approach, where the binding time of an object is 'divided' into one binding-time annotation per object instance variable.

Binding time of objects according to their class

In our model we define that an object has the same binding time as the binding time of its class (considering an abstract level). This approach has the advantage of simplicity: only one record of binding-time per class has to be maintained and thus avoids to have too much information to manage (as it would be the case of registering a different state for two instances of the same class) which would be unmanageable. This will be discussed further below. Thus all the objects of the same class have the same binding time.

More precisely, for any two locations l and m that points to objects of declared class C with n instance variables, the binding-time of the locations coincide:

$(bt(fd_1(l), ..., (bt(fd_n(l)))) = (bt(fd_1(m), ..., (bt(fd_n(m))))$

where bt stands for binding time function that receives an instance variable and returns its state, fd_i is the function defined above, o is an object n instance variables.

We can define the function:

$$bt\_class : class \rightarrow BT$$

that takes a class and maps it to the binding-time of that class. And assuming there exists a function

$$declared\_class : location \rightarrow Class$$

then we can write that for any location l pointing to an object of a given class C such that $declared\_class(l) = C$ the binding-time of a location l should satisfy:

$$bt(l) = bt\_class(declared\_class(l))$$

Figure 3.3 sketches this situation. We can observe that all the objects of the same class have the same binding time.

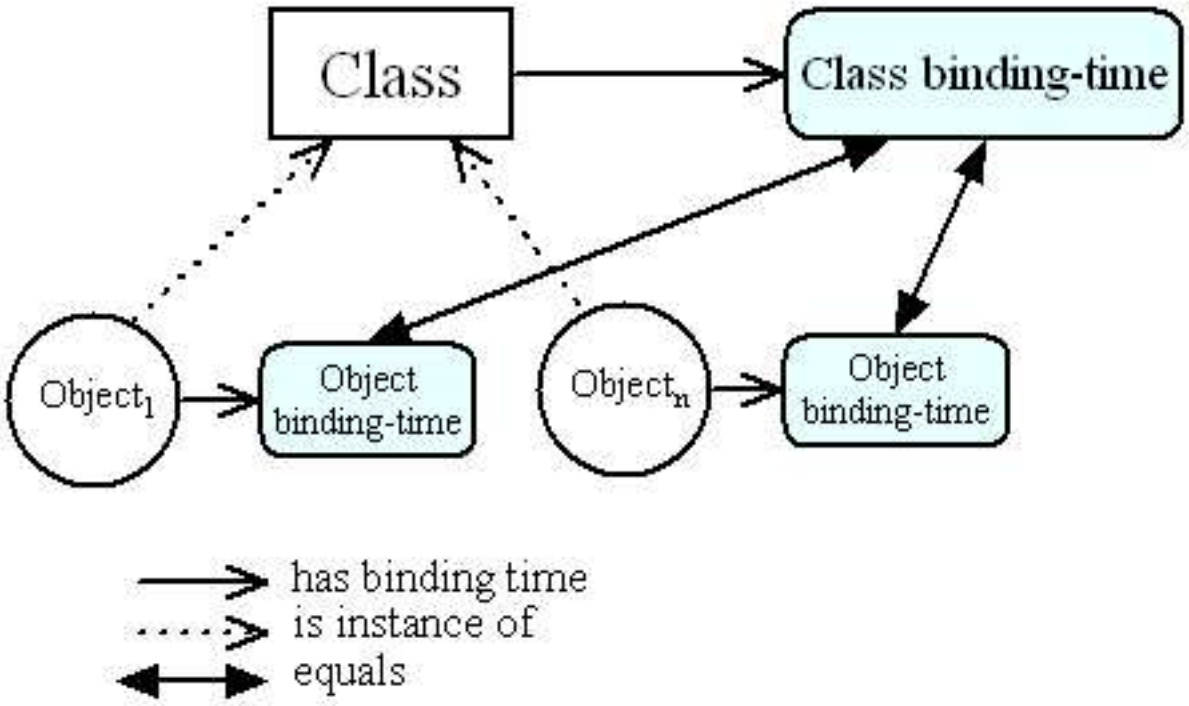


Figure 3.3: Binding-time of objects and their class

Equivalence Classes

The concept of merging the binding-time of all the instances of the same type has its counterpart at the syntactic level. This is because given two or more (syntactic) locations that reference objects of the same type, we take only one example, an ambassador (abstract level) of the set of syntactic locations that reference objects of the same type.

Thus the *equivalence class* is the set of (syntactic) locations that reference objects of the same type.

There are two kinds of equivalent locations, one kind corresponding to locations which are references with the same declared class C, which can be denoted C, and one kind correponding to fields, which can be denoted $C.f$ for the field f of the class C.

In order to avoid confusion the between mathematical notion of classes and the object-oriented classes we will rather talk about equivalence locations.

If l is a location, $\overline{l}$ is its equivalent location. Then we call $\overline{Loc}$ the domain of equivalence locations.

We define a function $\overline{bt} : \overline{Loc} \rightarrow BT$, which computes the binding time of an equivalent location. As a result, by definition, if l belongs to Loc and correponds either to a primitive field or a reference/object:

$$bt(loc) = \overline{bt}(\overline{loc})$$

In the following, for the sake of simplicity we will not distinguish any longer between bt and $\overline{bt}$ or Loc and $\overline{Loc}$, i.e. we will called locations the set including both what we have so far called locations and equivalent locations and consider bt as the function associating binding times to all these locations.

Let us assume that `p.x` and `q.x` are locations,

`p.x`, `q.x` $\in Loc$,

`p.x` and `q.x` have the same type,

then `A` is the equivalence class at an abstract level.

In the model this generates that:

$$bt(p.x) = bt\_class(A)$$

and

$$bt(q.x) = bt\_class(A)$$

with $bt\_class(A)$ representing all the locations that reference objects of a certain type.

Therefore there is only one constraint in the model. Of course the binding-time of `p.x` and `q.x` must be the same. Using locations in this way the analysis is correct, i.e. it will not deduce that the binding time of a field is Static when it is not.

As already mentioned, the advantage is that this analysis is easier. Otherwise, an alias analysis is needed (i.e. `p` and `q` could reference the same object). The disadvantage is that the analysis is less precise. This is because with this approach all the instances of the same class must have the same binding time. Since there is no type analysis, it is necessary then to take a safe approximation, which forces the equivalence class to be dynamic as soon as one of its member location is dynamic, which is to say that the binding-time of all the locations (syntactic level) is dynamic and then that all the instances (abstract level) are dynamic.

In general, the merging of binding times depends on the availability of information about real types. There are two cases: either there is no type analysis, or information about concrete types is available.

In the model presented here there is no type analysis, thus the type of a location can be any concrete type below the declared type of the object (the same approach was presented in [23]).

On the contrary, if information about type analysis were available, then the binding-time analysis of objects would be more precise. "The more precise the type inferencing algorithm, the smaller the set of types at each program point, and thus the fewer restrictions there are on the binding time of each class" [22].

In conclusion, in this model the binding time of an instance is the binding-time of its class.

Binding-time and inheritance

The binding-time of the *equivalence* classes of a program is influenced not only by how object instances are used in the program, but also by the inheritance relation between classes[18]. In this section we discuss this in an abstract level.

The point is that, in the abstract level, if A is a superclass of B, then the binding-time of B should be forced dynamic given that the binding-time of A is dynamic. This is because if one of the instance variables of A is dynamic[2] then than instance variable is inherited, and thus B has a dynamic instance variable. Figure 3.4 shows a tree representing the Inheritance relationship against the relationship between class binding times.

[2]Again at an abstract level, since that instance variable can also be an object and thus its binding time can be that of a structured location.

If A is superclass of B, then $bt\_class(\texttt{B}) \supseteq bt\_class(\texttt{A})$

Figure 3.4: Inheritance relationship is inverse to the class binding-time relationship

Not only that, but also having no type analysis the real type of an instance is unknown. Therefore, if its binding-time is dynamic (again referring to the abstract level) then every possible real type that the variable could take should be also dynamic.

At the syntactic level, we say that for a given location the set of possible real types is the declared type plus the complete set of subtypes of the declared type. A location that becomes dynamic forces its equivalence class (that in fact is the declared class) to become dynamic, thus all the subclasses of the one in which the location was declared become dynamic.

Another way of thinking about it is to consider that since there is no type analysis, then the real type of a location is unknown, forcing the model to take a safe approximation and reflect the new dynamic state to all the subtypes of the declared type.

If A is superclass of B , then $bt\_class(\texttt{B}) \supseteq bt\_class(\texttt{A})$

Where $\supseteq$ means that if A is dynamic (to a certain extent), then B is dynamic.

In summary, the model presents same linking of binding-time across classes as [22], where it is said that "the binding-time of two locations that are used at the same program point (field lookup or method invocation expression) must be equal."..." We use monovariant binding-time, so the classes of such two objects must have the same binding-time." ... "For field access or method invocation, the set of possible types is the complete set of subtypes of the type inferred for the expression. Thus the class that is used as the qualifying type of the self object in a field access or method invocation has the same binding-time as its subclasses."... Thus the binding-time of equivalence classes are linked across a common superclass if a location belonging to that superclass is subject to a field access or to a method invocation. In a way, this is propagating the dynamism.

Binding-time of Instance variables

Instance variables can be of primitive type or objects themselves. On the one hand if they are of base type, then their binding time can be static or dynamic. On the other hand if are objects, their binding time is the binding time of objects, which is a tuple as defined above.

Benefits of the definition

With the definition given above, the binding time of an object is a tuple of binding times. This leads to a better precision of the analysis, since the fact that one instance variable is dynamic does not mean that the whole object is annotated as dynamic and hence that the specializer might still be able to specialize the instance variables that are not dynamic, if there is any.

Let us consider an example to observe the benefits of the definition of object binding time as defined above. In this case, an instance `p` of class `Point` has two instance variables, `x` and `y`, of which bt(`p.x`) = S , but bt(`p.y`) = D.

Had we considered an object dynamic because at least one of its instance variables had dynamic binding-time, then we would have had to consider the object as dynamic, which would be 'understood' by a specializer (see section 2.2.3) as a hint that all the expressions referencing the object had to be residualized. If the analyzed program had managed 10.000 instances of class `Point` of which the abscissa was known, but not the ordinate, then it would have been the same that not knowing neither the abscissa nor the ordinate. The known information about the abscissa would have been useless.

On the contrary, using the approach taken in this work, where the binding-time of the object records the binding-time state of each instance variable, the specializer can still know that the abscissa is known at specialization time, and take appropriate action. That action can thus save the effort of computing the 10.000 abscissas in a residual program. And it is because the binding-time object state is not static nor dynamic, but a *tuple* of binding-time states.

3.5.3 References

Because we have assignments to allow the change of an object, we consider references. This section explains the role of the references in the Constraint-Based Analysis. We use its syntactic counterparts, locations, to follow the syntax-directed analysis.

First we map the notion of reference at an abstract level to the syntactic notion of location.

Then we explain that the binding-time of a location corresponds to the binding-time of its related object. Thus any reference of a certain type has the same binding-time as the equivalence class of the object it references, accordingly to what we stated is section 3.5.2.

From references to locations

References are represented at a syntactic level by locations. Locations were presented in section 3.3.1.

When we instantiate an object, we also create a reference to access it. Both the instance and the reference are then associated to a certain class, that is the defined *type* of the instance and the reference [3].

The type of the reference

Let Abe the type of objects that belong to class `A`, $\&A$ is the type of any reference that points to an object of that type. (We use & to denote the type of a reference.)

Thus any reference that points to objects of type A has type $\&$`A`, which means that all those references have the same type.

In our model all the references that point to objects of the same type have the same binding time, and that binding-time is the binding-time of the object referenced. Let us consider the functions,

$$bt\_reference : reference \rightarrow ReferenceBT$$
where bt, $bt \in BT$ and
$$bt\_object : object \rightarrow ObjectBT$$
where bt, $bt \in BT \Rightarrow bt\_reference(r) = bt\_object(o)$

We say that ReferenceBT = ObjectBT and $dref(r) = o$

$$Rightarrow bt\_reference(r) = bt\_object(o)$$

In figure 3.5 all the references to objects of a given type have the same type: $\&A$. As we have seen in figure 3.3, all the objects of the same class have the same binding-time (objectBT) and the references have the same binding-time than the object they are referencing (referenceBT).

[3] A type is not a class by definition, but in our language, each class corresponds to a type and

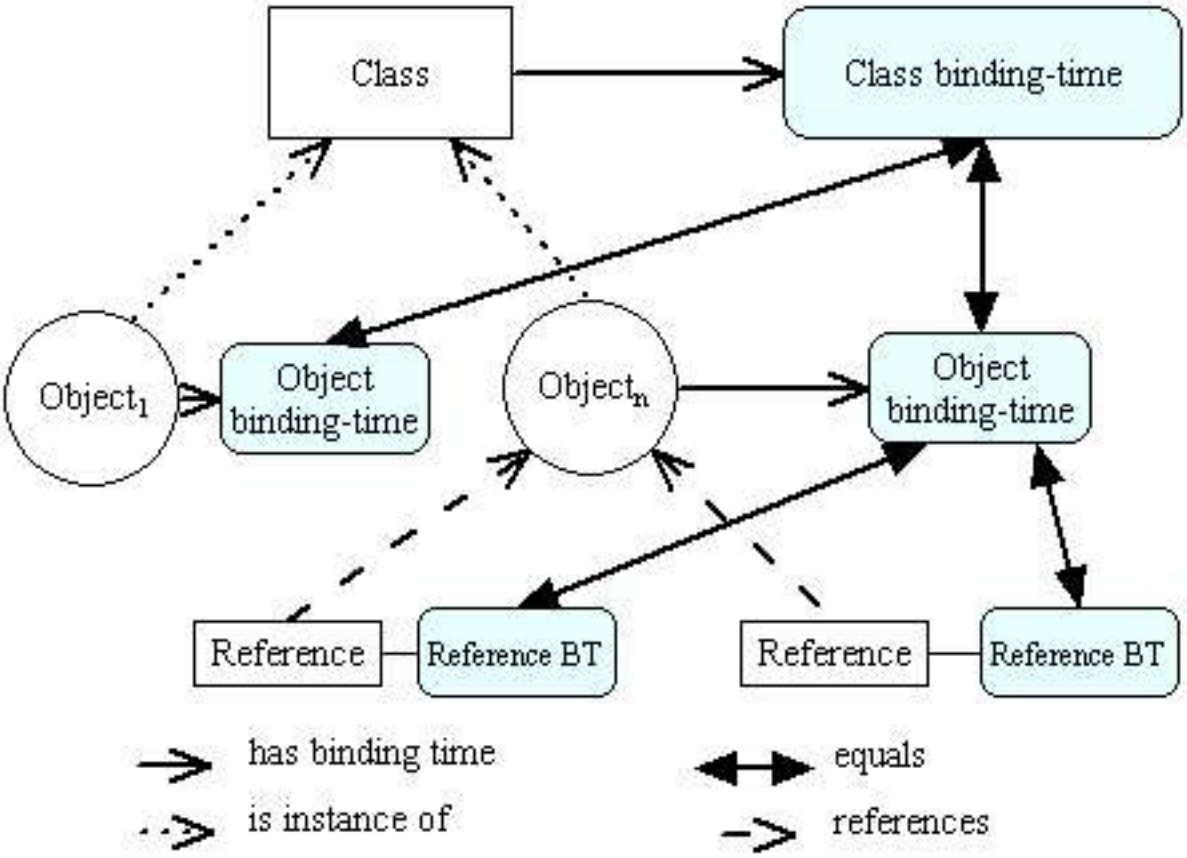


Figure 3.5: Type of the references

When we declare a reference we associate a type type (i.e. A) to this reference. That type is the class of which we declare the object. But later, during the program execution, the reference can reference objects belonging to class `A` or to objects that belong to subclasses of `A`.

It is so because the concrete type of the object that it references can be different at different program points. Thus, references can have a set of concrete types throughout the execution of the program. Figure 3.6 shows an example where the declared type of `aA` is A. The real type of `aA` is one of the types belonging to the set $\{A, B, C\}$ depending on the program point and the actual execution.

Since we can access objects only through references and if we do not forget that the reference is just a pointer to an object that has this or that type we can relax our notation and say that a reference points to a type.

vice-versa.

```
class A {...}
class B extends A {...}
class C extends A {...}
...

A aA = new A();
B aB = new B();
C aC = new C();

if (booleanTest)    aA = aB;
                    else aA = aC;
```

Figure 3.6: Declared and real type

In summary, the relationship between the real type of an object and its class binding time is:

$concrete\_type : reference \rightarrow C$ where r is a reference

and

$C = \{c \mid c$ is a concrete type of the objects that r references in the program$\}$

concrete_type is a function that given a reference r associates it with the set of concrete types of the objects that r references at different program points.

In the example above:

$$concrete\_type(aA) = \{A, B, C\} \Rightarrow bt\_class(A) = bt, bt \in BT \text{ and}$$
$$bt\_class(B) \supseteq bt\_class(A) \text{ and}$$
$$bt\_class(C) \supseteq bt\_class(A)$$

One would be tempted to calculate the binding time of a reference but we don't need to annotate references with their binding time. The binding-time of a reference is the binding-time of the object it references.

Partially Static Objects and the BT of References

The binding time of a reference is the binding time of the object the reference points to. Thus being `p`, `q` references, the binding time of `p` in `p = q` of references should be

the binding time of the class that `q` points to. For instance, in figure 3.7, supposing we have a class `Complex`, what binding time would we assign to the statement labelled (*)?

The answer is (S,D)

```
class Complex { double r,i; }
```

where `r` is static and `i` is dynamic, and the following program segment:

```
    Complex c,d;
    c = new Complex(S,D);
(*) d = c;
    double res = d.r + d.i;
```

Figure 3.7: Assignment of a reference occurring as the left-hand side of an assignment

In JSpec[1], a special binding time *SD*, related to use sensitivity is used. This binding time corresponds to a reference which should exist both at specialization and execution time, meaning that at specialization time, the assignment should be both executed and residualized.

We don't have use sensitivity. Nevertheless, the binding time of a reference is the binding time of the class it references thus the binding time of an assignment of references should be the binding time of the class that `q` points to.

Another question that we can make is if the statement labelled (*) should be residualized in the specialized program or not? The specializer could interpret this binding time (which has both a static and a dynamic component) as an SD binding time, meaning that it is performed both during specialization and residualized in the program. Of course, it would be a monovariant flow-insensitive use sensitivity, but that goes with the overall precision of the BTA. The binding time of a reference of partially static objects occurring at the left-hand side of an assignment is an important issue, but appears only when annotating program constructs and it is left as further research.

In conclusion the binding time of `d` can be said to be "partially static": it is composite and contains some dynamic fragments. This means that the assignment will indeed have to be annotated static and dynamic when dealing with program constructs, giving a weak form of use sensitivity.

3.5.4 Method Calls

We can now express the binding-time associated with a method call. In this section we explain the binding-time of the method-calls. It serves as a foundation for the constraint definition in next chapter.

Methods calls have two aspects to consider:

- the binding times of the arguments and
- the binding time of the returned value

The arguments are ordinary locations. Their binding times determine the binding times of the formal parameters.

There may be various calls to the same method m in the program. Observe that we assume that there is no method overloading, in fact, there are no two methods with the same name.

With a monovariant analysis, two cases have to be considered:

- all the calls $m(arg)$ are such that arg is static, m can be specialized loosing the arguments in the specialization and into as many $m_i()$ methods as different values of arg it had received.

  For instance, if the method $m(arg)$ is called twice, the first one with $arg = 1$ and the second one with $arg = 2$, then it can be specialized as $m_1()$ and $m_2()$ In each one of them, every occurrence of arg inside the method body is replaced by 1 or by 2, depending on the specialization on the method.

- there exists one instance of $m(arg)$ such that arg is dynamic, then the specializer is forced to residualize $m(arg)$ since there is a value of arg used inside the method that will be known only at run time.

  But the fact that one arg in one of the calls to $m(arg)$ is dynamic, does not mean that the arg of the other calls have to become dynamic. In this case, that is, in the case of a call to $m(arg)$ where the actual parameter is static and $m(arg)$ is not, the actual argument has to be lifted if possible (section 2.3.2). Note that it is not forced upwards to be dynamic, as observed in the constraint system (section 4.2.4).

It is also necessary to associate binding times to return values. Unfortunately, there is no syntactic location referring to these return values directly available in the program. This requires to create a specific location `return(`m`)` for each method m occurring in the program.

3.5.5 Example of Calculating Binding Times

Let us illustrate this chapter by computing the binding times of the rectangle `r` in the example of figure 3.9. We assume that `s` and `d` are static and dynamic, respectively.

```
class Point {
    int x;
    int y;
    ...
  }
class rectangle {
    Point p1;
    Point p2;

 ...
      Rectangle r;
      Point p1, p2;
      p1 = new Point(s, d);         // (1)
      p2 = new Point(d, d);         // (2)
      r = new Rectangle(p1, p2);    // (3)

  }
```

Figure 3.8: Example of alculating binding times

Let us note that this is not the way binding times will be computed using CBA.

Without flow sensitivity, a given location cannot have several binding times. It is necessary to take a safe approximation, that is, the highest binding time for each location. Hence,

$$bt(Point.x) = SUD = D$$

and

$$bt(Point.y) = DUD = D$$

As a result,

$$bt(r) = ((D, D), (D, D))$$

3.6 Summary

We have defined the concrete memory model facilitating the reasoning about binding-times of object-oriented languages.

Input: $bt(s) = S$ and
$bt(d) = D$
Line (1): $bt(p1) = bt\_class(Point)$
$= (bt(Point.x), bt(Point.y))$
$= (bt(s), bt(d)) = (S, D)$
that is, $bt(Point.x) = S$ and
$bt(Point.y) = D$
Line (2): $bt(p2) = bt\_class(Point)$
$= (bt(Point.x), bt(Point.y))$
$= (bt(d), bt(d)) = (D, D)$
that is, $bt(Point.x) = D$ and
$bt(Point.y) = D$
Line (3): $bt(r) = (bt(p1), bt(p2))$

Figure 3.9: Calculating the binding-time of a rectangle

We have a concrete and an abstract level. At a concrete level there are values, objects and references, whereas at the abstract level we think about binding-times, object binding-times and reference binding-times. We have established a map between elements of both levels: objects in the concrete level have a certain object binding-time at the abstract level. Values and references in the concrete level have their corresponding elements at the abstract level, that is binding-time and reference binding-time respectively.

At the syntactic level, we have introduced the notion of location and equivalence locations. The task of the analysis is to determine the binding times of locations and equivalence locations.

In this basic model a primitive value can only be either known/static or unknown/dynamic. The binding-time of objects is a tuple of binding-times, one for each instance variable, whereas all the objects of the same class have the same binding-time. The binding-time of the references is the binding-time of the objects they reference.

Chapter 4

Constraints

In chapter 2 we said that binding-time analysis of a subject program computes a division: a classification of each location as static or dynamic.

That division is found by building a set of constraints on the binding-time (3.4.2) of all locations.

This chapter explains actual constraint generation and presents the set of constraints used to calculate that division.

Expressing the constraints is done in a syntax-directed manner, i.e. the program is traversed and a set of constraints is generated for each encountered statement, depending on the grammar.

First, in section 4.1.2 we explain what is a constraint for the binding-time analysis for petitCafé. Then section 4.2 shows the constraints in an abstract way.

Finally, section 4.3 presents a summary.

4.1 BTA by Solving Constraints

Expressing the constraints is done in a syntax-directed manner, i.e. the program is traversed and a set of constraints is generated for each encountered statement, depending on the grammar.

These constraints essentially model location dependencies due in particular to assignments.

In a first step, the constraint solver makes a program tree and traverses it looking for locations and annotating them static or dynamic. The purpose of this is to be able later to analyze the whole set of annotations and deduce, by means of solving a system of inequations, the *safe state* of a location. This means that, in the end, a location is only annotated as static if it was never annotated dynamic previously, when the constraint solver traversed the tree annotating variables. So we calculate the binding time of variables using constraints.

4.1.1 Requirements

Binding-time analysis computes a division: a classification of variables among static and dynamic according to a know input.

It is *safe* if the division it computes is congruent in the sense that each variable is classified as static if it does not depend on a dynamic variable (section 2.2.3).

The analysis computes a congruent monovariant division of a source program. It is intra-procedural, flow-insensitive and context-insensitive.

4.1.2 Constraints in petitCafé

In petitCafé a *constraint* is an inequality on the binding-time of locations.

For instance, if `x` depends on `y`, we want to express the congruence requirement "if y is dynamic, then x must be dynamic too". We wish the ordering $D \supset S$ on the binding time values, and write $bt(\texttt{x}) \supseteq bt(\texttt{y})$. The constraint set can be solved and gives the binding-time for each variable [14].

Partial Order

We define $\subseteq$ to be the order relation on the binding-time BT of the abstract locations.

The binding-time of a location can be of type π and of type Γ (since a location can represent a variable or an object). When the binding-time of a location is of type π, that is S or D, then we are in the π domain, otherwise we are in the Γ domain.

For the domain π, $t \in \pi$. We impose an ordering $D \supset S$ on π:

$$t' \supseteq t \text{ iff } t = S \text{ or } t = t'$$

That is, $\supseteq$ means "is more dynamic than". This ordering extends pointwise to the Γ domain.

More precisely, if $\tau \in \Gamma$, $\tau = (t_1, ..., t_a)$, $\tau' = (t'_1, ..., t'_a)$,

$$(t_1, ..., t_a) \supseteq (t'_1, ..., t'_a) \text{ iff } t_j \supseteq t'_j \text{ for } j \in 1, ..., a$$

$\supseteq$ operator

The inequality uses the operator $\supseteq$. This operator, used to write the constraints, intuitively tells that if the right part of the in-equation is dynamic, it forces the left part to be dynamic. Nor does a change happens if the left hand side was dynamic nor if the right hand was static. In this case, if the left hand had a static annotation, it still can be static, but if it was 'pushed up' by an annotation that made it be dynamic, then in the end its final annotation will be dynamic.

To be linked to the domains, $\supseteq$ is actually an order relationship.

4.2 Constraint Generation

In this section we explain actual constraint generation[5]. Rather that giving an implementation view of it, which will be given in section 5.1.3, we start explaining them from the specifications. The constraints relate to the "abstract" locations according to the equivalence classes (that were introduced in section 3.5.2) rather than to syntactic locations.

This has to be done since there is not a one-to-one mapping between the "syntactic" locations and the "abstract" locations. That is, `p.x` and `q.x` are two different syntactic locations but if `p` and `q` have the same declared class A, it could be that they refer to the same "abstract" location that one could denote A.

There are only constraints on statements. Then we write the constraints for expressions, and finally the constraint for a method invocation and new.

We present below first the constraints related to intra-procedural analysis, then to inter-procedural analysis, and finally to class hierarchies.

4.2.1 Variable assignment

The assignment constraint is as follows:

Let us assume:

$l \in Loc$

$e \in Expr$ ($Expr$ is the syntactic domain of expressions)

$v1, v2 \in variable$

$bop \in BinayOperator$

$uop \in UnaryOperator$

$c \in literal$

$v.f \in field$

$t \in Type$

Then for an assignment $l = e$, the constraint is $bt(l) \supseteq btexp(e)$ where $btexp$ (of type $Expr \rightarrow BT$) is defined as follows:

$$
\begin{aligned}
btexp(\texttt{v1}bopv2) &= bt(v1) \cup bt(v2) \\
btexp(uopv) &= bt(v) \\
btexp(v) &= bt(v) \\
btexp(c) &= S \\
btexp(v.f) &= bt(v.f) \\
btexp((t)v) &= bt(v) \\
btexp(\_.m(\_)) &= bt(return(m)) \\
btexp(new\ c(\_)) &= bt\_class(c)
\end{aligned}
$$

where $return : Method \rightarrow Loc$ receives a method m and returns the return location associated to m.

Observe that even though a variable is not directly assigned a dynamic expression, there are cases where it should become dynamic anyway because of its *context*. For instance if the assignment takes place in a conditional branch and the test of the conditional is dynamic. This is done by generating a second constraint on the variable, as we will see in 4.2.2.

New

The binding-time of an instance_creation is as follows:

Let us assume:

$$l \in location$$
$$new\ class\_name(args) \in class\_instance\_creation\_exp$$

Then for an instance creation expression $l = new\ class\_name()$, the constraint is

$$bt(l) \supseteq bt\_class(class\_name)$$

where $bt\_class : class \rightarrow BT$ receives a class and returns the binding-time of that class.

The binding time of the expression is the binding time of the self object, that is the binding time of the $new\ Class(fd_1(deref(r)), \ldots, fd_n(deref(r)))$.

$statement(m, i), (r = new\ Class()) \in i$
and r is reference
and $Class$ is a class
and fd_i are instance variables
$\Rightarrow bt(r) \supseteq (bt(fd_1(deref(r))), \ldots, bt(fd_n(deref(r))))$

The binding time of the self object has the binding time type BT and is the n-tuple of binding times of instance variables.

In case of a new expression, additional constraints must be generated to deal with the arguments of the constructor. Basically, for each field $c.f$ of class c corresponding to an argument arg_i of the constructor, the following constraint has to be generated:

$$bt(c.f) \supseteq bt(arg_i)$$

Field assignment

Field assignment is an assignment of the form $v.f = e$, where e is an expression. The binding time of the field depends on the binding time of e as any assignment.

As in the case of variable assignment, the binding time of a field assignment is also decided by the places where the assignment occurs, for example if the assignment happens in one of the branches of a conditional, then the binding time of the conditional test decides also the binding time of the variable. We take into account this in

Conditioning Variable Assignments in section 4.2.2. In particular, the binding time of the field is dynamic if the binding time of the expression it is assigned is dynamic.

An assignment where the left part is a field can change the binding-time of the declared class of the object pointed to by v. This is because references have the same binding-time as objects and objects have the binding-time of their class. Consequently it is necessary to add a constraint to express the relationship between the binding-time of a class and the binding-time of its objects and between the binding-time of an object and the binding-time of its instance variables.

In this case, and for the sake of simplicity, we chose to add a constraint that relates the binding-time of the instance variables to the binding-time of the class of the object, that is the declared class of the reference that is pointing to the object.

Let us assume:

$$v.f\_i \in field$$

Then for any instance variable $v.f\_i$ of an object, the constraint is $class\_bt(declared\_class(v)) \supseteq bt(v.f\_i)$

Where $declared\_class : Loc \rightarrow Class$ receives a Location and

returns the Class associated to m,

and

$bt\_class : Class \rightarrow BT$ receives a class and

maps it to the binding-time of that class

Field Access

In terms of constraints the field binding-time is:

let us assume:

$v.f \in field$

$s \in \pi$

Then for a field access $v.f$, the constraint is:

$bt(v.f)$	$\supseteq bt_class(declared_class(v.f))$if $v.f$ is an object
$bt(v.f)$	$\supseteq s$ such that $s = S$ or $s = D$, if $v.f$ is of basic type

Where $declared\_class : Loc \rightarrow Class$ receives a Location y returns the Class associated to m, and $bt\_class : Class \rightarrow BT$ receives a class and maps it to the binding-time of that class.

If a field i is of primitive type, then its binding-time is S or D. If i is not a primitive type, then the binding time of i is a BT that can be more complex (a tuple for instance).
The binding time of the instance variable i of an object depends on what it *is* the instance variable, because the binding time has a different types depending on if the instance variable is an object or a value of a primitive type. If it is an object, then the field access will have a binding time of type Γ, since it is the binding time of the object that is stored in the instance variable. On the contrary, if it is a value of a base type what is in the instance variable i of object $g(r)$, then the binding time will be of type π whose values can be S or D.
As we saw in section 3.5.2, the binding time of an object is a tuple formed of binding times. An element i of this tuple is the union of the binding times of instance variable i, if it exists, on all the objects that belong to classes that object o have as real type throughout the analyzed program.
Considering this, we can say that:

$statement(m, i)$,x.field_i$\in i$ where $x.field\_i$ is actually a reference to an object or primitive type that is a field of the object pointed to by r, so $x.field\_i$ is $fd\_i(x)$

- $fd_i(deref(r))$ is the i_{th} instance variable of the object that is being pointed to by r
- r is a reference
- $deref(r)$ is the object to which the instance variables belong.

Here there is a statement i in a method m, the statement i has a Field access

1. If the instance variable fd_i of object $deref(r)$ is an object, let's say $deref(r)$ $\Rightarrow bt(fd_i(deref(r))) = \bigcup(bt(classOf_j(object)))\forall$ class j to which the object can belong.

2. the instance variable fd_i of object $deref(r)$ is a value of a base type $\Rightarrow fd_i(deref(r)) = v$ where v is a value $\Rightarrow bt(fd_i(deref(r))) = bt(v)$[1]

It happens that objects are referenced to by references, in this case our object o is referenced by the reference r. We can here use the fd and $deref$ functions defined in section 3.3.1 since a field access is nothing more than a reference to an object that is de-referenced and pinpointed a certain object's instance variable. In terms of our functions, that is to say that to access instance variable i of an object pointed to by r, we just apply the composition of $deref$ with fd, for instance $fd_i(deref(r))$.

4.2.2 Conditional

The binding time of a conditional depends on the binding time of its *test condition*[2].

Let us assume:

$$block1, block2 \in block$$
$$test \in variable$$

Then for a conditional `If`$testblock1block2$, the constraint is $bt(\texttt{If}) \supseteq bt(test)$

In another format, we write the same: $statement(m, i), conditional_i = i$, $bt(conditional_i) \supseteq bt(test(i))$

We say that in a method m there is a statement $conditional_i$ whose binding time depends on the binding time of the test of the conditional.

Here $conditional_i = i$ express that the sentence i is a conditional.

Conditional Nesting

But two ore more ifs can be nested one into another. In this case the binding time of the inner `if` depends on the binding time of the outer `if`, that is, of the `if` that

[1] v is the actual argument sent to the expression

[2] The test condition is the conditional condition that decides, depending on its value, if either the true branch or the false branch is going to be executed.

has the branch in which the inner `if` is. Later we will call this outer `if` the *enclosing* conditional of the first one. To reflect the fact that if the enclosing if is dynamic, the inner one must also be dynamic, there exists the following constraint:

Let us assume:

$$If_1\ test_1\ block_1\ block_2 \in stmt$$
$$If_2\ test_2\ block_3\ block_4 \in stmt$$
$$test_1,\ test_2 \in variable$$
$$block_1, block_2, block_3, block_4 \in block$$

Where If_2 is nested into If_1. That is,

$$If_2 test_2\ block_3\ block_4 \in block_1 \text{ or}$$
$$If_2 test_2\ block_3\ block_4 \in block_2$$

Then $bt(If_2) \supseteq bt(test_1)$

Conditional Assignments

As we said before, the assignment depends on the conditional. Given an assignment of the form $x = expression$, where x is a variable name and *expression* a valid *variable_asignment_expression*, if that assignment is a statement of one of the branches of an conditional, then *the binding time of x depends directly on the binding time of the test of the conditional*. This is because if the test is dynamic, then x is dynamic, and the specializer doesn't know that a branch will be eliminated at specialization time. So the x should be regarded as dynamic if the conditional's test is dynamic. This raises the following constraint:

Let us assume:

$$l \in Loc$$
$$e \in variable_assignment_exp$$
$$If_1\ test_1\ block_1\ block_2 \in stmt$$
$$l = e \in block_1 \text{ or } l = e \in block_2$$

Then for the assignment $l = e$, the constraint is

$$bt(l) \supseteq btexp(e)$$

if x is the left hand side of an assignment that appears in one of the branches of the if_i then generates the constraint $bt(x) \supseteq bt(if_1)$.

4.2.3 While

The handling of `while` statements does not present any special difficulty once conditional statements are handled.

4.2.4 Method Invocation

The analysis we make is monovariant (see section 2.3.3), which means that for each method invocation in the future only one specialization will be produced. In terms of binding-time analysis, this means that the binding time of each formal parameter is the uppermost state of the binding time of the corresponding actual parameter in each call. So if in one method call the actual parameter arg_i is static, but in another call to the same method the actual parameter arg_i is dynamic, then the binding-time analysis should inform that the formal parameter $form_i$ corresponding to the actual parameter arg_i should be considered dynamic. It is the same case as for variable assignment, if we think that the actual parameter is assigned to the formal parameter in each method call.

As said in 3.5.4, we assume that there is no method overloading, in fact, there are no two methods with the same name. Taking this into account, the constraint generation for method calls has two aspects: the constraint generation for the call and constraint generation for returning from the call (if the method does not return void).

The following sections show the different constraints that apply to method invocation according to the aforementioned two aspects of a method call.

Constraint generation for the call

The binding time of the formal arguments respect to the binding time of the actual arguments follows the same principles as the assignment. We can see that the constraints for this are the same as for the assignment $form_i = arg_i$.

Let us assume:

$arg_i, form_i \in location$

$arg_1, ..., arg_n \in args \forall i \in [1..n]$

$form_1, ..., form_n \in args \forall i \in [1..n]$

$form_1, ..., form_n \in method\_def \forall i \in [1..n]$

$m \in method\_name$

$x \in location$

$o \in location$

Then for an method call $x = o.m(arg_1, ..., arg_n)$, the constraint is

$$bt(form_i) \supseteq bt(arg_i)$$

Or in another way: for a method mc whose method definition is $mc(fom_i 1, ..., form_n)$, the method call of mc on the object referenced by o,
$o.mc(form_1, ..., form_n)$

Given the call $(o.mc(form_1, ..., form_n)$
$statement(m, i), (x = o.mc(form_1, ..., form_n) \in i \wedge form_j$ is the formal argument corresponding to the actual argument $arg_j \ \forall j \in [1, n]$ then generate the constraint $bt(form_j) \supseteq bt(arg_j)$

Let us assume:

$arg_1, ..., arg_n \in args \ \forall i \in [1, n]$
$m \in method\_name$
$form_1, ..., form_n \in location \ \forall i \in [1, n]$
$o.m(args) \in method\_invocation$
$x \in location$
$o \in location$

Then for an method call $x = o.m(arg_1, ..., arg_n)$, the constraint is

$bt(x.m(arg_1, ..., arg_n)) \supseteq bt(return(m))$
and also
$bt(x) \supseteq bt(return(m))$

4.2.5 Class Hierarchy

The binding-time of the class A depends on the binding-time of its superclass.

Let us assume:

$A, B \in class\_name$
A is superclass of B
$vA.f, vB.f \in field$
$vA, vB \in variable$
$ffield\_name$ of the declared class of vA

Then for any field f that B inherits from A there is the constraint:

$bt(vB.f) \supseteq bt(vA.f)$

4.3 Summary

We have presented actual constraint generation. We have defined the expression of the constraints with the variants in terms of the arguments. These expressions can be used to calculate the division among the values static and dynamic that had been defined in chapter 3.

The set of constraints is the basis of our binding-time analysis implementation (design and prototype) and determines the precision of the binding-time analysis.

These constraints essentially model location dependencies due, in particular, to assignments. Once the constraints are established, a solution in terms of static/dynamic states of the locations is found, so that the constraints are met. We will see in the next chapter how solve the problem of finding which locations can be static and which should dynamic so that all the constraints are satisfied.

Chapter 5

Constraint Solving

In chapter 4 we presented the constraints for petitCafé in an abstract way. In this section we show how to turn those constraints into REQS lingo. In particular how to turn inequations into equations.

First, section 5.1 presents the equation solver REQS and it shows how to turn the basic constraints from abstract expressions into an equation system that REQS can solve.

Then, section 5.2 presents petitAnalyzer, a tool to traverse the tree of the source program extracting relevant information.

Afterwards, section 5.3 explains the process making the BTA of a petitCafé program and give examples.

Finally, section 5.4 presents a summary.

5.1 The Equation Solver

So far we have the set of constraints and programs that we would like to analyze. The set of constraints was defined in section 4.2 and the programs are the ones that we can write following the grammar defined in section 3.2.1.

As said before, the constraints, generated for a concrete program, define an inequation system that reflects the relationship between the different program parts. In particular, they define the relationship with respect to the binding times. For instance, the assignment `x = y`, implies that if `y` is dynamic, then `x` should be annotated as dynamic. The variable assignment in section 4.2.1 enforces this. But this constraint system has to be solved, thus we use an equation solver to do it.

We use REQS ("résolution de système d'équations récursives")[19] to solve equations. REQS is a program that solves recursive-equation systems produced in the context of static analysis of programs. It is independent of the application domain

because it solves a system of equations on a domain of properties. That system should be the result of an analysis of the specific domain. The solution to this system represents the information looked for.

Now we have to build a bridge between the inequation constraint system and the equation system written in REQS lingo. Consequently we show the how to transform the abstract constraints so that can understand them. Then we write a program that traverses the tree of the program we want to analyze, detects the parts that we look for (assignments, conditionals and method calls) and writes the relationship information between those parts. Using this information we produce REQS equations.

5.1.1 REQS Basics

REQS is a program solving recursive-equation systems. It offers a number of predefined lattices (i.e boolean, T2) and allows using basic operators (i.e equality operator). Using them we build the equation systems that REQS solve.

Lattice

In particular REQS offers the predefined lattices *'T2'* [1] and *boolean*. We use T2, a lattice of two elements, top and bottom, where $\bot \subseteq \top$

We map the binding-time values S and D to $\bot$ and $\top$ respectively. $\bot$ and $\top$ are represented by the T2 predefined constants 'BOTTOM and 'TOP respectively. For instance,

```
y = 'BOTTOM
```

indicates that `y` is $\bot$, that maps to S (static), one of the values of the domain we are interested in. On the contrary,

```
x = 'TOP
```

indicates that variable `x` is D (dynamic). Thus, to write the equation system in REQS lingo, we can use as many variables as we need, but in the domain of values used by REQS, there are only two values: $\bot$ and $\top$.

Operators equality, INTER and UNION

REQS does not have the operator for inequations, applied to the lattice we use, T2, but it does provide the equality, INTER and UNION operators.

- equality between two operands, (i.e. $x = y$) means that the left operand has the same value ('TOP or 'BOTTOM) as the right operand.
- UNION returns the upper binding-time out of two elements of the lattice T2.

[1] T for treillis, lattice in French

- INTER returns the lower binding-time out of two elements of the lattice T2.

Let us assume the domain $\pi_R = \{\bot, \top\}$
The UNION operator is defined $\cup : (\pi_R, \pi_R) \rightarrow \pi_R$
where:

$$\cup(\top, \top) = \top \tag{5.1}$$
$$\cup(\bot, \top) = \top \tag{5.2}$$
$$\cup(\top, \bot) = \top \tag{5.3}$$
$$\cup(\bot, \bot) = \bot \tag{5.4}$$

And for the same domain $\pi_R = \{\bot, \top\}$
The INTER operator is defined $\cap : (\pi_R, \pi_R) \rightarrow \pi_R$
where:

$$\cap(\top, \top) = \top \tag{5.5}$$
$$\cap(\bot, \top) = \bot \tag{5.6}$$
$$\cap(\top, \bot) = \bot \tag{5.7}$$
$$\cap(\bot, \bot) = \bot \tag{5.8}$$

For the basic case of inequation used to write the constraints:

$$bt(l1) \supseteq bt(l2) \tag{5.9}$$

the equation written in REQS lingo is:

$$l1 = \cup(\cap\ l2\ 'TOP)\ l1$$

The proof takes into account all cases according to the values of $l1$ and $l2$:

In order to give the proof we will use an auxiliary function, $new\_value : variable \rightarrow \pi_r$ that takes a REQS variable and returns the value result of applying the REQS equation aforementioned.

We will use $\top$ and $\bot$ instead of 'TOP and 'BOTTOM for the proof to be consistent with the definition of lattice T2. Nevertheless, in the examples we will use 'TOP and 'BOTTOM, the constants defined for T2, since that is the way to write the equations defined in REQS.

The four cases according to the combinations of $\bot$ and $\top$ that $l1$ and $l2$ can present are:

Case 1: $l1 = \top$, $l2 = \top$

$$
\begin{array}{lll}
new\_value(l1) & = \cup\ (\cap\ l2\ \top)\ l1 & \\
& = \cup\ (\cap\ \top\top)\ \top & \text{replacing } l1 \text{ and } l2 \text{ for their values} \\
& = \cup\ \top\ \top & \text{for 5.5} \\
& = \top & \text{for 5.1}
\end{array}
$$

Result of applying the equation is $new\_value(l1) = \top$
Case 2: $l1 = \bot$, $l2 = \top$

$$
\begin{array}{lll}
new\_value(l1) & = \cup\ (\cap\ l2\ \top)\ l1 & \\
& = \cup\ (\cap\ \top\top)\ \bot & \text{replacing } l1 \text{ and } l2 \text{ for their values} \\
& = \cup\ \top\ \bot & \text{for 5.5} \\
& = \top & \text{for 5.3}
\end{array}
$$

Result of applying the equation is $new\_value(l1) = \top$
Case 3: $l1 = \top$, $l2 = \bot$

$$
\begin{array}{lll}
new\_value(l1) & = \cup\ (\cap\ l2\ \top)\ l1 & \\
& = \cup\ (\cap\ \bot\top)\ \top & \text{replacing } l1 \text{ and } l2 \text{ for their values} \\
& = \cup\ \bot\ \top & \text{for 5.6} \\
& = \top & \text{for 5.2}
\end{array}
$$

Result of applying the equation is $new\_value(l1) = \top$
Case 4: $l1 = \bot$, $l2 = \bot$

$$
\begin{array}{lll}
new\_value(l1) & = \cup\ (\cap\ l2\ \top)\ l1 & \\
& = \cup\ (\cap\ \bot\top)\ \bot & \text{replacing } l1 \text{ and } l2 \text{ for their values} \\
& = \cup\ \bot\ \bot & \text{for 5.6} \\
& = \bot & \text{for 5.4}
\end{array}
$$

Result of applying the equation is $new\_value(l1) = \bot$

When it happens that $new\_value(l1) = \top$ and $l1$ is $\bot$, REQS assigns the value $\top$ to $l1$ in an attempt to solve the equation system. This is exactly what interests us, since it means that it is solving the equation system taking into account the constraint 5.9. In particular, it is the case when $l1 = \bot$ and $l2 = \top$.

5.1.2 Turning Inequations into Equations

For REQS to answer the inequation system given for the constraints applied to the source program, we need to map the operators present in our constraint system into the operators provided by REQS.

REQS solves an equation system, but we can translate the $\supseteq$ operator we used to write our previously defined constraints into equations that REQS accepts.

Example of REQS solving $bt(l1) \supseteq bt(l2)$

We show through an example how REQS behaves when it finds the following specification.

```
type l1, l2, aux T2
l2 = 'TOP
l1 = 'BOTTOM
aux = (INTER l2 'TOP)
l1 = (UNION aux l1)
```

This code expresses in REQS lingo the basic abstract constraint $bt(l1) \supseteq bt(l2)$, in this case with $l1$ static and $l2$ dynamic. Note that $l1$ and $l2$ could be variable of primitive type.

Figure 5.1 shows the graph shown by REQS when running the example.

As expected, REQS answers that $l1$ should be annotated as dynamic. It is right because $l2$ was dynamic ('TOP in REQS lingo)[2] and $l1$ as well as $l2$ were bound by the constraint $bt(l1) \supseteq bt(l2)$. Figure 5.2 shows the answer of REQS[3].

5.1.3 Constraints in REQS

The answer of the petitAnalyzer needs to be translated to a system of equations that REQS can analyze. This section show how to translate the constraint system in terms of the 'abstract' inequations into REQS.

[2] 'BOTTOM is static in REQS lingo.

[3] "T" for 'TOP, had it answered 'BOTTOM, it had shown a "B".

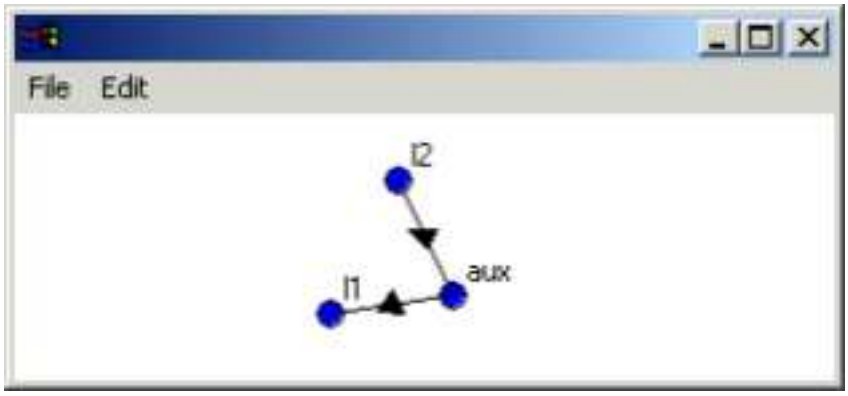


Figure 5.1: Graph shown by REQS

Figure 5.2: REQS answer solving $bt(l1) \supseteq bt(l2)$

To show the transition we take the program part (i.e. assignment) and the tuples that are produced for the first analyzer and indicate how to write this piece of the system in code understandable by REQS.

Literal

When the source program analyzer finds one literal, it records that it has found a literal by mapping it to the predefined tuple element `value`.

Example:

```
Int y;
y = 2;
```

This code is translated into the abstract tuple (see section 5.2):

$$(\texttt{assignment}, \texttt{y}, \texttt{value})$$

Any `value` element of a tuple is mapped to the 'BOTTOM value of the domain of lattice T2 in REQS. In fact, we can translate that into REQS in the following way:

```
type value, y T2
value = 'BOTTOM
y = value
```

Figure 5.3 shows the graph that REQS presents.

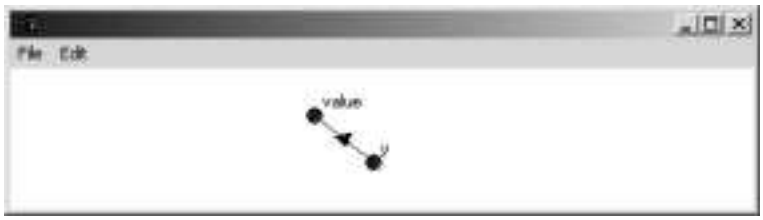


Figure 5.3: Graph presented by REQS with constants

Finally REQS indicates that `y` is static. Its answer is:

`value`: B

`y`: B

Assignment

When the analyzer finds an assignment i.e. $l\_1 = l\_2$ it produces the tuple:

`(assignment, l_1,l_2)`

For example, assume that `expression` is static in

```
Int x;
x = expression;
```

The source program analyzer associates the text above with the tuple

`(assignment, x, expression)`

And finally generates the REQS code

```
type x, expression T2
expression = 'BOTTOM
x = expression
```

To this system, REQS answers that `x` is "B", for 'BOTTOM, which translated to the domain of our model, means that `x` is static.

Had `expression` been dynamic, REQS code would have been:

```
type x, expression T2
expression = 'TOP
x = expression
```

And REQS would have answered that `x` is "T", for 'TOP, meaning that `x` is dynamic.

Sometimes it may happen that a variable is affected by constraints several times. For instance if a variable appears in the left-hand side of more than one assignment statement as in this example:

```
x = 1;
x = dynamic_expression;
```

In this case, the assignment constraint should be applied twice. This corresponds to the set of tuples

```
(assignment, x, value)
```

```
(assignment, x, dynamic_expression)
```

Which, translated into REQS equations, gives (`de` stands for a dynamic variable, it is a short name for `dynamic_expression`):

```
type x,x1,x2,value,de T2
de = 'TOP
value = 'BOTTOM
x1 = value
x2 = de
x = (UNION x1 x2)
```

The UNION operator, as expected, makes `x` be 'TOP. Figure 5.4 shows the graph produced by REQS. It shows the dependency of `x` with respect to `value` and `de`.

Finally the answer of REQS is:

```
value: B
de: T
x2: T
x: T
x1: B
```

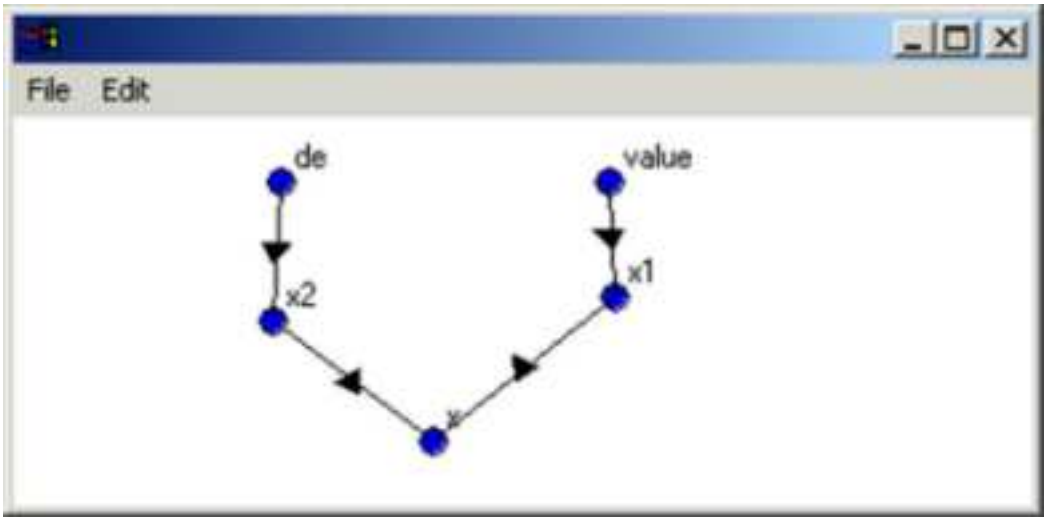


Figure 5.4: TOP prevailing over BOTTOM

Conditional

When there is a conditional in the code, there are several constraints that can apply. They are explained in 4.2.2. There is the *conditional constraint*, which derives in the application of a constraint on the *expression* to produce the binding time of the conditional test condition. This expression can also lead to add several constraints, depending on the expression.
Here it is an example of a constrained conditional.

```
t= x == 0;
if (t)   a = 1;     //t is the condition test
         else a = 2;
```

The output of the petitAnalyzer is:

- $(if, if_1, testif_1)$, conditional constraint. It is to enforce that the `if` depends on the test
- $(testif, testif_1, expression\_test)$, $testif_i \supseteq expression$ constraint. It is to say that $testif_1$ depends on the expression of the test
- expression constraints. There are two:

  `(expression, expression_test, x)`

  This is to say that the expression called *expression_test* depends on `x`, and

`(expression, expression_test, 0)`

idem, but it depends on 0, a value

- $(assignment, a, if_i)$, conditional assignment constraint. To say that if a is in the left side of an assignment in one of the branches of if_i, then the assignment depends of the binding time of an `if`, in particular, on the binding time of if_i.

Here is an example of a system written in such a way that REQS can understand and then solve it.

```
Type i, test, x, zero, a, one, two, a1 T2

i = (UNION (INTER test 'TOP) i1) //i depends on the test
test = (UNION (INTER x 'TOP) test1)
test = (UNION (INTER zero 'TOP) test1)
one = 'BOTTOM
two = 'BOTTOM
a1 = one
a1 = two
a = (UNION (INTER i 'TOP) a1) //a depends on the if
```

5.1.4 Example

The following example shows how the analysis is performed.

```
public class Example {
    int test;
    /** Creates a new instance of Example */
    public Example() {
    }
    public void method () {
        test=0;
    }
}
```

Given as example the Example program, the first analyzer shows the tree that contains an assignment and gives as a result: `(assignment, test, value)`

When we translate this into REQS constraints using the Variable Assignment pattern, we obtain code like the following one:

```
type x,value T2
value = 'BOTTOM
x = value
```

Which once solved by REQS gives the solution:

```
x: B
value: B
```

5.2 petitAnalyzer

We call *petitAnalyzer* the program that we made and that traverses the abstract syntax tree (AST) of the program to be analyzed and exposes the relevant relationships between the parts. It receives as input a program to analyze, traverses its AST looking for assignments, conditionals and method calls and returns a set of tuples. Each tuple indicates the statement found, for instance `assignment`, the name of the part whose binding time is affected by the statement, and the name of the part whose binding time affects the previous element of the tuple.

For example given the assignment `x = y`, the analyzer tells that it found an assignment, on the variable `x` and that the binding time that affects `x` binding time is the binding time of `y`.

Figure 5.5 shows petitAnalyzer analyzing a petitCafé program. We can observe that in order to express that it founds the assignment `x = y` in the source program, petitAnalyzer shows the tuple `(assignment, x, y)` as it shows the upper right text area.

It is because there is a constraint on `x`, variable assignment, that says that there is a relationship between the binding-time of `x` and the binding-time of `y`.

The relationship is that if `y` is dynamic, then `x` is forced *upward* to be dynamic. We say upward because, as said before, dynamic prevails over static. Thus the constraint variable assignment makes that the final result is variable `x` dynamic.

Another example is `y = 2` In this case the binding time of `2` is static because it is a constant, so it is bound by the constant constraint. The analyzer should simply inform that an assignment has been found on the `y` and that `y` is assigned some value that is static. `(assignment, y, value)`

`(assignment, y, value)`

Figure 5.6 shows the AST displayer, ASTDisplayer, showing the parsed tree of our program.

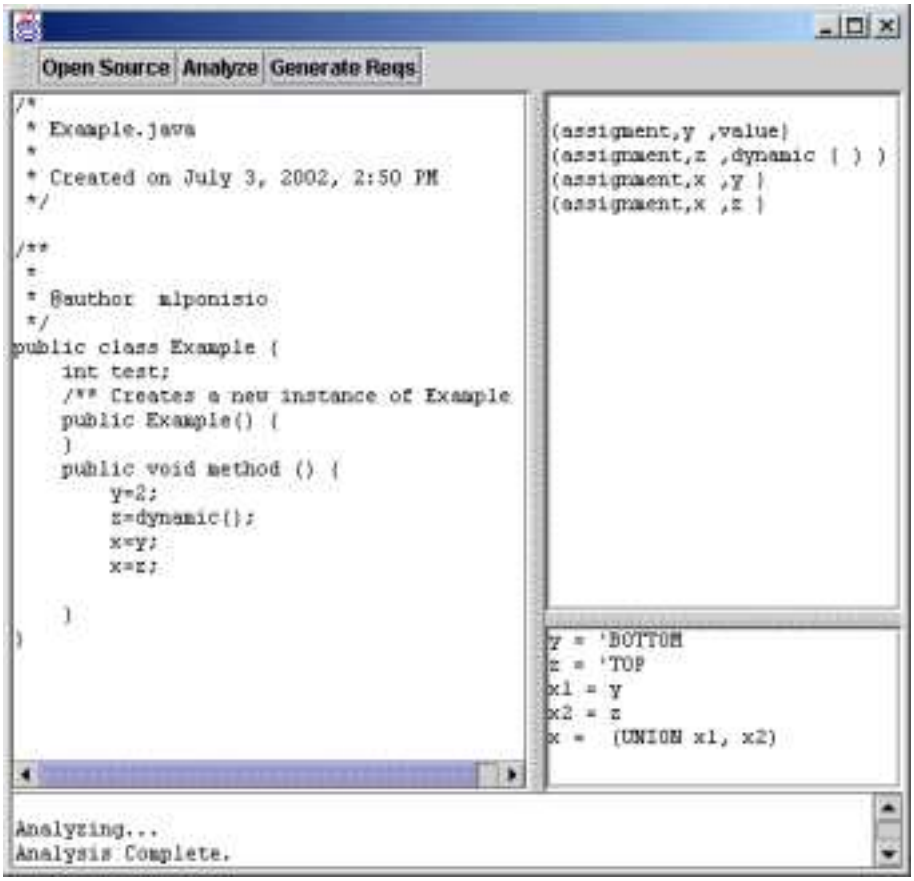


Figure 5.5: petitAnalyzer

5.2.1 Showing the Annotated Program

Once that we have the information about how to annotate each program part, we show the source petitCafé program colored. We show it with HighLight, a program that we made and that receives a petitCafé program and the information about the binding-time of the program parts and produces as output the program with the dynamic parts in red and the static parts in blue. So far it takes into account variables of primitive types with assignments and conditionals. Figure 5.7 shows HighLight displaying an annotated program.

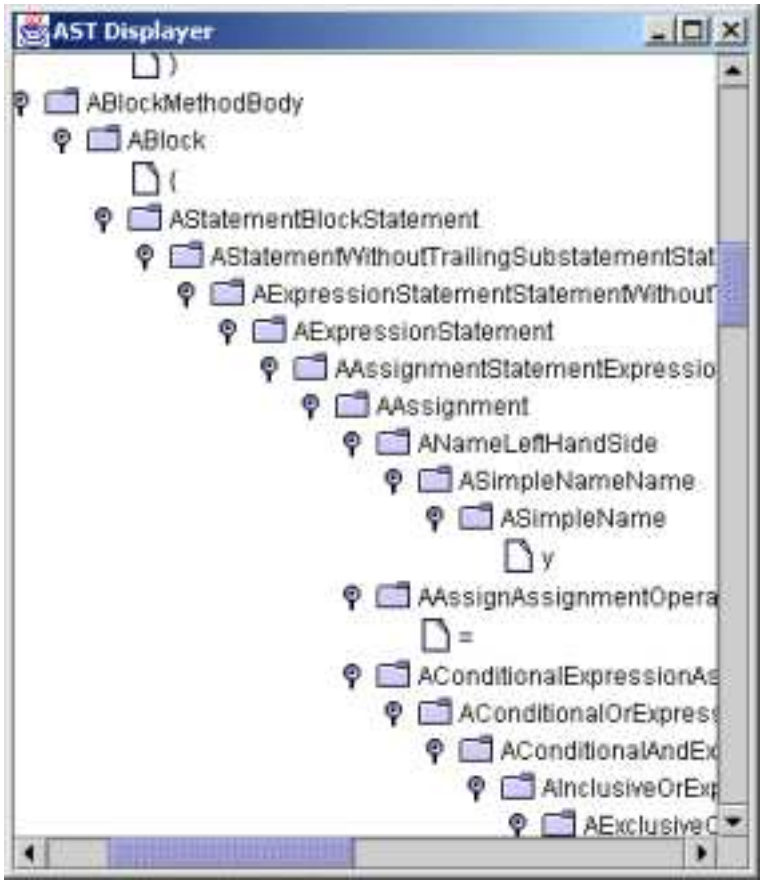


Figure 5.6: ASTDisplayer showing part of the parse tree

5.3 An overview of BTA of petitCafé programs

We perform BTA on petitCafé (a subset of Java) programs in order to find which program elements are static (known at specialization time) and which dynamic (unknown at specialization time). We rely on constraint-based program analysis [16]. Thus, in a first step we define the syntax for petitCafé and define the model on which we will work (i.e. how to express the binding-time of an object, locations, abstract and syntactic level).

Then we write a set of abstract constraints that defines an inequation system based on the model and on the characteristics of object-oriented languages, in particular Java. The goal is to have an inequation system that reflects the relationship between elements of the program in object-oriented languages. So that if there exists a relationship between two elements and the binding-time of one element affects the binding-time of another, that relationship is written as a constraint in the inequation

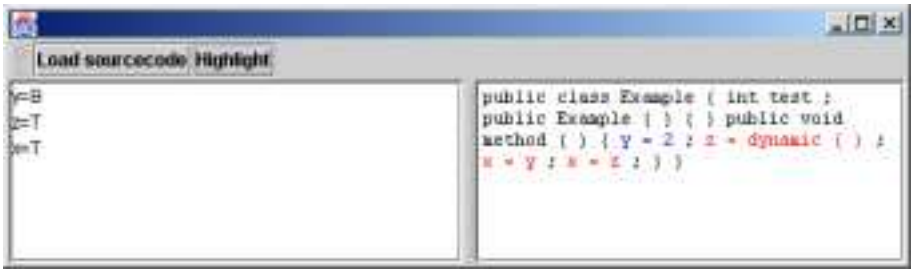


Figure 5.7: HighLight showing the annotated program according to the BTA performed

system.

Consequently, the solution of that inequation system shows which program parts of the source program are static and which dynamic.

In particular we use the equation solver named REQS to find this solution. Besides we wrote and used a program, petitAnalyzer, to look for relevant information in the source program (e.g. assignments). petitAnalyzer looks for assignments and conditionals in the source program and using the abstract constraint inequation system aforementioned (it is embedded in its logic) answers an equivalent equation system written in REQS lingo.

The overall process to find the BTA of a petitCafé program is, thus, the following:

1. Traverse the source program with petitAnalyzer. This returns the equation system written in REQS lingo.

2. Make REQS solve the system produced by petitAnalyzer.

Figure 5.8 shows this process.

5.4 Summary

We have presented an equation solver and explained how it can be used to solve inequation systems built with the constraints of chapter 4. We have shown how to turn the inequations into equations that can be understood by the equation solver and given examples as well.

Afterwards, we have presented a program that traverses a source program (the one to analyze) looking for assignments and conditionals on variables of basic type, producing the code for the equation solver.

Finally we have given an overview of the process.

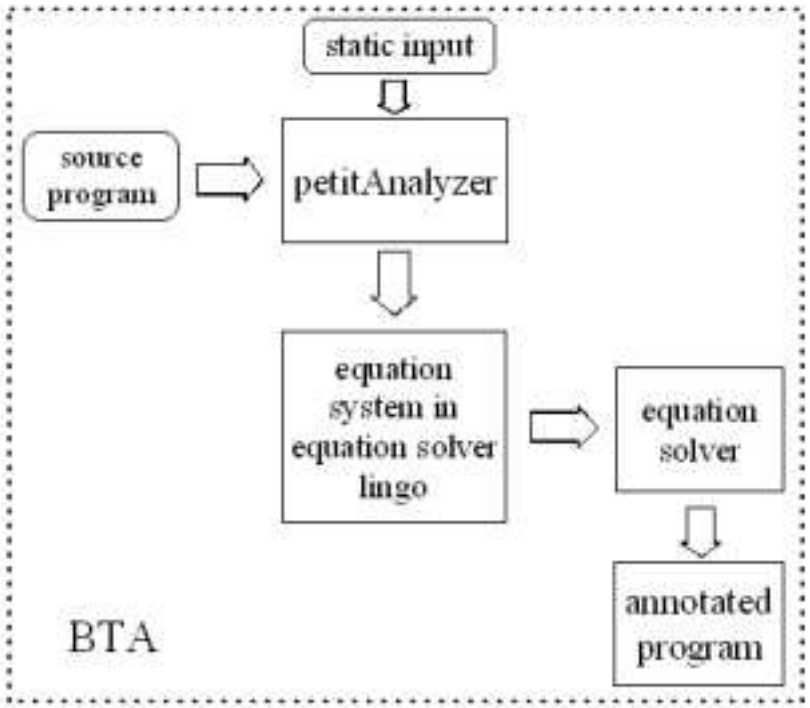


Figure 5.8: Binding-time analysis complete process

Chapter 6

Conclusion

In this chapter we recapitulate the motivations and goals of this work and what has been accomplished. Afterwards, we give our insight into what could be future work based on our approach and a final conclusion.

6.1 Motivation and Goal

Object-oriented languages allow the easy construction of general software components, but this pays the toll of inefficiency. Binding-time analysis is a fundamental technique for partial evaluation and partial evaluation allows to map generic software into specific implementations dedicated to specific purpose.

Our goal was to design model that helped concrete reasoning about binding-time analysis for object-oriented languages. That model had to be simple and flexible, capturing the basic characteristics of BTA for object-oriented languages.

The design would lay the foundations of correct and precise BTAs for Java, starting from the definitions of of the semantics of Java and relying on constraint-based program analysis.

6.2 Summary

To achieve our goal, we defined petitCafé, a subset of Java with the essential characteristics specific of object-oriented languages.

Moreover, we modelled a simple BTA for petitCafé using constraints and implemented a small prototype taking into account some of the issues included in the model. In order to do this, we studied the theoretical framework for program analysis provided by constraint-based analysis.

Then, we defined the constraints necessary to perform the BTA of programs based on the object-oriented paradigm, using the subset of Java we had previously defined.

Thus we created a simple and yet flexible model to facilitate reasoning about BTA for object-oriented languages. It considers the binding time of objects as tuple of binding times, recording the state of the instance variables and consequently adding a degree of precision.

Furthermore, we defined the software architecture making it possible to apply the constraints to a petitCafé program, creating an equation system, using an equation solver to answer it and producing the source program annotated according to the BTA performed.

6.3 Future Work

We leave as future work a proof of correctness of the constraint system.

As well as that, a short term work that would improve this work is the extension of the prototype (i.e to include method calls and field access).

A next step would be to actually support Java by defining the forward and backward translations from Java to petitCafé.

In addition, a next step would be to improve the BTA model, for instance, in order to include flow sensitivity (via an SSA transformation), use sensitivity and polyvariance.

Then to add support for Java features such as reflection, exceptions and multithreading.

In general, and maybe one of the most interesting works, would be to apply partial evaluation to the domain of software components, using it to configure each component to a specific functionality, as well as to optimize component interaction.

6.4 Final Conclusion

Our model captures the essence of binding-time analysis for the object-oriented paradigm. Its simplicity and flexibility facilitates the creation of extensions that are needed in order to analyze realistic programs. Furthermore, our model serves as a first base for reasoning about fundamental characteristics of binding-time analysis for object-oriented programs.

Bibliography

[1] Documentation of the first public release of JSPec. http://compose.labri.fr/prototypes/jspec/doc/.

[2] L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[3] A.W. Appel. *Modern compiler implementation in Java*. Cambridge University Press, 1998.

[4] Kenichi Asai. Binding-time analysis for both static and dynamic expressions. In *Static Analysis Symposium*, pages 117–133, 1999.

[5] Frédéric Besson. Forme équationnelle de l'analyse de flot de contrôle. IRISA, Working paper, 2001.

[6] E. Börger and W. Schulte. Programmer friendly modular definition of the semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

[7] C. Consel and S.C. Khoo. Parameterized partial evaluation. *ACM Transactions on Programming Languages and Systems*, 15(3):463–493, July 1993.

[8] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[9] James Gosling et al. *The Java Language Specification Second Edition*. Addison-Wesley, 2000.

[10] L. Hornof. *Static Analyses for the Effective Specialization of Realistic Applications*. PhD thesis, Université de Rennes I, June 1997.

[11] L. Hornof and J. Noyé. Accurate binding-time analysis for imperative languages: Flow, context and return sensitivity. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 63–73, Amsterdam, The Netherlands, June 1997. ACM Press. ACM SIGPLAN Notices, 32(12).

[12] L. Hornof, J. Noyé, and C. Consel. Effective specialization of realistic programs via use sensitivity. In P. Van Hentenryck, editor, *Proceedings of the Fourth International Symposium on Static Analysis, SAS'97*, volume 1302 of *Lecture Notes in Computer Science*, pages 293–314, Paris, France, September 1997. Springer-Verlag.

[13] N.D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–503, September 1996.

[14] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice Hall, 1993.

[15] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[16] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[17] H.R. Nielson and F. Nielson. *Semantics with Applications*. Wiley Professional Computing. John Wiley & Sons, 1991.

[18] N. Oxhoj, J. Palsberg, and M.I. Schwartzbach. Making type inference practical. In *ECOOP'92 - Object-Oriented Programming - 6th European Conference*, volume 615 of *Lecture Notes in Computer Science*, pages 329–349. Springer-Verlag, 1992.

[19] F. Ployette. L'outil de résolution de système d'équations récursives reqs, 1999. http://www.irisa.fr/lande/REQS/docHtml/REQS.html.

[20] U. Schultz and C. Consel. Automatic program specialization for java. Technical report, DAIMI, University of Aarhus, December 2000.

[21] U. Schultz, J. Lawall, C. Consel, and G. Muller. Towards automatic specialization of Java programs. In R. Guerraoui, editor, *ECOOP'99 - Object-Oriented Programming - 13th European Conference*, volume 1648 of *Lecture Notes in Computer Science*, pages 367–390, Lisbon, Portugal, June 1999. Springer-Verlag.

[22] Ulrik P. Schultz. Partial evaluation for class-based object-oriented languages. *Lecture Notes in Computer Science*, 2053, 2001.

[23] U.P. Schultz. *Object-Oriented Software Engineering Using Partial Evaluation*. PhD thesis, Université de Rennes I, December 2000.